A Trace-restricted Kronecker-factored Approximation to Natural Gradient

Kaixin Gao[1*], Xiaolei Liu[1*], Zhenghai Huang[1*], Min Wang[2], Zidong Wang[2], Dachuan Xu[3†], Fan Yu[2]

[1] School of Mathematics, Tianjin University, China
[2] Central Software Institute, Huawei Technologies Co. Ltd, China
[3] Department of Operations Research and Information Engineering, Beijing University of Technology, China
[1]{gaokaixin, liuxiaolei, huangzhenghai}@tju.edu.cn, [2]{wangmin106, wang1, fan.yu}@huawei.com, [3]xudc@bjut.edu.cn

Abstract

Second-order optimization methods have the ability to accelerate convergence by modifying the gradient through the curvature matrix. There have been many attempts to use second-order optimization methods for training deep neural networks. In this work, inspired by diagonal approximations and factored approximations such as Kronecker-factored Approximate Curvature (KFAC), we propose a new approximation to the Fisher information matrix (FIM) called Trace-restricted Kronecker-factored Approximate Curvature (TKFAC), which can hold the certain trace relationship between the exact and the approximate FIM. In TKFAC, we decompose each block of the approximate FIM as a Kronecker product of two smaller matrices and scaled by a coefficient related to trace. We theoretically analyze TKFAC's approximation error and give an upper bound of it. We also propose a new damping technique for TKFAC on convolutional neural networks to maintain the superiority of second-order optimization methods during training. Experiments show that our method has better performance compared with several state-of-the-art algorithms on some deep network architectures.

Introduction

Recently, deep learning has made great progress in a host of application areas, such as computer vision and natural language processing. However, as the size of deep neural networks (DNNs) increases rapidly, more and more computational power and time are needed to train these models. Therefore, efficient algorithms are necessary for training DNNs.

Stochastic Gradient Descent (SGD) (Bottou 1991) and its extension Stochastic Gradient Descent with momentum (SGDM) (Qian 1999) are ubiquitously used for training DNNs, due to low computational cost and ease of implementation. However, SGD is a first-order optimization method and only considers first-order gradient information, which leads to some deficiencies, including sensitivity to hyperparameter settings and relatively-slow convergence. By using the curvature matrix to correct gradient, second-order optimization methods have the ability to solve these deficiencies efficiently. The most well-known second-order optimization method may be the Newton's method. Natural gradient descent (NGD) (Amari 1998), which gives the steepest descent direction in the space of distributions, also can be viewed as a second-order optimization method (Martens 2014). However, second-order optimization methods are clearly not computationally competitive with first-order alternatives, because they need to invert the large curvature matrix (the Hessian matrix for Newton's method and the Fisher Information Matrix (FIM) for NGD) whose dimension is the number of model's parameters. In practice, they require cubic computation time and quadratic storage for every update. Obviously, it is impractical to directly apply second-order optimization methods for training DNNs with hundreds of millions of parameters. So a series of approximations have been proposed.

Many methods can be viewed as diagonal approximation of the curvature matrix (Becker and Lecun 1988; John, Elad, and Yoram 2011; Tieleman and Hinton 2012; Kingma and Ba 2014; Yao et al. 2020), and these algorithms are computationally tractable. However, we know that the curvature matrix of DNNs' objective function is highly non-diagonal, and hence, diagonal approximations lose much curvature matrix information. More complex algorithms are not limited to diagonal approximations, but instead focus on some correlations between parameters of neural networks and use the non-diagonal part of the curvature matrix, such as quasi-Newton methods (Likas and Stafylopatis 2000; Keskar and Berahas 2016; Berahas, Jahani, and Takáč 2019; Goldfarb, Ren, and Bahamou 2020), Hessian-Free optimization approach (Martens 2010; Martens and Ilya 2011; Kiros 2013; Pan, Innanen, and Liao 2017), and Kronecker-factored Approximate Curvature (KFAC) (Martens and Grosse 2015; Grosse and Martens 2016; Martens, Ba, and Johnson 2018; Zhang et al. 2018; George et al. 2018). These methods have achieved advanced performance on some complicated DNN models and training tasks.

In this paper, our main focus will be on the NGD. Motivated by the diagonal approximations, we think that the diagonal elements' information of the curvature matrix plays an important role, so we pay more attention to the diagonal information and keep the trace relationship in our approximation. Inspired by both diagonal and factored approximations, we present a new approximation to the FIM of DNNs called

[*]Equal contribution
[†]Corresponding author

Trace-restricted Kronecker-factored Approximate Curvature (TKFAC) based on the quadratic form estimator proposed in (Linton and Tang 2019).

Our approximation is built by two steps. In the first step, the FIM is approximated to a block-diagonal matrix according to the layers of DNNs as KFAC (Martens and Grosse 2015). In the second step, every block matrix is decomposed into a constant multiple of the Kronecker product of two smaller matrices and kept the traces equal based on the quadratic form estimator proposed in (Linton and Tang 2019). For DNNs, we first consider TKFAC on the fully-connected layers and give Theorem 1 to compute the factors. Then, the block matrix of convolutional layers can also be decomposed efficiently under a reasonable assumption by Theorem 3. We also discuss the approximation effect of TKFAC and give an upper bound of its approximation error in Theorem 4. Next, we consider the damping technique and provide a new damping scheme for TKFAC on convolutional neural networks (CNNs). Finally, to evaluate our proposed methods, we consider two variants of TKFAC compared with SGDM, Adam, and KFAC on the deep auto-encoder problems using fully-connected neural networks (FNNs) and the image classification tasks using CNNs. Experiments show that TKFAC is an effective method.

Our contributions are summarized as follows:

- Motivated by both diagonal and factored approximations, a new approximation to the FIM called TKFAC is proposed based on a quadratic form estimator, in which the information about the sum of diagonal elements between the exact and the approximate FIM can be maintained.
- The approximation effect of TKFAC is discussed. The visualization results of approximation errors show that TKFAC is indeed an effective approximation to the FIM. Furthermore, an upper bound of TKFAC's approximation error is given and proved, which is less than the upper bound of KFAC's error in general cases. Experiments on MNIST show that TKFAC can keep smaller approximation error than KFAC during training.
- Two damping techniques are adopted for TKFAC, including the normal damping used in previous works and the new automatic tuning damping proposed in this work, which can avoid the problem that the damping is large enough to dominate the FIM and transforms the second-order optimizer into the first-order one in our experiments.
- Two variants of TKFAC, that are TKFAC_nor using the normal damping and TKFAC_new using the new damping, are compared with several state-of-the-art algorithms. TKFAC_nor has better optimization performance while has good generalization ability. TKFAC_new converges faster than baselines and TKFAC_nor, and especially it has better performance on the ResNet network.

