THOR, Trace-Based Hardware-Driven Layer-Oriented Natural Gradient Descent Computation

Mengyun Chen[1*], Kaixin Gao[2*], Xiaolei Liu[2*], Zidong Wang[1*], Ningxi Ni[1*], Qian Zhang[3*]
Lei Chen[4†], Chao Ding[5], Zhenghai Huang[2], Min Wang[1],
Shuangling Wang[1], Fan Yu[1], Xinyuan Zhao[3], Dachuan Xu[3]

[1]Huawei Technologies Co. Ltd
[2]Tianjin University
[3]Beijing University of Technology
[4]Hong Kong University of Science and Technology
[5]Chinese Academy of Sciences
chenmengyun1, wang1, niningxi, wangmin106, wangshuangling1, fan.yu@huawei.com
gaokaixin, liuxiaolei, huangzhenghai@tju.edu.cn, zhangqian@emails.bjut.edu.cn
xyzhao, xudc@bjut.edu.cn, leichen@cse.ust.hk, dingchao@amss.ac.cn

Abstract

It is well-known that second-order optimizer can accelerate the training of deep neural networks, however, the huge computation cost of second-order optimization makes it impractical to apply in real practice. In order to reduce the cost, many methods have been proposed to approximate a second-order matrix. Inspired by KFAC, we propose a novel Trace-based Hardware-driven layer-ORiented Natural Gradient Descent Computation method, called THOR, to make the second-order optimization applicable in the real application models. Specifically, we gradually increase the update interval and use the matrix trace to determine which blocks of Fisher Information Matrix (FIM) need to be updated. Moreover, by resorting the power of hardware, we have designed a hardware-driven approximation method for computing FIM to achieve better performance. To demonstrate the effectiveness of THOR, we have conducted extensive experiments. The results show that training ResNet-50 on ImageNet with THOR only takes 66.7 minutes to achieve a top-1 accuracy of 75.9 % under an 8 Ascend 910 environment with MindSpore, a new deep learning computing framework. Moreover, with more computational resources, THOR can only takes 2.7 minutes to 75.9 % with 256 Ascend 910.

Introduction

Recently, deep learning has made significantly progress in various computer vision and natural language applications. However, with the increase of complexity of models, tons of parameters needed to be trained. For example, according to [Devlin et al. 2018] and [He et al. 2016], training BERT (over 340 million parameters) and ResNet-50 (over 23 million trainable parameters) will take around 3 days on 16 TPUv3 and 29 hours on 8 Tesla P100, respectively. Therefore, many efforts have been put to propose optimization solutions to reduce the training time.

Among all the proposed optimization techniques, the most popular and promising one is Stochastic Gradient Descent (SGD) [Robbins and Monro 1951], which is a first-order optimization algorithm. Specifically, SGD tries to minimize an objective function $J(\theta)$ with respect to the parameters θ, i.e., θ is updated as: $\theta \leftarrow \theta - \alpha \nabla_\theta J(\theta)$, where $\nabla_\theta J(\theta)$ is gradient, α represents the learning rate.

SGD is difficult to choose proper learning rate. Therefore, many variants of SGD have been introduced, such as Momentum [Qian 1999], AdaGrad [Zeiler 2012], Adam [Kingma and Ba 2014], etc. These improved optimization algorithms can use the historical information of gradient to adaptively update the parameters, making easier to adjust hyper-parameters. However, considering that the loss function of the neural network is a highly non-convex function and the curvature of loss surface is unbalanced, using second-order matrix information will speed up the convergence.

Specifically, for the second-order optimization algorithm, the parameters θ are usually updated by $\theta \leftarrow \theta - \alpha G^{-1} \nabla_\theta J(\theta)$, where G^{-1} is the inverse of second-order information matrix G. The definitions of G in different second-order optimization algorithms are not the same. Common second-order optimization algorithms include Newton's method and natural gradient method, where their second-order information matrix G is Hessian matrix (HM) and Fisher information matrix (FIM), respectively. The biggest challenge to use second-order optimizer is that its computation increases cubically and space cost increases quadratically compared to SGD. Therefore, it is quite impractical to compute the inverse of second-order information matrix directly.

To reduce the computation cost of the second-order optimizer, quite a few approximation approaches have been proposed. For instance, for Newtons method, Quasi-Newton methods [Nocedal and Wright 2006] can be used to approximate the inverse of HM. One of the advantages of these methods over the classical Newton method is that the HM does not need to be inverted explicitly. In particular, the Limited-memory BFGS (L-BGFS) algorithm [Zhu et al. 1997]has

[*]Equal contribution.
[†]Corresponding author

been implemented and used to speed up the training process in Deep Neural Networks (DNN) (e.g., [Le et al. 2011]). Other structured stochastic Quasi-Newton methods are also developed and studied recently in [Keskar and Berahas 2016, Berahas, Jahani, and Takáč 2019]. Another class of Newton type second-order methods is the Hessian Free optimization method [Martens 2010, Kiros 2013, Pan, Innanen, and Liao 2017], in which the matrix-free conjugate-gradient (CG) algorithms are used to approximate the true Hessian matrix. However, these CG algorithms usually require lots of iterations to reach the desired accuracy, in particular for ill-condition cases.

Unlike the Newton type methods, Kronecker-factored Approximate Curvature (KFAC) [Martens and Grosse 2015, Grosse and Martens 2016, Martens, Ba, and Johnson 2018] is a second-order method based on natural gradient method. More precisely, in KFAC, one computes the inverse of the FIM by computationally tractable approximations such as block-diagonal approximation and tridiagonal-block diagonal approximation. [George et al. 2018] have introduced an Eigenvalue-corrected Kronecker Factorization (EKFAC) which can approximate FIM much better than KFAC does. [Osawa et al. 2019, 2020] have demonstrated that KFAC is efficient in large-scale distributed computing for deep neural networks. Overall, among all these methods, the approximation scheme for the inverse of FIM is crucial for improving the efficiency of the second-order optimizer, since the current exact strategies still require significant computing power in practice.

