Second-order Optimization Made Practical for Deep Learning: A Preliminary Analysis

Zixuan Wang (GT ID: 903925145) [1] **Mingzheng Huang (GT ID: 903798236)** [1]

Abstract

Second-order optimization is traditionally considered as impractical for deep learning tasks due to its substantial computational demands. In this paper, we explore the feasibility of applying second-order optimization methods on shallow neural networks and address the challenge of calculating and inverting the Hessian. We initially apply Newton's method, revealing significant computational challenges. Next, we implement damping modifications and Hessian-free approach with Hessian-Vector Product (HVP) and conjugate gradient (CG) techniques for Newton's method. Finally, we apply Hessian-free Gauss-Newton method whose Hessian matrix is guaranteed to be positive semidefinite to our network.

We test our methods with MNIST dataset, and the experiments show that Hessian-free Gauss-Newton receives the best result compared to SGD, Newton's method, and Hessian-free Newton's method. Code is available at Github.

1. Introduction

Optimizing deep learning models has predominantly applied first-order optimization techniques, such as stochastic gradient descent (SGD) and its variants, due to their simplicity and scalability, particularly in handling large datasets and complex model architectures. However, first-order methods, which rely solely on gradient information, can sometimes be inefficient in navigating the complex loss landscapes of deep neural networks, often leading to suboptimal convergence rates.

Using the loss function's second derivatives or curvature information, second-order optimization techniques theoretically offer more precise navigation through these landscapes, promising faster convergence to minima. Central to these methods is the Hessian matrix, which encapsulates this curvature information. In theory, second-order methods like Newton's algorithm could significantly outperform their first-order counterparts in certain scenarios, particularly in the proximity of optima where a quadratic function can well approximate the loss surface.

Second-order methods have not been widely used in deep learning despite their potential advantages. The primary barrier has been the computational intensity of calculating and inverting the Hessian matrix, especially for models with massive parameters, such as deep neural networks. The Hessian matrix grows quadratically with the number of parameters and poses challenges in terms of memory requirements and computational complexity, making its use impractical for large-scale applications.

In this paper, we explore the viability of second-order optimization methods in deep learning. We address the critical challenge of computational demand by starting with a controlled experiment on a smaller scale. Using the MNIST dataset, we implement a basic neural network model and apply second-order optimization methods. Our experiments use the JAX library, a tool designed for high-performance numerical computations, on a NVIDIA V100 32GB GPU.

We begin by applying Newton's method and quickly confront the expected computational challenges. Building on this, we explore more feasible approaches, including damping strategies to ensure positive definiteness of the Hessian and Hessian-free methods that approximate the inverse Hessian through MVP and CG techniques. Our findings demonstrate that, with these adjustments, second-order optimization methods can be made computationally feasible and effective in training deep learning models and even outperform first-order optimization method such as SGD.

2. Related Work

Optimization techniques in deep learning have been the subject of extensive research focusing on both first- and second-order methods. Much of this research has revolved around first-order optimization methods, such as Stochastic Gradient Descent (SGD) (1) and its variants, including Adam (9) and RMSprop (15). These methods are celebrated for their simplicity and effectiveness in managing large-scale datasets and complex neural network architectures.

Despite their widespread adoption, first-order methods are not without their limitations. Bottou et al. (2) pointed out one of the critical issues is their reliance solely on gradient information. This reliance can lead to suboptimal convergence rates and inefficiencies, particularly in certain challenging scenarios. This has led researchers to explore second-order methods, which theoretically offer more precise navigation through loss landscapes and promise faster convergence to minima. Boyd et al. (3) documented the theoretical advantages of second-order methods, particularly their ability to utilize the curvature information encapsulated in the Hessian matrix. These methods are especially advantageous in scenarios where quadratic approximations are valid near optima.

However, the adoption of second-order methods in deep learning has been largely limited by computational and memory challenges, especially when dealing with large-scale tasks. Martens (10) and Pascanu et al. (13) highlighted the complexity involved in calculating and inverting the Hessian matrix, which grows quadratically with the number of parameters in a model. The research community has explored various modifications and strategies to address these computational burdens. Notable among these are Hessian-free methods (5). Pearlmutter et al. (14) applied Hessian-vector product technique to a one pass gradient calculation algorithm, a relaxation gradient calculation algorithm, and two stochastic gradient calculation algorithms, obviating any need to calculate the full Hessian. Martens (10) developed a 2nd-order optimization method based on the Hessian-free approach, and applied it to training deep auto-encoders. Without using pre-training, they obtained better results than traditional methods.