Related Work

There have been many attempts to apply NGD or its approximations to train DNNs. The main computational challenge to use NGD is to store and invert the FIM. Recently, some works have considered the efficient Kronecker-factored approximation to the FIM, such as KFAC (Martens and Grosse 2015; Grosse and Martens 2016; Martens, Ba, and Johnson 2018; Martens and Grosse 2015; Zhang et al. 2018; Bae, Zhang, and Grosse 2018) and Eigenvalue-corrected Kronecker Factorization (EKFAC) (George et al. 2018).

KFAC was first proposed in (Martens and Grosse 2015) for FNNs, which provides a successful approximate natural gradient optimizer. KFAC starts with a block-diagonal approximation of the FIM (with blocks corresponding to entire layers), and then approximates each block as a Kronecker product of two much smaller matrices. By the property that the inverse of a Kronecker product of two matrices is equal to the Kronecker product of their inverses, each block can be inverted tractably. Therefore, NGD can be used to train DNNs efficiently. Then, KFAC is extended to CNNs (Grosse and Martens 2016), recurrent neural networks (Martens, Ba, and Johnson 2018) and variational Bayesian neural networks (Zhang et al. 2018; Bae, Zhang, and Grosse 2018) and shows significant speedups during training. George et al. (2018) tracked the diagonal variance in the Kronecker-factored eigenbasis and proposed EKFAC based on KFAC. What's more, KFAC also has been applied to large-scale distributed computing for DNNs and shows excellent experimental performance (Ba, Grosse, and Martens 2017; Osawa et al. 2019; Pauloski et al. 2020).

On the other hand, covariance matrices are of great importance in many fields and there have been many new methodological approaches to covariance estimation in the large dimensional case (i.e., the dimension of the covariance matrix is large compared with the sample size) (Ledoit and Wolf 2004; Fan and Mincheva 2011; Hoff 2016; Linton and Tang 2019). Recently, Linton and Tang (2019) proposed an estimator of the Kronecker product model of the covariance matrix called quadratic form estimator and showed that this estimator has good properties in theory. In this work, we draw inspiration from the diagonal and Kronecker-factored approximations to the FIM and propose TKFAC by means of this quadratic form estimator.

Background and Notation

In this section, we will introduce the background and give some notations.

Deep Neural Networks

Given a training dataset $D_{train} = (x, y)$ containing (input, target) examples (x, y). Define $f(x, \omega)$ to be the neural network function related to input x, where ω are the parameters. Consider the loss function $\mathcal{L}(y, f(x,\omega)) = -\log p(y|f(x,\omega))$, where p represents the density function of model's predictive distribution P. The objective function which we wish to minimize during training is the expected loss

$$h(\omega) = \mathbb{E}_{(x,y)\in D_{train}}[-\log p(y|f(x,\omega))].$$

For simplicity, throughout the rest of this paper we will use the following notation for derivatives of the loss w.r.t. some arbitrary variable V

$$\mathcal{D}V = \mathrm{d}\mathcal{L}(y, f(x,\omega))/\mathrm{d}V,\ g_l = \mathcal{D}s_l.$$

Consider a fully-connected layer of a feed-forward DNN. The computation performed in this layer can be given by

$$s_l = W_l a_{l-1}, \quad a_l = \varphi_l(s_l),$$

where $a_{l-1} \in \mathbb{R}^{m_{l-1}}$ is the input of this layer (the activation from the previous layer), $W_l \in \mathbb{R}^{m_l \times m_{l-1}}$ (the bias is ignored for convenience, which does not affect our analysis and conclusion) is the weight matrix, φ_l is the activation function, l represents the l-th layer and we refer to s_l as the pre-activation in this layer. By the chain rule, the derivatives of the weights are given by $\mathcal{D}W_l = g_l a_{l-1}^\top$.

For a convolutional layer, the pre-activation $\mathcal{S}_l \in \mathbb{R}^{n_l \times o_{l-1}}$ (for convenience, we directly give the operation expressed in matrix form for convolutional layers) is computed as

$$\begin{aligned} \widehat{\mathcal{A}_{l-1}} &= \text{im2col}(\mathcal{A}_{l-1}) \in \mathbb{R}^{n_{l-1}k_l^2 \times o_{l-1}}, \\ \mathcal{S}_l &= W_l \widehat{\mathcal{A}_{l-1}} \in \mathbb{R}^{n_l \times o_{l-1}}, \end{aligned}$$

where $\text{im2col}(\cdot)$ is a function to rearrange image blocks into columns, $\mathcal{A}_{l-1} \in \mathbb{R}^{n_{l-1} \times o_{l-1}}$ is the input of this layer, $W_l \in \mathbb{R}^{n_l \times n_{l-1}k_l^2}$ is the weight matrix, n_{l-1} and n_l are the numbers of input and output channels, o_{l-1} is the number of spatial locations and k_l is the kernel size. The the weights' gradient is computed by $\mathcal{D}W_l = \mathcal{D}\mathcal{S}_l(\widehat{\mathcal{A}_{l-1}})^\top \in \mathbb{R}^{n_l \times n_{l-1}k_l^2}$.

We will define

$$\omega = [\omega_1, \dots, \omega_L]^\top = [\text{vec}(W_1)^\top, \dots, \text{vec}(W_L)^\top]^\top$$

in the rest of this paper, which is a vector contains all the parameters of a DNN with L layers and ω_l consists all the parameters of the l-th layer.

Natural Gradient

NGD can be interpreted as a second-order optimization method (Martens 2014), in which the curvature matrix is the FIM and given by

$$F = \mathbb{E}[\mathcal{D}\omega \mathcal{D}\omega^\top] = \text{cov}(\mathcal{D}\omega, \mathcal{D}\omega),$$

where the expectation is associated with x sampled from the training distribution and y sampled according to the prediction distribution P.

The natural gradient is usually defined as $F^{-1}\nabla_\omega h$ (Amari 1998), which provides the update direction for natural gradient descent. The parameters are updated by

$$\omega \leftarrow \omega - \alpha F^{-1}\nabla_\omega h,$$

where α is the learning rate and $\nabla_\omega h$ is the gradient. More discussion of the natural gradient can refer to (Martens 2014).