To address the issues of inefficient computing FIM, in this paper, we propose an efficient approximation algorithm based on natural gradient, named Trace-based Hardware-driven layer-ORiented Natural Gradient Descent Computation (THOR), to compute FIM. Firstly, we observe from experiments that the FIM for each layer usually changes rapidly in the first few iterations and then tends to be stable. Therefore, it is reasonable to increase the update interval of the inverse of FIM in a proper manner without the loss of convergence rate. Secondly, we make further decision to decide which blocks of FIM need to be updated. Thirdly, we introduce a new approximation scheme by using a hardware-driven matrix splitting scheme to approximate the FIM, which can be regarded as finding an optimal trade-off point between the computational efficiency and the information loss of FIM.

Overall, the contributions of our work can be summarized as follows:

• Under the assumption that the FIM converges to a stationary distribution, we gradually increase the update interval of the inverse of FIM to save the overall computational time.

• Instead of using the Frobenius norm based updating rule proposed in [Osawa et al. 2019], we introduce a more computationally tractable trace-based updating rule for FIM for each layer.

• We approximate the block diagonal matrix based on KFAC to a smaller matrix by splitting matrix dimensions, which trade the loss of FIM for efficient computation.

• Last but not the least, with THOR, we are able to train ResNet-50 on ImageNet in 66.7/4.1/2.7 minutes with a top-1 accuracy of 75.9 % using 8/128/256 Ascend 910 on MindSpore. Furthermore, part of our algorithm has been open sourced [1], and the code will continue to be improved in the future.

Algorithm 1 THOR

Require: $T_{\mathrm{FIM}}, T_{\mathrm{INV}}$: FIM and its inverse matrix update intervals
Require: ω_1, ω_2: two positive threshold parameters used in Eq. (9)
Require: `size`: the split dimension of FIM
Require: α: the learning rate
Require: λ: the damping
Require: l: the number of the networks layers
  $k \leftarrow 0$
  while convergence is not reached **do**
    for $i; i \leq l; i++$ **do**
      if $k \equiv 0 \pmod{T_{\mathrm{FIM}}}$ **then**
        Update the factors A_{i-1} and G_i
      end if
      if $k \equiv 0 \pmod{T_{\mathrm{INV}}}$ **then**
        Compute Δ^k using Eq. (8)
        if Δ^k is updated according to Eq. (9) **then**
          Using `size` to split the factors A_{i-1} and G_i according to Eq. (10)
          Update the inverse of split matrix $\hat{A}_{i-1}^{-1}$ and $\hat{G}_i^{-1}$
        end if
      end if
      $\theta_i^{(k+1)} \leftarrow \theta_i^{(k)} - \alpha((\hat{A}_{i-1}^{(k)} + \lambda I)^{-1} \otimes (\hat{G}_i^{(k)} + \lambda I)^{-1}) \nabla_{\theta_i} J^k$
    end for
    $k \leftarrow k + 1$
  end while

  return θ

Background and Notations

The purpose of deep neural network training is to find a set of model parameters $\theta \in \mathbb{R}^n$ to minimize the loss function $J(\theta)$. Given the cross-entropy loss function:

$$J(\theta) = \mathbb{E}[-\log p(y|x, \theta)], \tag{1}$$

where x, y are the training input and label, $p(y|x, \theta)$ represents the density function of a predictive distribution $\mathbb{P}_{y|x}$.

The Natural Gradient

Our algorithm is based on the natural gradient proposed by [Amari 1998]. Natural gradient gives the steepest direction of the target function when the parameter space has a Riemannian metric structure. In other words, it gives the largest change of the loss function per unit change of the model. The distance between the distribution $\mathbb{P}_\theta$ and $\mathbb{P}_{\theta+\delta\theta}$ can be measured with the K-L divergence. More recent discussion of the natural gradient can be found in [Martens 2020, Ollivier et al. 2017]. Natural gradient is typically defined as $F^{-1}\nabla_\theta J(\theta)$,

[1] THOR: https://gitee.com/mindspore/mindspore/tree/master/model_zoo/official/cv/resnet_thor.

where $F \in \mathbb{R}^{n\times n}$ is FIM. With the predictive distribution defined as $\mathbb{P}_{y|x}$, FIM is formulated as

$$F = \mathbb{E}[\nabla_\theta \log p(y|x,\theta) \nabla_\theta \log p(y|x,\theta)^T], \tag{2}$$

It is impractical to compute the inverse of FIM directly in a deep neural network since it has over millions of parameters.

KFAC

KFAC is an efficient method for approximation natural gradient, which approximates FIM by block-diagonal or block-tridiagonal matrices. Based on nice motivation and rigorous mathematical derivation, it has exquisitely settled the problem of complex computation for inverting the second order information matrix. [Osawa et al. 2019] have proved that block-diagonal KFAC has good results in large-scale DNN and block-diagonal KFAC computes more efficiently than block tridiagonal. Thus, we focus on block-diagonal KFAC to approximate FIM in this work.

KFAC is a two-step approximation method. In the first step, KFAC decomposes the FIM into block matrices according to the layers of the neural network, by assuming that parameters of different layers are independent. Then the calculation of the inverse of FIM is simplified as the inverse of these small blocks. In the second step, these block matrices are further approximated by the Kronecker product of two much smaller matrices which we call Kronecker factors. Since the inverses of the Kronecker product of two matrices are equal to the Kronecker product of their inverses, and these two smaller matrices are easier to calculate and invert than calculating and inverting the entire block matrix. KFAC greatly simplifies FIM calculation.

Consider a deep neural network with l layers and denote the outputs of the i-th layer as s_i, the inputs of the i-th as a_{i-1} which is the activations of previous layer and θ_i is a weight vector of i-th layer.