Recent advancements in the field have seen a growing interest in the practical application of second-order methods in deep learning. Martens (11) proposed K-FAC, an algorithm that can be much faster than stochastic gradient descent with momentum in practice. Grosse et al. (6) proposed Kronecker Factors for Convolution, which is similar to K-FAC, based on a structured probabilistic model for the distribution over backpropagated derivatives. Gupta et al. (7) proposed Shampoo for stochastic optimization overtensor spaces, whose runtime per step is comparable in practice to that of simple gradient methods such as SGD, AdaGrad, and Adam.

3. Methods

3.1. Newton's method

Newton's method, like gradient descent, is an iterative method that update the parameters of an objective function by computing first derivatives and second derivatives. Consider a cost function $f(w)$, the update rule for Newton's method is given by

$$w^{t+1} = w^t - \alpha(\nabla^2 f(w^t))^{-1}\nabla f(w^t) \tag{1}$$

where α is the learning rate, $g = \nabla f(w^t)$ is the first derivatives of the cost function, and $H = \nabla^2 f(w^t)$ is called Hessian matrix, whose entries are the second derivatives of the cost function.

The Hessian matrix H measures the curvature of f at a point w. If f is strictly convex, then H is always positive definite. Convex functions are very convenient to minimize because there are no spurious local optima. However, most cost functions in deep neural networks are non-convex, thus causing many stationary points during optimization. For example, if H has some positive and some negative eigenvalues, w^* would be a saddle point. If H is negative definite, $w*$ would be a local maximum.

Thus, in practice, Hessian is damped so that $H' = H + \lambda I$ for some constant $\lambda \geq 0$ (10).

3.2. Newton's method with damping

If we simply apply damping parameter to the Newton's method, the update rule would be

$$w^{t+1} = w^t - \alpha(\nabla^2 f(w^t) + \lambda I)^{-1}\nabla f(w^t) \tag{2}$$

By adding a small multiple of I we can at least guarantee that Hessian matrix is positive semidefinite at a local minimum. The damping parameter λ can be considered as controlling how conservative the approximation is by adding $\lambda\|d\|^2$ to the curvature estimate for each dimension (10). In practice, using a fixed setting of λ is not feasible because the relative scale of Hessian is changing.

Although damping can be quite effective to guarantee the positive definiteness of Hessian, it's impractical to compute and store the Hessian explicitly because its dimension is the number of parameters, which is typically in millions.

3.3. Hessian-free Newton's method

Hessian-free optimization, also known as truncated-Newton (5), has been studied for decades (12). In standard Newton's method, parameters are optimized by solving x in the linear system $Hv = g$ ($v = H^{-1}g$, then we can get $w^{t+1} = w^t - \alpha v$). Hessian-free method utilized Hessian-Vector Product and conjugate gradient instead to solve this problem.

For a $N-$dimensional vector v, Hv can be easily computed using finite differences at the cost of a single extra gradient evaluation via the identity (10)

$$Hv = \lim_{\epsilon \to 0} \frac{\nabla f(w + \epsilon v) - \nabla f(w)}{\epsilon} \tag{3}$$

which means for a scalar-valued function $f : \mathbb{R}^N \to \mathbb{R}$ with continuous second derivatives, the Hessian at a point $x \in \mathbb{R}^N$ is written as $\partial^2 f(x)$. A Hessian-vector product function is then able to evaluate

$$v \to \partial^2 f(x) \cdot v \tag{4}$$

Instead of generating the full Hessian matrix, we can think of H as the Jacobian of the gradient. We can utilize

$$\partial^2 f(x)v = \partial[x \to \partial f(x) \cdot v] = \partial g(x) \tag{5}$$

where $g(x)$ is also a scalar-valued function that dots the gradient of f at x with the vector v, whose time cost is linear in the cost of a forward pass.

A Hessian-vector product function can be useful in a conjugate gradient (CG) algorithm for minimizing convex functions. In CG, the inner solver is truncated, i.e., run for only a limited number of iterations. For truncated Newton methods to work, the inner solver needs to produce a good approximation in a finite number of iteration. In the worst case, CG will require N iterations to converge.