Quadratic Form Estimator

Linton and Tang (2019) considered the following Kronecker product model. For a covariance matrix $\Theta \in \mathbb{R}^{n \times n}$. Let $n = n_1 \times n_2 \times \cdots \times n_r$, where $n_j \in \mathbb{Z}$ and $n_j \geq 2$ for $j \in \{1, \dots, r\}$. Suppose that

$$\Theta = \delta \times \Theta_1 \otimes \Theta_2 \cdots \otimes \Theta_r, \tag{1}$$

where $0 < \delta < \infty$ is a scalar parameter and $\Theta_j \in \mathbb{R}^{n_j \times n_j}$ satisfying $\text{tr}(\Theta_j) = n_j$ for $j \in \{1, \dots, r\}$. Based on this model, the quadratic form estimator with good theoretical properties is proposed (see for (Linton and Tang 2019)).

Our Method

Consider a DNN with L layers, the FIM can be expressed as

$$\begin{aligned} F &= \mathbb{E}[\mathcal{D}\omega \mathcal{D}\omega^\top] \\ &= \begin{pmatrix} \mathbb{E}[\mathcal{D}\omega_1 \mathcal{D}\omega_1^\top] & \cdots & \mathbb{E}[\mathcal{D}\omega_1 \mathcal{D}\omega_L^\top] \\ \vdots & \ddots & \vdots \\ \mathbb{E}[\mathcal{D}\omega_L \mathcal{D}\omega_1^\top] & \cdots & \mathbb{E}[\mathcal{D}\omega_L \mathcal{D}\omega_L^\top] \end{pmatrix}. \end{aligned}$$

Our method is started with a block-diagonal approximation to the FIM, that is

$$F = \text{diag}(F_1, F_2, \dots, F_L), \tag{2}$$

where $F_l = \mathbb{E}[\mathcal{D}\omega_l \mathcal{D}\omega_l^\top] = \mathbb{E}[\text{vec}(\mathcal{D}W_l)\text{vec}(\mathcal{D}W_l)^\top] \in \mathbb{R}^{m_{l-1}m_l \times m_{l-1}m_l}$ for any $l \in \{1, 2, \dots, L\}$. Although the FIM can be approximated to L block matrices in Eq. (2), the dimension of each block matrix F_l is still too large to invert F_l easily in practice, so further approximation is necessary.

TKFAC for Fully-connected Layers

For a fully-connected layer, the block matrix $F_l \in \mathbb{R}^{m_{l-1}m_l \times m_{l-1}m_l}$ is computed by

$$\begin{aligned} F_l &= \mathbb{E}[\mathcal{D}\omega_l \mathcal{D}\omega_l^\top] = \mathbb{E}[\text{vec}(g_l a_{l-1}^\top)\text{vec}(g_l a_{l-1}^\top)^\top] \\ &= \mathbb{E}[(a_{l-1}a_{l-1}^\top) \otimes (g_l g_l^\top)] = \mathbb{E}[\Lambda_{l-1} \otimes \Gamma_l], \end{aligned} \tag{3}$$

where $\Lambda_{l-1} = a_{l-1}a_{l-1}^\top \in \mathbb{R}^{m_{l-1} \times m_{l-1}}$ and $\Gamma_l = g_l g_l^\top \in \mathbb{R}^{m_l \times m_l}$. Then, we give our approximation to F_l by Theorem 1. According to Eq. (3), we use the simplified the model given by Eq. (1) for fully-connected layers.

Theorem 1. *Let $F_l = \mathbb{E}[\Lambda_{l-1} \otimes \Gamma_l] \in \mathbb{R}^{m_{l-1}m_l \times m_{l-1}m_l}$ and suppose that F_l can be decomposed as a Kronecker product scaled by a coefficient δ_l, i.e.,*

$$F_l = \delta_l \Phi_l \otimes \Psi_l. \tag{4}$$

Then, we have

$$\delta_l = \frac{\mathbb{E}[\text{tr}(\Lambda_{l-1})\text{tr}(\Gamma_l)]}{\text{tr}(\Phi_l)\text{tr}(\Psi_l)}, \tag{5}$$

$$\Phi_l = \frac{\text{tr}(\Phi_l)\mathbb{E}[\text{tr}(\Gamma_l)\Lambda_{l-1}]}{\mathbb{E}[\text{tr}(\Lambda_{l-1})\text{tr}(\Gamma_l)]} \in \mathbb{R}^{m_{l-1} \times m_{l-1}}, \tag{6}$$

$$\Psi_l = \frac{\text{tr}(\Psi_l)\mathbb{E}[\text{tr}(\Lambda_{l-1})\Gamma_l]}{\mathbb{E}[\text{tr}(\Lambda_{l-1})\text{tr}(\Gamma_l)]} \in \mathbb{R}^{m_l \times m_l}. \tag{7}$$

Note that in Theorem 1, $\text{tr}(\Phi_l)$ and $\text{tr}(\Psi_l)$ are unknown, but it doesn't affect the computation because we can assume that $\text{tr}(\Phi_l)$ and $\text{tr}(\Psi_l)$ are arbitrary constants. In practice, we may assume that $\text{tr}(\Phi_l) = \text{tr}(\Psi_l) = 1$ to reduce computing costs. In this case, Eq. (5)-Eq. (7) can be simplified as

$$\begin{aligned} \delta_l &= \mathbb{E}[\text{tr}(\Lambda_{l-1})\text{tr}(\Gamma_l)], \\ \Phi_l &= \frac{\mathbb{E}[\text{tr}(\Gamma_l)\Lambda_{l-1}]}{\mathbb{E}[\text{tr}(\Lambda_{l-1})\text{tr}(\Gamma_l)]}, \\ \Psi_l &= \frac{\mathbb{E}[\text{tr}(\Lambda_{l-1})\Gamma_l]}{\mathbb{E}[\text{tr}(\Lambda_{l-1})\text{tr}(\Gamma_l)]}. \end{aligned} \tag{8}$$

In the rest of this paper, we all use the simplified formulas as Eq. (8).

TKFAC for Convolutional Layers

For a convolutional layer, the block matrix can be computed by

$$F_l = \mathbb{E}[\mathrm{vec}(\mathcal{DS}_l(\widehat{\mathcal{A}_{l-1}})^\top)\mathrm{vec}(\mathcal{DS}_l(\widehat{\mathcal{A}_{l-1}})^\top)^\top], \tag{9}$$

where $F_l \in \mathbb{R}^{n_{l-1}n_l k_l^2 \times n_{l-1}n_l k_l^2}$. Note that we can't decompose Eq. (9) directly as fully-connected layers. In order to extend KFAC to convolutional layers, Grosse and Martens (2016) adopted some assumptions. Based on these assumptions, we propose an assumption for convolutional layers.