In the first step, KFAC approximates FIM into block matrix:

$$\begin{aligned} F &\approx \text{diag}(F_1, F_2, ..., F_l) \\ &= \text{diag}\left(\mathbb{E}[D_{\theta_1} D_{\theta_1}^T], \mathbb{E}[D_{\theta_2} D_{\theta_2}^T], \cdots, \mathbb{E}[D_{\theta_l}, D_{\theta_l}^T]\right), \end{aligned} \tag{3}$$

where $D_\theta = -\frac{\mathrm{d}\log p(y|x,\theta)}{\mathrm{d}\theta}$.

In the second step, each block of FIM can be rewritten as

$$\begin{aligned} F_i &= \mathbb{E}[D_{\theta_i} D_{\theta_i}^T] = \mathbb{E}[a_{i-1}a_{i-1}^T \otimes g_i g_i^T] \\ &\approx \mathbb{E}[a_{i-1}a_{i-1}^T] \otimes \mathbb{E}[g_i g_i^T] = A_{i-1} \otimes G_i, \end{aligned} \tag{4}$$

where $\otimes$ denotes the Kronecker product, $g_i = D_{s_i}$, $A_{i-1} = \mathbb{E}[a_{i-1}a_{i-1}^T]$ and $G_i = \mathbb{E}[g_i g_i^T]$. Since $(A \otimes B)^{-1} = A^{-1} \otimes B^{-1}$ for any matrices A and B, we can compute the block-diagonal FIM easily as

$$F_i^{-1} = (A_{i-1} \otimes G_i)^{-1} = A_{i-1}^{-1} \otimes G_i^{-1}, \tag{5}$$

Furthermore, KFAC uses a damping technique in [Martens and Grosse 2015] by adding λI to the Kronecker factors. Finally, the weight vector θ_i with i-th layer can be updated as follows:

$$\theta_i^{(k+1)} \leftarrow \theta_i^{(k)} - \alpha((A_{i-1}^{(k)} + \lambda I)^{-1} \otimes (G_i^{(k)} + \lambda I)^{-1}) \nabla_{\theta_i} J^{(k)}, \tag{6}$$

where α represents the learning rate.

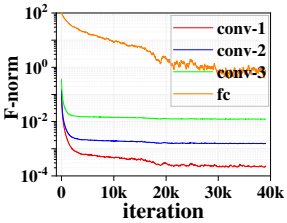


Figure 1: Changes of the Frobenius norm for FIM. We choose the fully-connected layer and three different convolution layers when training the CIFAR-10 dataset on ResNet-18 using KFAC. We record data every 20 iterations.

THOR

As mentioned in Section Introduction, although KFAC could accelerate convergence, it still has no advantage on the overall training time compared with first-order optimizer due to the high computation cost of Kronecker product. To address this problem, we propose a novel algorithm called Trace-based Hardware-driven layer-ORiented Natural Gradient Descent Computation (THOR). In THOR, we first use a gradually increasing update interval for updating the inverse of FIM. Second, instead of updating the whole inverse of FIM, we further determine to update matrix blocks which are guided by trace-based rules. Finally, by combining with the hardware performance, we trade a little loss of FIM for efficient approximating the matrix blocks. The detailed steps of the THOR optimizer are given in Algorithm 1.

Update with Trace Constraint

In order to reduce the computation and achieve faster training, KFAC and its variants all reduce the frequency of computing the FIM and its inverse matrix [Martens and Grosse 2015, Grosse and Martens 2016, George et al. 2018, Zhang et al. 2018, Osawa et al. 2019]. They update the FIM and its inverse matrix every few iterations. In particular, [Osawa et al. 2019] discussed the change rate of the FIM on the ResNet-50 network for ImageNet classification, and adopted a heuristic scheme. They further reduced the update rate after 500 iterations to accelerate training. However, the fixed update is not highly profitable in the later stage of training. In other words, the update times in the later training are still very large, which costs many computing resources but can't greatly improve the training effect. Therefore, we propose a new updating scheme in this subsection.

Figure 1 illustrates the changes of the Frobenius norm for FIM at each layer. We can clearly observe that the FIM for each layer changes rapidly in the first few iterations and then tends to be stable. Based on existing research [Martens and Grosse 2015, Grosse and Martens 2016, Osawa et al. 2019] and our experiments, it is reasonable to assume $\{F^k\}_{k=1}^n$ as a Markov process converging to a stationary distribution π, where F^k represents the FIM updated at the k-th iteration. Under this assumption, we can gradually increase the update interval of the FIM and its inverse matrix during training. However, as shown in Figure 1, some layers tend to stabilize faster than others, it is too rough to set the same update

split dimension	1	16	32	64	128	256	512	1024	2048
matrix number ($L < 1\%$)	4	4	5	9	13	23	32	46	48
performance (μs)	35	59	83	121	212	534	1418	3738	9824
normalized loss of matrix	0.0741	0.0741	0.0926	0.1667	0.2407	0.4259	0.5926	0.8519	0.8889
normalized performance	1	0.6014	0.4276	0.2939	0.1669	0.0664	0.0250	0.0095	0.0036

Table 1: The data about split dimension on ResNet-50 with Ascend 910

interval for all blocks of FIM. Therefore, it is more reasonable to select which blocks of FIM need to be updated. Moreover, we can stop updating the FIM and its inverse matrix for each layer if the FIM becomes stable. For example, if we stop updating the FIM after the k-th iteration for the i-th layer, then the parameters will be computed by $\theta_i^{(k+t)} = \theta_i^{(k+t-1)} - \alpha(F_i^{(k)})^{-1}\nabla_{\theta_i} J^{(k)}$, $t = 0, 1, 2, \cdots$.