Conjugate gradient itself is an algorithm for the solution of linear equations (8), whose matrix is positive definite. Therefore, in our linear system $Hv = g$, we still have to guarantee that H is positive definite by adding the damping parameter. For $v^* = H^{-1}g$, it's obvious that v^* is also the minimizer for

$$f(x) = \frac{1}{2}x^T Ax - b^T x, x \in \mathbb{R}^{\mathbb{N}} \tag{6}$$

whose second derivative is $H(f(x)) = A$, and first derivative is $\nabla f(x) = Ax - b = g$. Algo. 1 shows the most commonly used algorithm of CG, which iteratively generate a set of conjugate directions and optimize along these independently. The algorithm stores only a small constant number of vectors in memory which are the same dimension as x and use a single matrix-vector product by A every iteration, which is cheap. In every iteration, x_k can be regarded as the projection of x on the Krylov subspace.

Algorithm 1 Conjugate Gradient Method

```
 1: $r_0 := b - Ax_0$
 2: **if** $r_0$ is sufficiently small **then**
        return $x_0$ as the result
 3: **end if**
 4: $p_0 := r_0$
 5: $k := 0$
 6: **repeat**
 7:     $\alpha_k := \frac{r_k^T r_k}{p_k^T A p_k}$
 8:     $x_{k+1} := x_k + \alpha_k p_k$
 9:     $r_{k+1} := r_k - \alpha_k A p_k$
10:     **if** $r_{k+1}$ is sufficiently small **then**
11:         exit loop
12:     **end if**
13:     $\beta_k := \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}$
14:     $p_{k+1} := r_{k+1} + \beta_k p_k$
15:     $k := k + 1$
16: **until** $r_{k+1}$ is sufficiently small
17: return $x_{k+1}$ as the result
```

Algo. 2 shows the skeleton of the Hessian-free optimization method. Note that here we add λv to ensure positive definiteness.

Algorithm 2 The Hessian-free optimization method

```
1: **for** $t = 1, 2, \dots$ **do**
2:     $g^t \leftarrow \nabla f(w^t)$
3:     compute/adjust $\lambda$ by some method
4:     define $H'(w^t) = H(w^t)v + \lambda v$
5:     $p^t \leftarrow CG - Minimize(H'(w^t), g^t)$
6:     $w^{t+1} \leftarrow w^t - \alpha p^t$
7: **end for**
```

3.4. Hessian-free Gauss-Newton

In practice, we rarely use Hessian of deep neural networks, even though implicit Hessian-vector product. This is because many architectures are not twice differentiable (such as ReLU), or H might have negative eigenvalues and require a large damping parameter λ.

For a loss function $L_{x,z,t}(w)$ where x is the input, z is the original network output, t is the target, w is the network parameters, the Hessian can be decomposed as

$$\nabla^2 L_{x,z,t}(w) = J_{zw}^T H_z J_{zw} + \sum_a \frac{\partial L}{\partial y_a} \nabla_w^2 z_a \tag{7}$$

where y_a is the softmax of z. Here, J_{zw} is Jacobian from network output to network parameters, $H_z = \nabla_z^2 L(z,t)$ is the output Hessian, only from the loss function of the outputs to the outputs, which is convex. The first term involves first derivatives of the network and second derivatives of the loss function; the second term involves second derivatives of the network and first derivatives of the cost function, which vanishes whenever the loss is minimized individually for each training example. This happens if every training example is fit perfectly in the case of linear regression.

The first term is denoted as

$$G = J_{zw}^T H_z J_{zw} \tag{8}$$

called Gauss-Newton Hessian. Unlike H, G requires only first derivatives of the network function, and also as long as the loss function is convex, G is guaranteed to be positive semidefinite. In practice, we can also add damping parameter for better performance.

Similar to H, it's impractical to represent G explicitly, so a simple idea is to also use matrix-vector product. Multiplying by G involves 3 steps: multiplying by J_{zw}, multiplying by H_z, multiplying by J_{zw}^T. The process is similar to be implemented in practice.