Assumption 1. Consider the products of activations $\mathcal{A}_{l-1}$ and pre-activation derivatives $\mathcal{DS}_l$, then these products are uncorrelated at any two different spatial locations.

Under this assumption, we can obtain that Eq. (9) is the sum of several Kronecker products of two smaller matrices in Theorem 2. Then, the decomposition of F_l for convolutional layers is given in Theorem 3.

Theorem 2. For the block matrix F_l of a convolutional layers defined by Eq. (9), let

$$\begin{aligned} \mathcal{DS}_l &= [\breve{u}_1, \breve{u}_2, \ldots, \breve{u}_{o_{l-1}}] \in \mathbb{R}^{n_l \times o_{l-1}}, \\ (\widehat{\mathcal{A}_{l-1}})^\top &= [\breve{a}_1, \breve{a}_2, \ldots, \breve{a}_{o_{l-1}}]^\top \in \mathbb{R}^{o_{l-1} \times n_{l-1}k_l^2}, \end{aligned}$$

where $\breve{u}$ is the vector of each column for $\mathcal{DS}_l$ and $\breve{a}$ is the vector of each column for $\mathcal{A}_{l-1}$. If Assumption 1 holds, we have

$$F_l = \sum_{i=1}^{o_{l-1}} \mathbb{E}[\breve{a}_i\breve{a}_i^\top \otimes \breve{u}_i\breve{u}_i^\top]. \tag{10}$$

Theorem 3. Let $F_l \in \mathbb{R}^{n_{l-1}n_l k_l^2 \times n_{l-1}n_l k_l^2}$ be the FIM of a convolutional layer and suppose that F_l can be decomposed as a Kronecker product scaled by a coefficient δ_l, i.e.,

$$F_l = \delta_l \Phi_l \otimes \Psi_l, \tag{11}$$

where $\mathrm{tr}(\Phi_l) = \mathrm{tr}(\Psi_l) = 1$. If Assumption 1 holds, we have

$$\begin{aligned} \delta_l &= \sum_{i=1}^{o_{l-1}} \mathbb{E}[\mathrm{tr}(\breve{a}_i\breve{a}_i^\top)\mathrm{tr}(\breve{u}_i\breve{u}_i^\top)], \\ \Phi_l &= \sum_{i=1}^{o_{l-1}} \frac{\mathbb{E}[\mathrm{tr}(\breve{u}_i\breve{u}_i^\top) \times (\breve{a}_i\breve{a}_i^\top)]}{\mathbb{E}[\mathrm{tr}(\breve{a}_i\breve{a}_i^\top)\mathrm{tr}(\breve{u}_i\breve{u}_i^\top)]} \in \mathbb{R}^{n_{l-1}k_l^2 \times n_{l-1}k_l^2}, \\ \Psi_l &= \sum_{i=1}^{o_{l-1}} \frac{\mathbb{E}[\mathrm{tr}(\breve{a}_i\breve{a}_i^\top) \times (\breve{u}_i\breve{u}_i^\top)]}{\mathbb{E}[\mathrm{tr}(\breve{a}_i\breve{a}_i^\top)\mathrm{tr}(\breve{u}_i\breve{u}_i^\top)]} \in \mathbb{R}^{n_l \times n_l}. \end{aligned} \tag{12}$$

Analysis of Approximation Error

In this subsection, we discuss the approximation error of TKFAC and give some comparisons. Firstly, we give a visualization result of TKFAC's approximation error on MNIST in Figure 1. Then, we consider the approximation error $\|F - F_{\mathrm{TKFAC}}\|_F$ in Theorem 4, where F is the exact FIM defined by Eq. (3), F_{TKFAC} is defined by Eq. (4) and $\|\cdot\|_F$ is the Frobenius norm of a matrix. We give an upper bound of TKFAC's approximation error. Finally, we compare the approximation errors of KFAC and TKFAC in experiment. For simplicity, we omit the subscript l in this subsection.

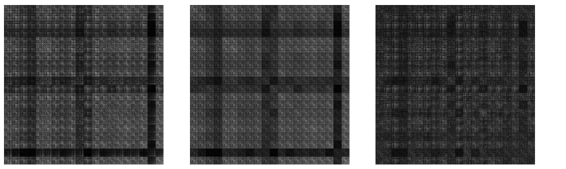


Figure 1: A comparison of the exact FIM F and our approximation F_{TKFAC}. We use TKFAC_nor to train MNIST on FNN. The network architecture is 196-20-20-20-20-10. We show the result of the FIM of the first layer with 20 units, which is a 400×400 matrix. On the left is the exact FIM F, in the middle is our approximation F_{TKFAC}, and on the right is the absolute error of these. The white level corresponding to the size of the absolute values.

Figure 1 shows the visualization results of the exact FIM F (on the left), TKFAC's approximation F_{TKFAC} (in the middle) and the absolute error of these (on the right). In Figure 1, we can see that F_{TKFAC} is very close to the exact F and the approximation error is small. Therefore, TKFAC's approximation is efficient. Next, we give an upper bound of TKFAC's approximation error.

Theorem 4. Suppose that $F^{(i)} = a^{(i)}a^{(i)\top} \otimes g^{(i)}g^{(i)\top} = \Lambda^{(i)} \otimes \Gamma^{(i)}$, $i \in \{1, 2, \ldots, N\}$ are the FIMs of different inputs (here N is the batch-size). Let F be the exact FIM defined by Eq. (3) and F_{TKFAC} be the approximate FIM in TKFAC defined by Eq. (4). We have

$$\|F - F_{\mathrm{TKFAC}}\|_F \le \frac{2(N-1)}{N} \times \max_{i<j,\ i,j \in \{1,2,\ldots,N\}} \left\{ \sqrt{\mathrm{tr}(\Lambda^{(i)})\mathrm{tr}(\Lambda^{(j)})\mathrm{tr}(\Gamma^{(i)})\mathrm{tr}(\Gamma^{(j)})} \right\}.$$