To determine whether to update or stop updating, we shall introduce an adaptive trace-based updating rule. In [Osawa et al. 2019], the Frobenius norm $\|\cdot\|_F$ is used to estimate the changes of FIM for each layer, which does not have good scalability and may not suitable for large-scale tasks. However, it is well-known that for any matrix X, the relationship of its Frobenius norm $\|X\|_F$ and nuclear norm $\|X\|_*$ can be expressed as follows:

$$\|X\|_F \leq \|X\|_* \leq \sqrt{r}\|X\|_F, \tag{7}$$

where $r = \mathrm{rank}(X)$ and $\|\cdot\|_*$ is the nuclear norm [Recht, Fazel, and Parrilo 2010, Srebro, Rennie, and Jaakkola 2005] of a matrix(i.e., the sum of singular values of a matrix). It is well-known that for any matrix X, the trace of the absolute value of matrix X $|\mathrm{tr}(X)|$ is also smaller or equal the nuclear norm $\|X\|_*$ and the equality holds if X is a positive semidefinite matrix. Therefore, $|\mathrm{tr}(X)|$ can also be used to estimate the changes of FIM for each layer. More importantly, the computational cost of $|\mathrm{tr}(X)|$ is linear which means it has much better scalability. Therefore, in THOR, for the i-th layer, we further define the following relative change rate:

$$\Delta^k = \frac{\big||\mathrm{tr}(F_i^k + \lambda I)| - |\mathrm{tr}(F_i^{k-1} + \lambda I)|\big|}{|\mathrm{tr}(F_i^{k-1} + \lambda I)|}, \tag{8}$$

Then, we adopt the following trace-based updating scheme of FIM and its inverse for each layer based on the above relative change rate Δ^k:

$$\begin{cases} \text{update } F_i^k, & \text{if } \Delta^k \in (\omega_1, +\infty) \\ \text{do not update } F_i^k \text{ and set } F_i^k = F_i^{k-1}, & \text{if } \Delta^k \in [\omega_2, \omega_1] \\ \text{stop update } F_i^k \text{ and set } F_i^{k+t} \equiv F_i^{k-1} \text{ for all } t = 1, 2, \ldots & \text{if } \Delta^k \in [0, \omega_2) \end{cases} \tag{9}$$

where ω_1 and ω_2 are two given positive threshold parameters.

In Figure 2 and Figure 3, we demonstrate the changes of Δ^k of some layers on two different networks. It can be seen clearly that Δ^k is relatively large at the beginning, and then fluctuates around a relatively fixed small value after a few iterations. For most layers, Δ^k lies in the interval $(0.001, 0.01)$ and fluctuates around 0.001 for some layers. Therefore, we

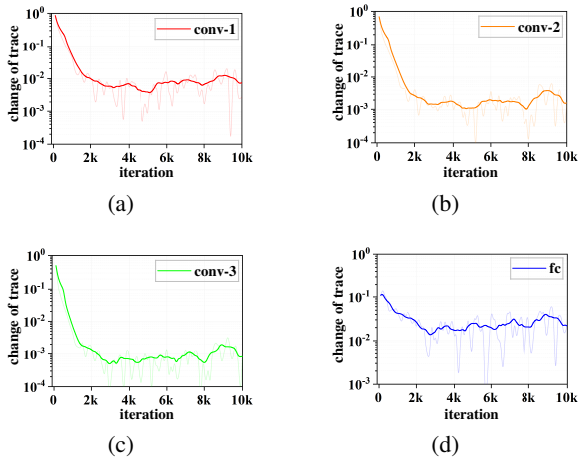


Figure 2: Change rate vs iterations on ResNet-18. We choose three different convolution layers and the fully-connected layer when training the CIFAR-10 dataset using K-FAC. We record data every 20 iterations.

provide a recommendation of the choices $\omega_1 = 0.01$ and $\omega_2 = 0.001$, which have performed well for training, confirmed by experiments in Section Experiments. We believe that it is reasonable to increase the update interval of F_i if $\Delta^k \in [0.001, 0.01]$, and stop updating F_i if $\Delta^k \in [0, 0.001)$.

Hardware-driven Matrix Split

Due to the huge number of parameters existed in the deep neural networks, the computation of the inverse of Kronecker factors matrix is still very costly ($O(l \cdot n^3)$, where l is number of the network layers and n is the typical dimension of the Kronecker factors). To achieve better performance, we need to make a further approximation of FIM. As FIM can be treated as a covariance matrix over the gradient D_θ of the loss function, which is defined in Eq. (3). TONGA [Roux, Manzagol, and Bengio 2008] makes a block-diagonal approximation to FIM by assuming independence between the neurons of a neural network. Similarly, KFAC [Martens and Grosse 2015] treats D_{θ_i} is more "important" to itself than D_{θ_j} does, where $j \neq i$, which implies that the diagonal blocks contain more information for the current layer. Therefore, in KFAC, one can approximate FIM by using a block-diagonal matrix in a given layer.

In order to compute FIM more efficiently, in our algorithm,

	Momentum	KFAC	THOR	THOR_stop	THOR_NT
Best Test Acc	94.31%	94.42%	95.00%	95.09%	94.40%
Time Per Epoch	13.29s	65.01s	17.64s	17.16s	18.09s
Time(93%)	809.51s	1704.261s	656.84s	622.92s	670.95s
Time(94%)	889.154s	4032.43s	1139.11s	1092.24s	1155.28s
Time(95%)	NaN	NaN	1555.54s	1350.72s	NaN

Table 2: The computational result of ResNet-18 on CIFAR-10

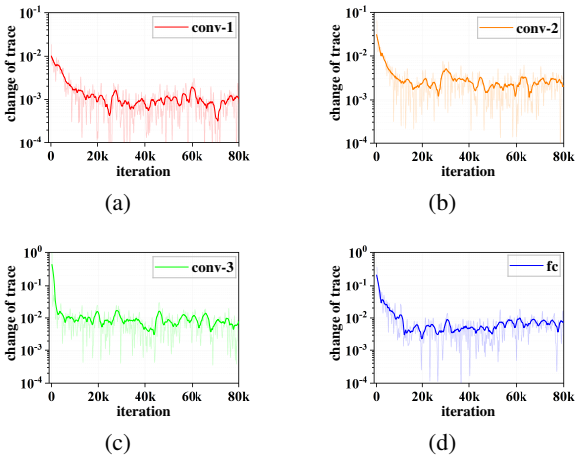


Figure 3: Change rate vs iterations on ResNet-50. We choose three different convolution layers and the fully-connected layer when training the ImageNet dataset using K-FAC. We record data every 200 iterations.