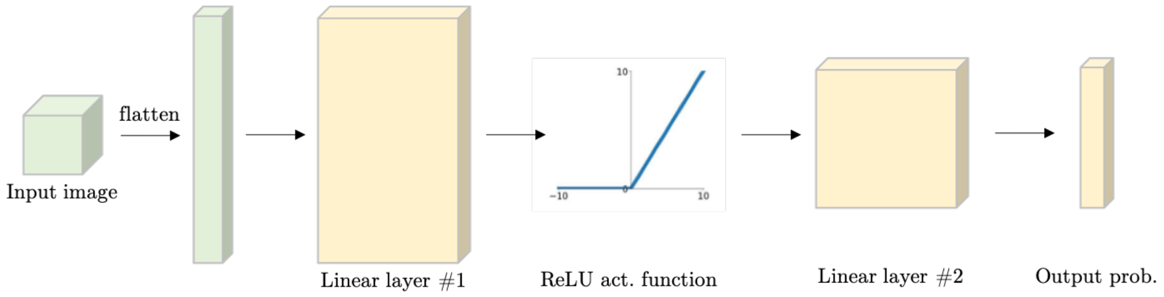


Figure 1. Network structure.

3.5. Network structure and losses

The structure of our network is shown in Fig. 1. In order to ensure that we can calculate the full Hessian of the network, we design a shallow network which have fewer parameters and reduces computational overhead. The network contains a linear layer, ReLU activation, followed by another linear layer.

The Log-Softmax function is applied to the output layer of the neural network to obtain a probability distribution over the classes. Then it is multiplied element-wise with the corresponding one-hot encoded target labels. Finally, take the negative of the product as the final loss, which can be denoted as

$$l = -\sum_{i=1}^{n} y_i \log(p_i) \tag{9}$$

where y_i is the one-hot encoded vector for the label, p_i is the predicted probability for class i obtained from the softmax output.

Table 1. Training time for every method.

Method	Time (s)
SGD	222
Newton's	21257
Damped Newton's	34933
Hessian-free Newton's	244
Hessian-free Gauss-Newton	246

4. Experiments

4.1. Training details

In our experiment, we use MNIST dataset, which has 60,000 training images and 10,000 test images. Each example is a 28×28 grayscale image of a handwritten digit from 0 to 9.

The network is trained with JAX (4) framework with a NVIDIA V100 32GB GPU. The JAX library excels for its auto-differentiation capabilities, which are crucial for implementing second-order optimization methods. JAX's capability to differentiate in forward and reverse modes allows efficient auto differentiation, which is integral for calculating derivatives and Hessians. Also enhanced by GPU acceleration, JAX's auto differentiation effectively manages Hessian-vector product calculations.

Table 2. Training loss for every method.

Method	Training loss
SGD	0.2121
Newton's	2.305
Damped Newton's	0.2837
Hessian-free Newton's	0.2614
Hessian-free Gauss-Newton	**0.2049**

We train the network for 7 epochs. The training batch size is set to 64. We record the training loss every batch and test the average accuracy at each epoch.

For SGD, we set $\alpha = 0.1$. For damped Newton's method and Hessian-free Newton's method, we both set the damping parameter $\lambda = 32$ to ensure the positive definiteness of Hessian. For Hessian-free Gauss-Newton, we set $\lambda = 4$ for better performance. We all set $\lambda = 0.5$ for second-order optimization methods.

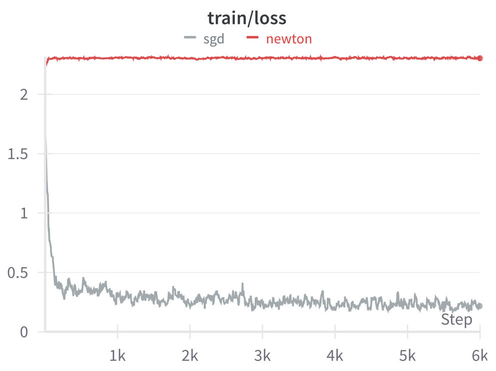
Table 3. Testing accuracy for every method.

Method	Test accuracy
SGD	0.9138
Newton's	0.1028
Damped Newton's	0.9003
Hessian-free Newton's	0.9099
Hessian-free Gauss-Newton	**0.9239**

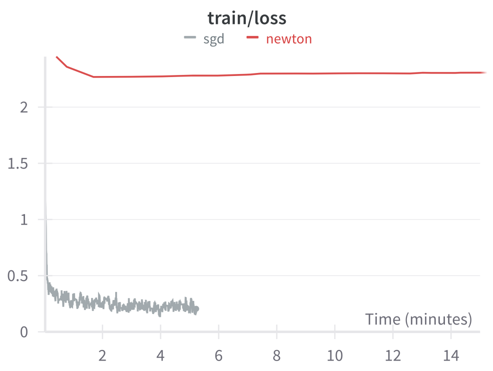
4.2. Quantitative results

4.2.1. Training time

Tab. 1 shows the training time for first-order optimization method and second-order optimization methods. We can see that native Newton's methods are quite time-consuming, proving that they are computationally expensive and are not the most practical choice for deep learning models. On the contrary, the speed of Hessian-free methods is on par with that of SGD, proving the effectiveness of our methods.