We also consider the KFAC approximation to the FIM, which is give by $F_{\mathrm{KFAC}} = \mathbb{E}[\Lambda] \otimes \mathbb{E}[\Gamma]$. Similar to the discussion of $\|F - F_{\mathrm{KFAC}}\|_F$, we can also obtain an upper bound of the the approximation error $\|F - F_{\mathrm{KFAC}}\|_F$ under the same proof process as Theorem 4.

$$\|F - F_{\mathrm{KFAC}}\|_F \le \frac{2(N-1)}{N} \times \max_{i<j,\ i,j \in \{1,2,\ldots,N\}} \left\{ \frac{(\mathrm{tr}(\Lambda^{(i)}) + \mathrm{tr}(\Lambda^{(j)}))(\mathrm{tr}(\Gamma^{(i)}) + \mathrm{tr}(\Gamma^{(j)}))}{4} \right\}.$$

Note that in general cases except that $\mathrm{tr}(\Lambda^{(i)}) = \mathrm{tr}(\Lambda^{(j)})$ and $\mathrm{tr}(\Gamma^{(i)}) = \mathrm{tr}(\Gamma^{(j)})$ for any $i, j \in \{1, 2, \ldots, N\}$, the upper bound of TKFAC is smaller than KFAC. But this does not mean that TKFAC's approximation error is smaller than KFAC. Such a result may be difficult to obtain in theory, so we compare the approximation errors of these two methods in experiment.

In Figure 2, we show the comparison of $\|F - F_{\mathrm{KFAC}}\|_F$ and $\|F - F_{\mathrm{TKFAC}}\|_F$ on MNIST, and we give the curve of the sum of each layer's error with iterations. It can be seen that $\|F - F_{\mathrm{TKFAC}}\|_F$ is smaller than $\|F - F_{\mathrm{KFAC}}\|_F$, which indicates that our approximation may be more exact than KFAC in this case.

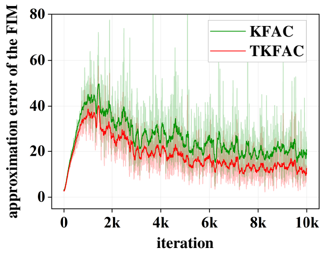


Figure 2: The curve of the sum of each layer's approximation error with iterations for KFAC and TKFAC. The model is same as described in Figure 1.

Using TKFAC for Training

To use TKFAC for training DNNs, some tricks should be employed. In this section, we mainly introduce the normal damping technique, a new damping technique for TKFAC on CNNs and the exponential moving averages. We also give the pseudocode of TKFAC in Algorithm 1.

The Normal Damping Technique

Karakida, Akaho, and Amari (2019) showed that most eigenvalues of the FIM of DNNs are close to zero, while only a small number of eigenvalues take on large values. This leads to most eigenvalues of the inverse FIM to be extremely huge or even infinite, which causes computational difficulty and inaccuracy. To make TKFAC stable, we add λI to the FIM, and we will take the fully-connected layer as an example to illustrate our damping method in the following. For the fully-connected layer, we add $\lambda I_{m_{l-1}m_l} \in \mathbb{R}^{m_{l-1}m_l \times m_{l-1}m_l}$ into $F_l =\in \mathbb{R}^{m_{l-1}m_l \times m_{l-1}m_l}$, i.e.,

$$F_l + \lambda I_{m_{l-1}m_l} = \delta_l \Phi_l \otimes \Psi_l + \lambda I_{m_{l-1}} \otimes I_{m_l}. \tag{13}$$

In order to invert $F_l + \lambda I_{m_{l-1}m_l}$ easily in computation using the properties of the Kronecker product, we approximate it by the following formula, which has been proposed in (Martens and Grosse 2015).

$$\hat{\Phi}_l = \sqrt{\delta_l}\Phi_l + \sqrt{\lambda} I_{m_{l-1}}, \quad \hat{\Psi}_l = \sqrt{\delta_l}\Psi_l + \sqrt{\lambda} I_{m_l}, \tag{14}$$

where δ_l, Φ_l and Ψ_l are defined by Eq. (8) (Eq. (12) for convolutional layers).

A New Damping Technique for CNNs

In experiments, we find that this damping technique has some limitations for convolutional layers. In Figure 3, we compare two ratios, which are the damping divided by the mean of the diagonal elements of $\sqrt{\delta_l}\Phi_l$ and $\sqrt{\delta_l}\Psi_l$ for some layers of ResNet20 when training CIFAR-10 using TKFAC. As shown in Figure 3 (a) and Figure 3 (b), we find that the damping of the convolutional layers will soon be much larger than the mean of its diagonal elements, which means that the damping may play a major role rather than the FIM shortly after the start of training. But this problem doesn't exist in the fully-connected layer as shown by the purple curves. This may limit to use the information of the FIM adequately. So we adopt the following damping technique for convolutional layers, which has good performance in our experiments. That is

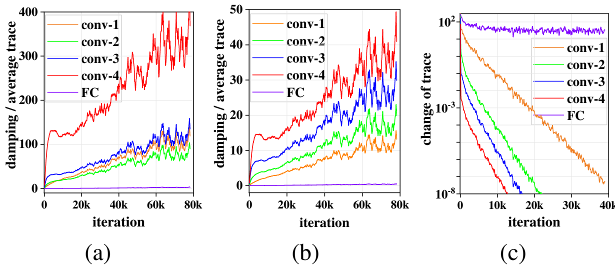


Figure 3: Some comparison results. We choose four different convolutional layers and the fully-connected layer from ResNet20 network when training CIFAR-10 using TKFAC. We record data every 100 iterations. In (a), we show the changes of $\sqrt{\lambda}/(\mathrm{tr}(\sqrt{\delta_l}\Phi_l)/(n_{l-1}k^2))$. In (b), we show the changes of $\sqrt{\lambda}/(\mathrm{tr}(\sqrt{\delta_l}\Psi_l)/n_l)$. In (c), we show the changes of traces without restricting δ_l.

$$\begin{aligned}
\tilde{\delta}_l &= \max\{\nu, \delta_l\}, \\
\tilde{\Phi}_l &= \sqrt{\tilde{\delta}_l}\Phi_l + \frac{\tilde{\delta}_l}{n_{l-1}k_l^2} I_{n_{l-1}k_l^2 \times n_{l-1}k_l^2}, \\
\tilde{\Psi}_l &= \sqrt{\tilde{\delta}_l}\Psi_l + \frac{\tilde{\delta}_l}{n_l} I_{n_l \times n_l},
\end{aligned} \tag{15}$$

where δ_l, Φ_l and Ψ_l are defined by Eq. (7) and ν is a reasonably large positive scalar.