we further split the input of the i-th layer's into j groups vectors, i.e., $a_{(i-1)1}, a_{(i-1)2}, ..., a_{(i-1)j}$ and assume that different groups $a_{(i-1)s}$ and $a_{(i-1)t}$ are independent, where $s \neq t$. As a consequence, the outputs of the i-th layer's block split, denoted as $s_{i1}, s_{i2}, \ldots, s_{ij}$, are also independent. Under the independent assumption, we can approximate the Kronecker factors A_{i-1} and G_i for computing the i-th FIM block F_i by the following block diagonal matrices:

$$\begin{aligned}\hat{A}_{i-1} &\approx \text{diag}\left(\mathbb{E}[a_{(i-1)1}a_{(i-1)1}^T], \mathbb{E}[a_{(i-1)2}a_{(i-1)2}^T],\right. \\ &\quad \left.\ldots, \mathbb{E}[a_{(i-1)j}a_{(i-1)j}^T]\right), \\ \hat{G}_i &\approx \text{diag}\left(\mathbb{E}[D_{s_{i1}}D_{s_{i1}}^T], \mathbb{E}[D_{s_{i2}}D_{s_{i2}}^T], \ldots, \mathbb{E}[D_{s_{ij}}D_{s_{ij}}^T]\right) \\ &\approx \text{diag}\left(\mathbb{E}[g_{i1}g_{i1}^T], \mathbb{E}[g_{i2}g_{i2}^T], \ldots, \mathbb{E}[g_{ij}g_{ij}^T]\right).\end{aligned} \tag{10}$$

In Figure 4, we compare the difference between the K-FAC block-diagonal approximation $\tilde{F}$ (Figure 4(a)) and the proposed splitting approximation $\hat{F}$ (Figure 4(b)). We calculate the errors of two approximations $\tilde{F}$ and $\hat{F}$, which are around 5% after 10 iterations. Interestingly, the relative difference between two approximations $\tilde{F}$ and $\hat{F}$ reduces to 1% after 50 iterations. One possible reason is that the independence assumption is more likely to be satisfied when the proportion of element value on the diagonal block increases. Obviously,

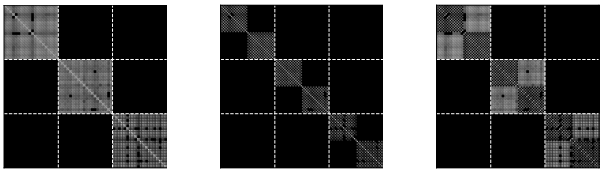


Figure 4: A comparison between the KFAC block-diagonal $\tilde{F}$ and Hardware-driven split matrix $\hat{F}$. We use the deep neural network to train MNIST for 10 iterations. The network architecture is 768-20-20-20-10, in which the middle three layers trained with FIM matrix. The dashed line indicate the separation by layers. (a) is the figure of $\tilde{F}$, (b) is the figure of $\hat{F}$ which split dimension is 10, (c) is the absolute error between (a) and (b).

the smaller the split dimension, the less time cost on computation (better efficiency), but the larger information loss compared to original matrix. Therefore, the group number j is a trade-off between the information loss and computation efficiency.

The processes of calculating the information loss (loss of matrix) and the computation efficiency (performance) are given as follows.

Loss of matrix. First, we set the tolerable information loss to 1% which means the split matrix contains 99% information of the original Kronecker factors. The information loss L is measured by the spectral norm, which is defined as follows:

$$L = 1 - \sqrt{\frac{\lambda_{\max}(\hat{A}\hat{A}^T)}{\lambda_{\max}(AA^T)}}, \tag{11}$$

where $\lambda_{\max}(\cdot)$ is the largest eigenvalue of the matrix, A is the original matrix, and $\hat{A}$ is the split matrix. Second, we count the number of the matrices whose information loss L is below 1% in each predefined split dimension. Finally, these counts are normalized by dividing the total number of matrices.

Performance. We measure the time it costs to invert the matrix of each shape in the predefined split dimensions on the Ascend 910/Tesla V100. And the normalized performance of a specific split dimension is defined as follows:

$$\text{normalized}_n = \frac{p_1}{p_n}, \tag{12}$$

where normalized_n is the normalized performance of a specific split dimension n, p_1 is the performance data of the matrix with split on the fist dimension.

		learning rate					damping	
		α_{warmup}	α_{target}	e_{warmup}	e_{end}	p_{decay}	λ_0	ρ_{decay}
BS=256	THOR	-	0.045	-	70	6	0.03	0.87
BS=256	THOR_stop	-	0.050	-	70	6	0.03	0.87
BS=256	THOR_NT	-	0.045	-	72	6	0.03	0.87
BS=4096	THOR	0.005	0.45	5	55	6	0.3	0.2
BS=8192	THOR	0.01	0.8	5	48	6	0.6	0.3

Table 3: Hyper-parameters of our methods on ImageNet

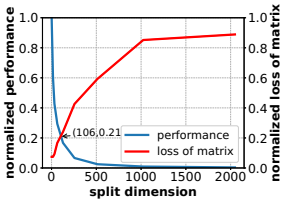


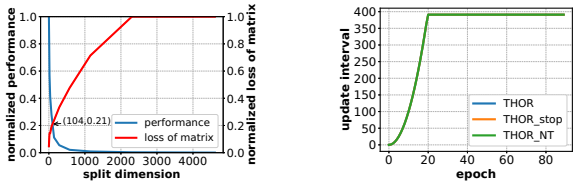
Figure 5: The trade-off between loss of matrix and performance of Ascend 910. In this experiment, the matrix is the Kronecker factor A from ResNet-50 and the split dimension list is [1, 16, 32, 64, 128, 256, 512, 1024, 2048]. Normalized performance 1 represents the best performance while 0 represents the worst one. And for normalized loss of matrices, 1 represents the maximum loss of the original matrix and 0 represents the minimum loss.