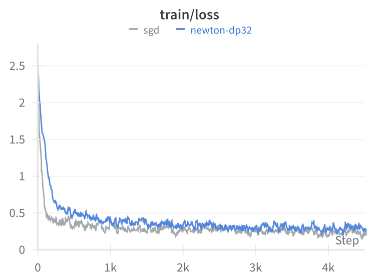


(1) Train loss for SGD and Newton based on steps.

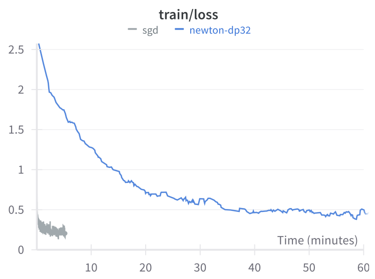


(2) Train loss for SGD and Newton based on time.

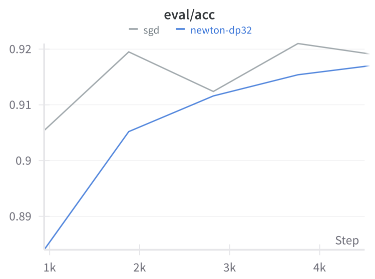
Figure 2. Learning curves for Newton's method.



(1) Train loss for SGD and damped Newton based on steps.



(2) Train loss for SGD and damped Newton based on time.



(3) Test accuracy for SGD and damped Newton.

Figure 3. Learning curves for damped Newton's method.

4.2.2. Training loss and test accuracy

Tab. 2 and Tab. 3 show the training loss and testing accuracy for every method. The best result is in bold while the second best result is underlined. The experiment shows that the Newton's method without damping doesn't converge, while the Gauss-Newton method received the best performance.

4.2.3. Learning curves

Fig. 2 shows the train loss based on steps and time separately for SGD and Newton's method. The experiment shows that the standard Newton's method doesn't converge. Meanwhile, the training time for Newton's method is much longer than that of SGD.

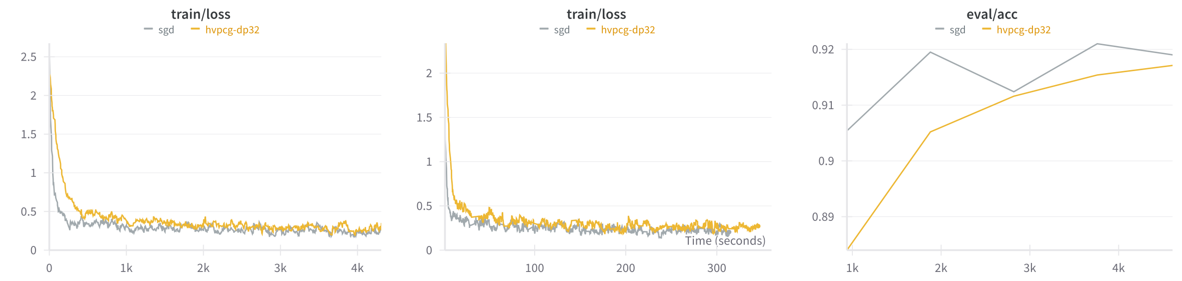
Fig. 3 shows the train loss based on steps and time separately for SGD and damped Newton's method, and the test accuracy of two methods. Both methods converge very well, but it still takes a very long time to train Newton's method. Also, the performance of Newton's method is worse than that of SGD.

Fig. 4 shows the train loss based on steps and time separately for SGD and Hessian-free Newton's method, and the test accuracy of two methods. We can see the speed of Hessian-free Newton's method is on par with that of SGD, proving the efficiency of Hessian-Vector Product and conjugate gradient algorithm.

Fig. 5 shows the train loss based on steps and time separately for SGD, Hessian-free Newton's method, and Hessian-free Gauss-Newton, and the test accuracy of three methods. The speed of Gauss-Newton is nearly the same as that of SGD, and it receives the best result among all methods.