Algorithm 1 TKFAC_nor

```
Require: learning rate $\alpha$, damping parameter $\lambda$, momentum parameter $\tau$, exponential moving average parameter $\varepsilon$, FIM and its inverse update intervals $\mathrm{T_{FIM}}$ and $\mathrm{T_{INV}}$
  $t \leftarrow 0$, $m \leftarrow 0$, Initialize $\{\hat{\Phi}_l\}_{l=1}^L$ and $\{\hat{\Psi}_l\}_{l=1}^L$
  while convergence is not reached do
    Select a new mini-batch
    for all $l \in \{1, 2, \dots, L\}$ do
      if $t \equiv 0 \pmod{\mathrm{T_{FIM}}}$ then
        Update the factors $\hat{\Phi}_l$ and $\hat{\Psi}_l$ using Eq. (17) and Eq. (18)
      end if
      if $t \equiv 0 \pmod{\mathrm{T_{INV}}}$ then
        Compute the inverses of $\hat{\Phi}_l$ and $\hat{\Psi}_l$
      end if
      Compute $\nabla_{\omega_l} h$ using backpropagation
      Compute the approximated natural gradient $(\hat{\Phi}_l^{-1} \otimes \hat{\Psi}_l^{-1})\nabla_{\omega_l} h$
      $\zeta \leftarrow -\alpha(\hat{\Phi}_l^{-1} \otimes \hat{\Psi}_l^{-1})\nabla_{\omega_l} h$
      $m \leftarrow \tau m + \zeta$ (Update momentum)
      $\omega_l \leftarrow \omega_l + m$(Update parameters)
    end for
    $t \leftarrow t + 1$
  end while
  return $\omega$
```

The reason why we restrict δ_l is that δ_l may become very small and can't promise that Φ_l and Ψ_l don't have very small eigenvalues. In Figure 3 (c), we choose the traces of the fully-connected layer and four different convolutional layers from ResNet20 when training CIFAR-10 using TKFAC with this damping technique but without restricting δ_l. We can see that the values of δ_l are very different, and some values are very small or even become close to zero. So the training becomes unstable and a constraint on δ_l is necessary. What's more, we also notice that in Figure 3 (c), δ_l of convolutional layers and the fully-connected layer vary widely. In fact, the role of ν is to expand the elements of the FIM for convolutional layers by some times. However, this expansion does not work on the fully-connected layer, because the trace of the fully-connected layer is greater than ν. In order to keep pace with convolutional layers, we also expanded the FIM of the fully-connected layer by a factor of β, where

$$\beta = \max_{l\in\{1,\dots,L\}} [\tilde{\delta}_l/\delta_l]. \tag{16}$$

Algorithm

We also use exponential moving averages as previous works, which take the new estimate to be the old one weighted by ε, plus the estimate computed on the new mini-batch weighted by $1-\varepsilon$. That is

$$\hat{\Phi}_l^{(t+1)} \leftarrow \varepsilon\hat{\Phi}_l^{(t)} + (1-\varepsilon)\hat{\Phi}_l^{(t+1)}, \tag{17}$$
$$\hat{\Psi}_l^{(t+1)} \leftarrow \varepsilon\hat{\Psi}_l^{(t)} + (1-\varepsilon)\hat{\Psi}_l^{(t+1)}, \tag{18}$$

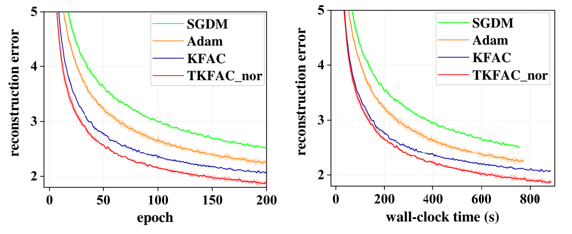
where $\hat{\Phi}_l$ and $\hat{\Psi}_l$ are computed by Eq. (14) (It's similar to $\tilde{\Phi}_l$ and $\tilde{\Psi}_l$ if we use the new damping technique) and t is the iteration. Finally, the parameters are updated

$$\omega_l^{(t+1)} \leftarrow \omega_l^{(t)} - \alpha((\hat{\Phi}_l^{(t)})^{-1} \otimes (\hat{\Psi}_l^{(t)})^{-1})\nabla_{\omega_l} h^{(t)}.$$

In this paper, we define two variants of TKFAC: TKFAC_nor and TKFAC_new (only for CNNs), in which the normal damping technique and the new damping technique are used, respectively. Algorithm 1 gives a high level pseudocode of TKFAC_nor. TKFAC_new is similar to Algorithm 1 except the differences as follows: a) We adopt the damping technique defined by Eq. (15) rather than Eq. (14), and the damping does not need to be tuned as a hyperparameter during training. b) We use a parameter ν to restrict traces given in Eq. (15), and expand the FIM of the fully-connected layer by a factor of β defined by Eq. (16).

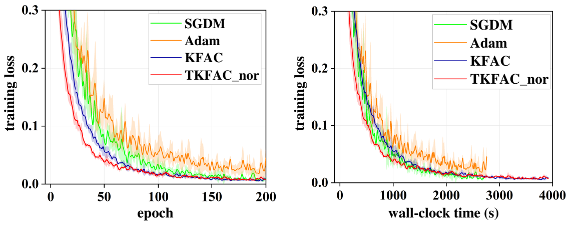
Experiments

In this section, we evaluate TKFAC's performance on the auto-encoder and image classification tasks. Our experiments mainly consist of two parts. In the first part, we focus on the effectiveness of our approximation, so we mainly consider the optimization performance of our method based on some previous tricks (TKFAC_nor) compared with other optimization methods. In the second part, we pay more attention to the effect of our new damping method and TKFAC's generalized performance, so we compare TKFAC_nor and TKFAC_new with other methods on CNNs.



(a) Reconstruction error (b) Wall-clock time

Figure 4: The curves of reconstruction error on MNIST.



(a) Training loss (b) Wall-clock time

Figure 5: The curves of training loss for CIFAR-10 on VGG16.