For example, on ResNet-50 with Ascend 910, we set split dimension list as [1, 16, 32, 64, 128, 256, 512, 1024, 2048] and the total number of Kronecker factors A is 54. The relevant data are reported in Table 1. Figure 5 plots the normalized data in Table 1. We can find the intersection point is (106, 0.21), which represents the trade-off between the computation efficiency and the loss of the matrix. Thus, we choose the split dimension as 128 which is the closest point to the intersection point in the split dimension list.

Experiments

To test the performance, we apply THOR to train ResNet-18 for CIFAR-10 and ResNet-50 for ImageNet. In these experiments, we implement our method in three variants: THOR, THOR_stop with early stopping and THOR_NT without trace-based updating rule. We have compared THOR, THOR_stop and THOR_NT with KFAC and Momentum on CIFAR-10. However, we only compared our methods with Momentum on ImageNet, since KFAC cannot finish the training in a reasonable time on large model. For example, KFAC takes 2s to calculate the FIM inversion on Tesla V100 while THOR only takes 200ms. Please note that we didn't compare to Adam because Adam fails to obtain the accuracy of Momentum for ResNet-50, the highest accuracy achieved by Adam is 73.48% [You et al. 2019]

For all our experiments, we average the results of 5 runs and we use a normal distribution to generalize the starting points.



(a) (b)

Figure 6: The hyper-parameters of training ResNet-18 on CIFAR-10. (a) The split dimension list is [1, 9, 18, 36, 72, 144, 288, 576, 1152, 2304, 4608], we set split dimension as 72. (b) The same update interval of FIM on THOR, THOR_stop and THOR_NT.

CIFAR-10

Setup. In this experiment, we use pytorch on 1 Tesla v100 and train ResNet-18 on CIFAR-10 with batch-size 128. Split dimension and the update interval can be found in Figure 6. And we set the same learning rate for Momentum, KFAC, THOR, THOR_stop and THOR_NT and same damping for KFAC, THOR, THOR_stop and THOR_NT. The learning rate $\alpha^e$for e epoch and the damping $\lambda^{(e)}$ are defined as follows:

$$\begin{aligned} \alpha^{(e)} &= 0.1 \times 10^{-\lfloor \frac{e}{30} \rfloor}, \\ \lambda^{(e)} &= 0.3 \times 10^{-\lfloor \frac{e}{30} \rfloor}. \end{aligned} \tag{13}$$

where $\lfloor \cdot \rfloor$ means the floor function. The weight decay for Momentum, KFAC, THOR, THOR_stop and THOR_NT is set to 0.0005. The trace thresholds are set to $(\omega_1, \omega_2) = (0.01, 0)$ for THOR, $(\omega_1, \omega_2) = (0.01, 0.001)$ for THOR_stop and $(\omega_1, \omega_2) = (0, 0)$ for THOR_NT. The update interval for KFAC is set to 20.

Results. Figure 7 (a) shows that THOR, THOR_stop, THOR_NT and KFAC converge faster than Momentum in the first 30 epochs, and all of them are able to reach high train accuracy. It can be seen from Figure 7 (c) that THOR, THOR_stop, THOR_NT and KFAC are faster than Momentum in the first 30 epochs, and second-order algorithms are able to achieve higher test accuracy than Momentum. In particular, THOR can reach 95% test accuracy in this experiment. Figure 7 (b) shows that, our methods outperform KFAC, but have no advantage compared to Momentum in terms of the training loss. However, for test accuracy, THOR is 152.67s faster, THOR_NT is 138.56s faster and THOR_stop is 186.59s faster than Momentum with 93% test accuracy and the summary of computational results can be seen from Table 2. Note

	Momentum	THOR	THOR_stop	THOR_NT
Best Test Acc	76.04%	75.92%	75.92%	76.00%
Time Per Epoch	90.00s	102.15s	100.05s	103.65s
Time(74.9%)	6569.86s	3674.88s	3405.26s	3747.74s
Time(75.9%)	7020.98s	4083.20s	4004.47s	4148.03s

Table 4: The computational result of ResNet-50 on ImageNet

that in this experiment, for the second-order methods, we use the same learning rate α as that of Momentum. After adjusting the parameters, we can get better results. For instance, THOR_stop is 435s faster than Momentum when reaching 93% test accuracy by tuning learning rate.

In addition, we also did experiments on EKFAC which needs to obtain the eigenvalues of the second-order matrix. Therefore, its inversion is based on eigendecomposition, while THOR and KFAC are based on Cholesky factorization which is faster than eigendecomposition. Thus, compared with EKFAC, the advantage of THOR is more obvious that THOR_stop is 3809s faster than EKFAC with 93% test accuracy.

Furthermore, we did ablation study to see how frequency updating strategy, trace-based updating rule and matrix split affect the results on ResNet-18 + CIFAR-10 and respectively named each improved algorithm as THOR_tr, THOR_fre and THOR_sp. Our study showed that THOR_tr which accelerated 65% compared to original KFAC algorithm on 90 epochs, while THOR_fre and THOR_sp respectively accelerated 48.5% and 40.5% compared to original KFAC algorithm on 90 epochs. THOR_sp gained lower acceleration since ResNet-18's fisher information matrix is much smaller than other models.

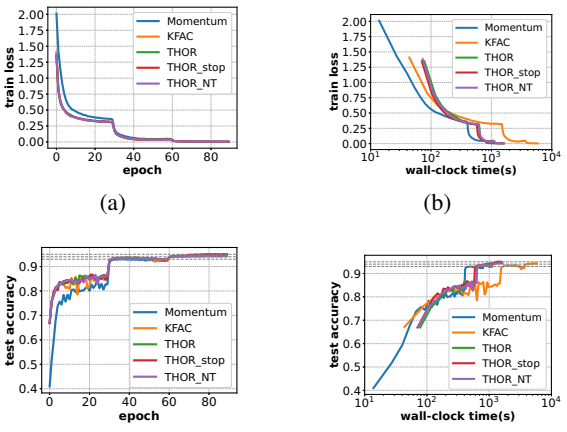


Figure 7: ResNet-18 on CIFAR-10. (a) The training loss with epoch. (b) The training loss with wall-clock time. (c) The test accuracy with epoch. (d) The test accuracy with wall-clock time.