5. Conclusion

In this paper, we have proved that second-order optimization can be made practical for the deep learning models by applying Hessian-Vector Product and Conjugate Gradient algorithm. We experiment with the SGD, standard Newton's method, damped Newton's method, Hessian-free Newton's method, and Hessian-free Gauss-Newton method. The experiments show that Hessian-free Gauss-Newton receives the best result among all methods, while also having comparable training speed with SGD.

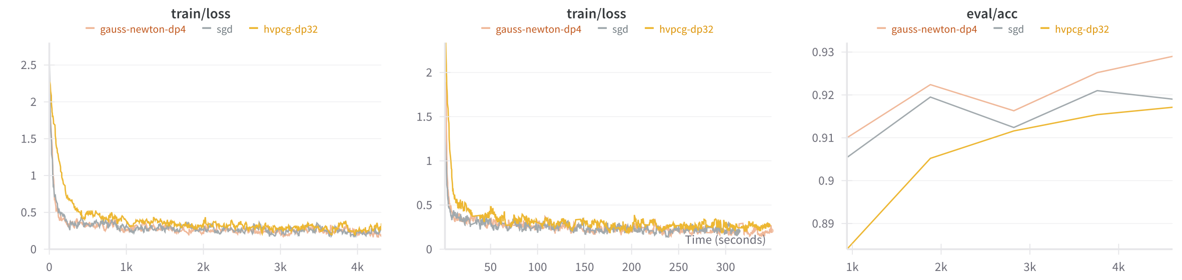


(1) Train loss for SGD and Hessian-free Newton based on steps.

(2) Train loss for SGD and Hessian-free Newton based on time.

(3) Test accuracy for SGD and Hessian-free Newton.

Figure 4. Learning curves for Hessian-free Newton's method.



(1) Train loss for SGD, Hessian-free Newton, and Hessian-free Gauss-Newton based on steps.

(2) Train loss for SGD, Hessian-free Newton, and Hessian-free Gauss-Newton based on time.

(3) Test accuracy for SGD, Hessian-free Newton, and Hessian-free Gauss-Newton.

Figure 5. Learning curves for Hessian-free Gauss-Newton.

References

[1] Léon Bottou. Stochastic gradient descent tricks. In *Neural Networks: Tricks of the Trade: Second Edition*, pages 421–436. Springer, 2012.

[2] Léon Bottou et al. Optimization methods for large-scale machine learning. *SIAM Review*, 60(2):223–311, 2018.

[3] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

[4] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs. 2018.

[5] Ron S Dembo and Trond Steihaug. Truncated-newton algorithms for large-scale unconstrained optimization. *Mathematical Programming*, 26(2):190–212, 1983.

[6] Roger Grosse and James Martens. A kronecker-factored approximate fisher matrix for convolution layers. *International Conference on Machine Learning*, 2016.

[7] Vineet Gupta, Tomer Koren, and Yoram Singer. Shampoo: Preconditioned stochastic tensor optimization. In *International Conference on Machine Learning*, pages 1842–1850. PMLR, 2018.

[8] Magnus R Hestenes, Eduard Stiefel, et al. Methods of conjugate gradients for solving linear systems. *Journal of research of the National Bureau of Standards*, 49(6):409–436, 1952.

[9] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *Conference on Learning Representations*, 2014.

[10] James Martens. Deep learning via hessian-free optimization. *International Conference on Machine Learning*, 2010.

[11] James Martens and Roger Grosse. Optimizing neural networks with kronecker-factored approximate curvature. In *International conference on machine learning*, pages 2408–2417. PMLR, 2015.

[12] J. Nocedal and S. Wright. *Numerical Optimization*. Springer Series in Operations Research and Financial Engineering. Springer New York, 2006.

[13] Razvan Pascanu and Yoshua Bengio. Revisiting natural gradient for deep networks. In *International Conference on Learning Representations*, 2013.

[14] Barak A Pearlmutter. Fast exact multiplication by the hessian. *Neural Computation*, 6(1):147–160, 1994.

[15] Fangyu Zou, Li Shen, Zequn Jie, Weizhong Zhang, and Wei Liu. A sufficient condition for convergences of adam and rmsprop. In *Proceedings of the IEEE/CVF Conference on computer vision and pattern recognition*, pages 11127–11135, 2019.