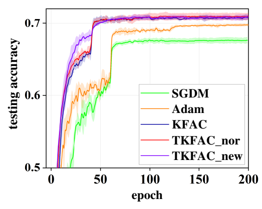
Setup

Throughout this paper, we use three different datasets, MNIST (Lecun and Bottou 1998), CIFAR-10 and CIFAR-100 (Krizhevsky, Hinton et al. 2009). We adopt a standard data augmentation scheme including random crop and horizontal flip for CIFAR-10 and CIFAR-100. For MNIST, we use an 8-layer FNN. For CIFAR-10 and CIFAR-100, we use VGG16 (Simonyan and Zisserman 2014) and ResNet20 (He et al. 2016). We choose SGDM, Adam (Kingma and Ba 2014) and KFAC (Martens and Grosse 2015) as baselines and we use batch normalization for all methods. Other related methods are also considered, but they tend to have similar or worse performance than these baselines.

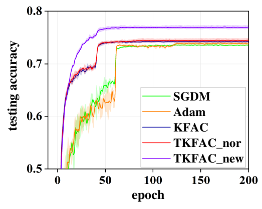
All experiments are run on a single RTX 2080Ti GPU using TensorFlow. We mainly follow the experimental setup without weight decay in (Zhang et al. 2019). The hyperparameters including the initial learning rate α, the damping parameter λ and the parameter ν are tuned using a grid search with values $\alpha \in$ {1e-4, 3e-4, . . . , 1, 3}, $\lambda \in$ {1e-8, 1e-6, 1e-4, 3e-4, 1e-3, . . . , 1e-1, 3e-1} and $\nu \in$ {1e-4, 1e-3, . . . , 10}. The moving average parameter ε and the momentum are set to 0.95 and 0.9, respectively. The update intervals are set to $T_{\text{FIM}} = T_{\text{INV}} = 100$. All experiments are run 200 epochs and repeated five times with a batch size of 500 for MNIST and 128 for CIFAR-10/100. The results are given as mean $\pm$ standard deviation.

Part I

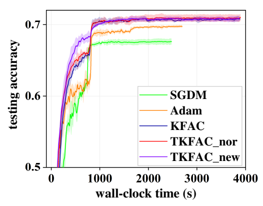
In this part, the main concern is to verify the validity of our approximation and compare different optimization methods, so TKFAC_new is not considered. Following previous work (Martens and Grosse 2015; Grosse and Martens 2016; Gold-



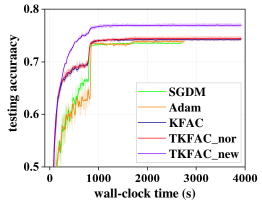
(a) VGG16-accuracy



(b) ResNet20-accuracy



(c) VGG16-time



(d) ResNet20-time

Figure 6: The curves of testing accuracy for CIFAR-100 on VGG16 and ResNet20.

farb, Ren, and Bahamou 2020), we mainly give the results of the training set. The learning rate is kept constant as George et al. (2018).

We first consider an 8-layer auto-encoder task on MNIST, which is a standard task used to benchmark optimization methods (Hinton and Salakhutdinov 2006; Martens and Grosse 2015; George et al. 2018; Goldfarb, Ren, and Bahamou 2020). Following these previous works, we report the reconstruction error on the training set and the results are shown in Figure 4. Figure 4(a) shows the curve of error with epochs throughout training. We can see that TKFAC_nor minimizes the error faster per epoch than other baselines and achieves lowest error after 200 epochs. As shown in Figure 4(b), TKFAC_nor has similar computation time (slightly more) to KFAC and it still minimizes the error faster than other baselines in terms of time.

Next, we evaluate TKFAC on CNNs. The dataset and model used here are CIFAR-10 and VGG16, respectively. Figure 5(a) shows that TKFAC_nor's training loss has a faster decline rate in the first few epochs. And all methods reach similar loss after 200 epochs except Adam. In terms of time, TKFAC_nor still has some advantages over SGDM as given in 5(b).

Part II

In this part, we pay more attention to evaluating TKFAC's generalized performance. We compare TKFAC_nor, TKFAC_new with SGDM, Adam, and KFAC on CIFAR-10 and CIFAR-100. The CNNs used here are VGG16 and ResNet20. A learning rate schedule is also used. The initial learning rate is multiplied by 0.1 every 40 epochs for KFAC/TKFAC_nor/TKFAC_new and every 60 epochs for SGDM/Adam. Experimental results are given as follows.

Figure 6 shows the curves of the testing accuracy of CIFAR-100 on VGG16 and ResNet20 in terms of epochs and time. We can see that all the second-order optimizers (KFAC, TKFAC_nor and TKFAC_new) converge faster than SGDM and Adam. The convergence rate of TKFAC_nor is slightly faster than KFAC on VGG16, but there is little difference in ResNet20. It is obvious that TKFAC_new has a faster convergence rate than KFAC, especially on ResNet20. The final testing accuracies of these methods are given in Table 1. TKFAC_nor can achieve higher average accuracy than SGDM, Adam and KFAC in all cases. Especially, TKFAC_new greatly improves the final accuracy on ResNet20 (for example, improve 2.83% than KFAC) with significantly faster convergence rate.

Dataset		CIFAR-10	CIFAR-100
VGG16	SGDM	91.77± 0.25	67.63± 0.42
	Adam	92.57± 0.31	69.74± 0.19
	KFAC	92.70± 0.39	70.84± 0.27
	TKFAC_nor	93.33± 0.20	**71.06±0.45**
	TKFAC_new	**93.42±0.11**	70.91± 0.34
ResNet20	SGDM	92.78± 0.39	73.55± 0.08
	Adam	93.57± 0.22	73.79± 0.21
	KFAC	93.69± 0.16	74.18± 0.27
	TKFAC_nor	93.76± 0.19	74.43± 0.41
	TKFAC_new	**94.66± 0.12**	**77.01± 0.34**

Table 1: Testing accuracy of CIFAR-10 and CIFAR-100.

Experimental results show that TKFAC_nor outperforms than other baselines in most cases. TKFAC_new can accelerate the convergence and improve the testing accuracy on ResNet20, which shows the effectiveness of our new damping technique, although it does not improve much on VGG16. Of course, due to the limitation of computational resource, the performance of TKFAC in large-scale problems needs to be further verified. We also want to emphasize that the new damping scheme is our preliminary attempt and it may be limited by the network structure. For other second-order optimization methods, we think that their damping may have the same problem as TKFAC, that is the damping will be large enough to dominate the curvature and turns them into first-order optimizers. So appropriate damping schemes or other techniques should be used to avoid this problem, and our approach may provide a direction.