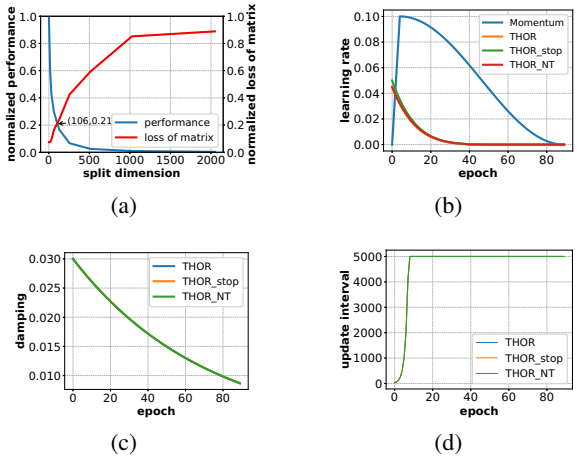


Figure 8: The hyper-parameters of training ResNet-50 on ImageNet. (a) The split dimension list is [1, 16, 32, 64, 128, 256, 512, 1024, 2048], we set split dimension as 128. (b) Learning rate on Momentum, THOR, THOR_stop and THOR_NT. (c) Damping on THOR, THOR_stop and THOR_NT. (d) Same update interval on THOR, THOR_stop and THOR_NT.

ImageNet

Setup. In this experiment, we implement THOR on MindSpore with 8 Ascend 910 and train ResNet-50 on ImageNet with batch-size 256. The weight decay for these methods is set to 0.0005 and the label smoothing is set to 0.1. The trace thresholds are set to $(\omega_1, \omega_2) = (0.01, 0)$ for THOR, $(\omega_1, \omega_2) = (0.01, 0.001)$ for THOR_stop and $(\omega_1, \omega_2) = (0, 0)$ for THOR_NT. Split dimension, learning rate, damping and update interval can be found in Figure 8. The learning rate $\alpha^{(e)}$ for e epoch is determined as follows:

$$\alpha^{(e)} = \alpha_{target} \cdot (1 - \frac{e}{e_{end}})^{p_{decay}}, \tag{14}$$

where α_{target} is the target learning rate, e_{end} is the end of decay epoch, p_{decay} is the decay rate. Figure 9 shows the impact of target learning rate and decay rate on reaching the test accuracy after 40 epochs with batch-size 256. For the larger batch size, the warmup strategy makes the training result better. The specific strategy is given as follows:

$$\begin{cases} \alpha^{(e)} = \alpha_{warmup} + \frac{\alpha_{target} - \alpha_{warmup}}{e_{warmup}} \cdot e, & \text{if } e \le e_{warmup} \\ \alpha^{(e)} = \alpha_{target} \cdot (1 - \frac{e - e_{warmup}}{e_{end}})^{p_{decay}}, & \text{if } e > e_{warmup} \end{cases} \tag{15}$$

where α_{warmup} is the initial learning rate, e_{warmup} is the end of the warm-up.

	Hardware	Software	Batch size	Optimizer	Time	Accuracy
[He et al. 2016]	Tesla P100 × 8	Caffe	256	SGD	29 hr	75.3%
[Goyal et al. 2017]	Tesla P100 × 256	Caffe2	8192	SGD	60 min	76.3%
Google 0.7-2 [https://mlperf.org]	NVIDIA V100 × 8	TensorFlow	2496	LARS	88.56 min	75.9%
[Osawa et al. 2020]	Tesla V100 × 128	Chainer	4096	SP-NGD	32.5min	74.8%
[Osawa et al. 2020]	Tesla V100 × 256	Chainer	8192	SP-NGD	16.9min	75.3%
our work	Ascend 910 × 8	MindSpore	256	THOR	66.7min	75.9%
our work	Ascend 910 × 128	MindSpore	4096	THOR	4.1min	75.9%
our work	Ascend 910 × 256	MindSpore	8192	THOR	2.7min	75.9%

Table 5: The result of large batchsize of ResNet-50 on ImageNet

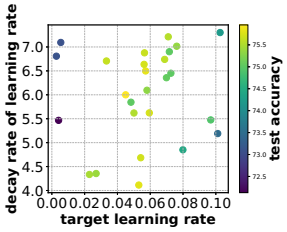


Figure 9: The learning rate of ResNet-50 on ImageNet.

The damping λ adopts the following decreasing rule:

$$\lambda^{(e)} = \lambda_{(0)} \cdot \rho_{decay}^{(\frac{e}{10})}, \tag{16}$$

where $\lambda_{(0)}$ is the initial damping, ρ_{decay} is the decay rate of the damping.The hyper-parameters for our methods are shown in Table 3.

Results. Figure 10 (a)(c) show that the convergence speed of THOR, THOR_NT and THOR_stop are faster than Momentum. Momentum needs 78 epochs to converge while THOR, THOR_NT, THOR_stop only needs 40 epochs; In the Figure 10(b)(d), our methods take less time than Momentum, specifically, THOR needs 68.1min,THOR_NT needs 69.1min and THOR_stop only takes 66.7min to converge while Momentum needs 117min, the results show in Table 4. And THOR is also competitive in the sense of end-to-end training time with various batch sizes, it takes 4.1min/2.7min to reach test accuracy 75.9% with batch-size 4096/8192 in Table 5.