Conclusion and Future Work

In this work, we presented a new method called TKFAC. The important property of our method is to keep the traces equal. We considered the approximation error and gave some analyses. We also discussed the damping technique, and adopted a new damping technique for TKFAC on CNNs. In experiments, we showed that TKFAC have better performance than SGDM, Adam and KFAC in most cases.

Of course, we think TKFAC can also be extended to other DNNs. We also want to explore TKFAC's performance on more datasets and more diversified network architectures. TKFAC will also be applied on MindSpore[1], which is a new deep learning computing framework. These problems are left for future work.

[1]https://www.mindspore.cn/

References

Amari, S.-I. 1998. Natural gradient works efficiently in learning. *Neural Computation* 10(2): 251–276.

Ba, J.; Grosse, R.; and Martens, J. 2017. Distributed second-order optimization using Kronecker-factored approximations. In *International Conference on Learning Representations*.

Bae, J.; Zhang, G.; and Grosse, R. 2018. Eigenvalue corrected noisy natural gradient. In *Workshop of Bayesian Deep Learning, Advances in Neural Information Processing Systems*.

Becker, S.; and Lecun, Y. 1988. Improving the convergence of back-propagation learning with second-order methods. In *Proceedings of the 1988 Connectionist Models Summer School*.

Berahas, A. S.; Jahani, M.; and Takáč, M. 2019. Quasi-Newton methods for deep learning: Forget the past, just sample. *arXiv preprint arXiv:1901.09997* .

Bottou, L. 1991. Stochastic gradient learning in neural networks. In *Proceedings of Neuro-Nmes 91*.

Fan, J.; and Mincheva, L. M. 2011. High-dimensional covariance matrix estimation in approximate factor models. *Annals of Statistics* 39(6): 3320–3356.

George, T.; Laurent, C.; Bouthillier, X.; Ballas, N.; and Vincent, P. 2018. Fast approximate natural gradient descent in a Kronecker factored eigenbasis. In *Advances in Neural Information Processing Systems*, 9550–9560.

Goldfarb, D.; Ren, Y.; and Bahamou, A. 2020. Practical quasi-Newton methods for training deep neural networks. *arXiv preprint arXiv: 2006.08877v1* .

Grosse, R.; and Martens, J. 2016. A Kronecker-factored approximate Fisher matrix for convolution layers. In *International Conference on Machine Learning*, 573–582.

He, K.; Zhang, X.; Ren, S.; and Sun, J. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, 770–778.

Hinton, G. E.; and Salakhutdinov, R. R. 2006. Reducing the dimensionality of data with neural networks. *Science* 313(5786): 504–507.

Hoff, P. D. 2016. Equivariant and scale-free Tucker decomposition models. *Bayesian Analysis* 11(3).

John, D.; Elad, H.; and Yoram, S. 2011. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research* 12(Jul): 2121–2159.

Karakida, R.; Akaho, S.; and Amari, S.-I. 2019. Pathological spectra of the Fisher information metric and its variants in deep neural networks. *arXiv preprint arXiv:1910.05992* .

Keskar, N. S.; and Berahas, A. S. 2016. ADAQN: An adaptive quasi-Newton algorithm for training RNNs. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, 1–16.

Kingma, D. P.; and Ba, J. 2014. Adam: A method for stochastic optimization. In *International Conference on Learning Representations*.

Kiros, R. 2013. Training neural networks with stochastic Hessian-free optimization. In *International Conference on Learning Representations*.

Krizhevsky, A.; Hinton, G.; et al. 2009. Learning multiple layers of features from tiny images .

Lecun, Y.; and Bottou, L. 1998. Gradient-based learning applied to document recognition. *Proceedings of the IEEE* 86(11): 2278–2324.

Ledoit, O.; and Wolf, M. 2004. A well-conditioned estimator for large-dimensional covariance matrices. *Journal of Multivariate Analysis* 88(2): 365–411.

Likas, A.; and Stafylopatis, A. 2000. Training the random neural network using quasi-Newton methods. *European Journal of Operational Research* 126(2): 331–339.

Linton, O. B.; and Tang, H. 2019. Estimation of the Kronecker covariance model by partial means and quadratic form. *arXiv preprint arXiv:1906.08908* .

Martens, J. 2010. Deep learning via Hessian-free optimization. In *International Conference on Machine Learning*, 735–742.

Martens, J. 2014. New insights and perspectives on the natural gradient method. *arXiv preprint arXiv:1412.1193v1* .

Martens, J.; Ba, J.; and Johnson, M. 2018. Kronecker-factored curvature approximations for recurrent neural networks. In *International Conference on Learning Representations*.

Martens, J.; and Grosse, R. 2015. Optimizing neural networks with Kronecker-factored approximate curvature. In *International Conference on Machine Learning*, 2408–2417.

Martens, J.; and Ilya, S. 2011. Deep learning via Hessian-free optimization. In *International Conference on Machine Learning*, 1033–1040.

Osawa, K.; Tsuji, Y.; Ueno, Y.; Naruse, A.; Yokota, R.; and Matsuoka, S. 2019. Large-scale distributed second-order optimization using Kronecker-factored approximate curvature for deep convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 12359–12367.

Pan, W.; Innanen, K. A.; and Liao, W. 2017. Accelerating Hessian-free Gauss-Newton full-waveform inversion via L-BFGS preconditioned conjugate-gradient algorithm. *Geophysics* 82(2): R49–R64.

Pauloski, J. G.; Zhang, Z.; Huang, L.; Xu, W.; and Foster, I. T. 2020. Convolutional neural network training with distributed K-FAC. *arXiv preprint arXiv:2007.00784v1* .

Qian, N. 1999. On the momentum term in gradient descent learning algorithms. *Neural Networks* 12(1): 145–151.

Simonyan, K.; and Zisserman, A. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* .

Tieleman, T.; and Hinton, G. 2012. Lecture 6.5-RMSProp: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural Networks for Machine Learning* 4(2): 26–31.

Yao, Z.; Gholami, A.; Shen, S.; Keutzer, K.; and Mahoney, M. W. 2020. AdaHessian: An adaptive second order optimizer for machine learning. *arXiv preprint arXiv:2006.00719v1* .

Zhang, G.; Sun, S.; Duvenaud, D.; and Grosse, R. 2018. Noisy natural gradient as variational inference. In *International Conference on Machine Learning*, 5847–5856.

Zhang, G.; Wang, C.; Xu, B.; and Grosse, R. 2019. Three mechanisms of weight decay regularization. In *International Conference on Learning Representations*.