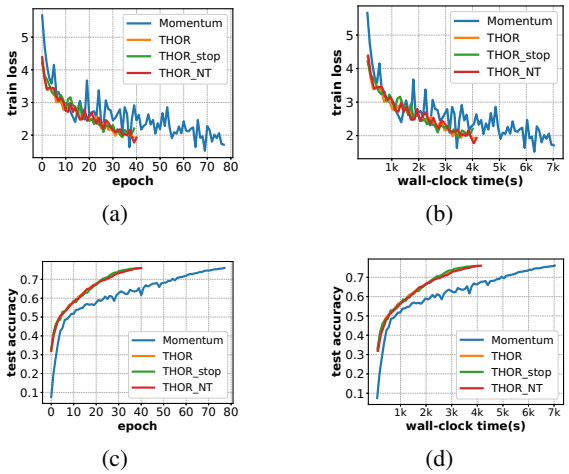


Figure 10: ResNet-50 on ImageNet. (a) The training loss with epoch. (b) The training loss with wall-clock time. (c) The test accuracy with epoch. (d) The test accuracy with wall-clock time.

Related Work

Second-order optimizer could accelerate convergence but the computational complexity of the inverse of FIM is $O(n^3)$ (where n is the dimension of FIM). Therefore, various approximations of the second-order information matrix have been proposed in recent years. KFAC [Martens and Grosse 2015] approximates the FIM as two much smaller matrices based on network structure and Kronecker products. However, it still requires a lot of computing power and does not have ideal scalability which is crucial for large-scale tasks. EKFAC [George et al. 2018] tried to solve this problem by using more accurate eigenvalues to reduce approximate error than KFAC, but its inversion is based on eigendecomposition which makes EKFAC slower than KFAC per step. More recently, [Osawa et al. 2019, 2020] implemented an improved KFAC on ResNet-50 for ImageNet with powerful computational resources (1024 Tesla V100). In terms of the wall-clock time, the result is quite promising (it takes 5.5min to achieve a top-1 accuracy of 75.4% on ResNet-50 for ImageNet). In our work, the proposed methods are more efficient, we train ResNet-50 on ImageNet to 75.9% in 2.7 minutes with 256 Ascend 910. Moreover, we are able to achieve a top-1 accuracy of 75.9% in 66.7 minutes with much less computational resources (8 Ascend 910) than [Osawa et al. 2019, 2020].

Conclusion

In this paper, we propose the THOR to speed up the convergence. This algorithm assumes FIM will converge to a stationary distribution and uses the trace of matrix block to increase the update interval of matrix blocks, and makes a more radical approximation to the matrix block. The experiments on CIFAR-10 and ImageNet clearly demonstrate that THOR can converge much faster than Momentum. Especially on the ImageNet, THOR's overall time is much less than that of Momentum. THOR only uses 66.7 minutes to converge with 8 Ascend 910, which only takes half the time of Momentum. In the future, we will apply THOR to other deep learning models to speed up their training time, such as BERT [Devlin et al. 2018] and GPT-2 [Radford et al. 2019].

References

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

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* .

Devlin, J.; Chang, M.-W.; Lee, K.; and Toutanova, K. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* .

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.

Goyal, P.; Dollár, P.; Girshick, R.; Noordhuis, P.; Wesolowski, L.; Kyrola, A.; Tulloch, A.; Jia, Y.; and He, K. 2017. Accurate, large minibatch SGD: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677* .

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.

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. Springer.

Kingma, D. P.; and Ba, J. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* .

Kiros, R. 2013. Training neural networks with stochastic Hessian-free optimization. *arXiv preprint arXiv:1301.3641* .

Le, Q. V.; Ngiam, J.; Coates, A.; Lahiri, A.; Prochnow, B.; and Ng, A. Y. 2011. On optimization methods for deep learning. In *International Conference on Machine Learning*, 265–272.

Martens, J. 2010. Deep learning via Hessian-free optimization. In *International Conference on Machine Learning*, volume 27, 735–742.

Martens, J. 2020. New Insights and Perspectives on the Natural Gradient Method. *Journal of Machine Learning Research* 21: 1–76.

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.

Nocedal, J.; and Wright, S. 2006. *Numerical optimization*. Springer Science & Business Media.

Ollivier, Y.; Arnold, L.; Auger, A.; and Hansen, N. 2017. Information-geometric optimization algorithms: A unifying picture via invariance principles. *The Journal of Machine Learning Research* 18(1): 564–628.

Osawa, K.; Tsuji, Y.; Ueno, Y.; Naruse, A.; Foo, C.; and Yokota, R. 2020. Scalable and Practical Natural Gradient for Large-Scale Deep Learning. *IEEE Transactions on Pattern Analysis and Machine Intelligence* .

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.

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

Radford, A.; Wu, J.; Child, R.; Luan, D.; Amodei, D.; and Sutskever, I. 2019. Language models are unsupervised multitask learners. *OpenAI Blog* 1(8): 9.

Recht, B.; Fazel, M.; and Parrilo, P. A. 2010. Guaranteed minimum-rank solutions of linear matrix equations via nuclear norm minimization. *SIAM Review* 52(3): 471–501.

Robbins, H.; and Monro, S. 1951. A stochastic approximation method. *The Annals of Mathematical Statistics* 400–407.

Roux, N. L.; Manzagol, P.-A.; and Bengio, Y. 2008. Topmoumoute online natural gradient algorithm. In *Advances in Neural Information Processing Systems*, 849–856.

Srebro, N.; Rennie, J.; and Jaakkola, T. S. 2005. Maximum-margin matrix factorization. In *Advances in Neural Information Processing Systems*, 1329–1336.

You, Y.; Li, J.; Reddi, S.; Hseu, J.; Kumar, S.; Bhojanapalli, S.; Song, X.; Demmel, J.; Keutzer, K.; and Hsieh, C.-J. 2019. Large Batch Optimization for Deep Learning: Training BERT in 76 minutes. In *International Conference on Learning Representations*.

Zeiler, M. D. 2012. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701* .

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

Zhu, C.; Byrd, R. H.; Lu, P.; and Nocedal, J. 1997. Algorithm 778: L-BFGS-B: Fortran subroutines for large-scale bound-constrained optimization. *ACM Transactions on Mathematical Software (TOMS)* 23(4): 550–560.