Rich Information is Affordable: A Systematic Performance Analysis of Second-order Optimization Using K-FAC

Yuichiro Ueno
ueno.y.ai@m.titech.ac.jp
Tokyo Institute of Technology
AIST-Tokyo Tech RWBC-OIL, AIST
Tokyo, Japan

Kazuki Osawa
oosawa.k.ad@m.titech.ac.jp
Tokyo Institute of Technology
Tokyo, Japan

Yohei Tsuji
tsuji.y.ae@m.titech.ac.jp
Tokyo Institute of Technology
Tokyo, Japan

Akira Naruse
anaruse@nvidia.com
NVIDIA
Tokyo, Japan

Rio Yokota
rioyokota@gsic.titech.ac.jp
Tokyo Institute of Technology
AIST-Tokyo Tech RWBC-OIL, AIST
Tokyo, Japan

ABSTRACT

Rich information matrices from first and second-order derivatives have many potential applications in both theoretical and practical problems in deep learning. However, computing these information matrices is extremely expensive and this enormous cost is currently limiting its application to important problems regarding generalization, hyperparameter tuning, and optimization of deep neural networks. One of the most challenging use cases of information matrices is their use as a preconditioner for the optimizers, since the information matrices need to be updated every step. In this work, we conduct a step-by-step performance analysis when computing the Fisher information matrix during training of ResNet-50 on ImageNet, and show that the overhead can be reduced to the same amount as the cost of performing a single SGD step. We also show that the resulting Fisher preconditioned optimizer can converge in 1/3 the number of epochs compared to SGD, while achieving the same Top-1 validation accuracy. This is the first work to achieve such accuracy with K-FAC while reducing the training time to match that of SGD.

CCS CONCEPTS

• **Computing methodologies** → *Neural networks.*

KEYWORDS

information matrix; distributed training; performance optimization

ACM Reference Format:
Yuichiro Ueno, Kazuki Osawa, Yohei Tsuji, Akira Naruse, and Rio Yokota. 2020. Rich Information is Affordable: A Systematic Performance Analysis of Second-order Optimization Using K-FAC. In *26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '20), August 23–27, 2020, Virtual Event, USA*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3394486.3403265

KDD '20, August 23–27, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7998-4/20/08…$15.00
https://doi.org/10.1145/3394486.3403265

1 INTRODUCTION

Rich information from first and second-order derivatives such as the Hessian, Jacobian, generalized Gauss-Newton matrix, Fisher information matrix (FIM), and the gradient's (non-central) covariance matrix (also known as empirical Fisher) have many potential applications in both theoretical and practical problems in deep learning. For example, information criteria that characterizes the generalization of deep learning models involves the computation of the Hessian, covariance, and FIM [22]. Predicting optimal hyperparameters involves the Hessian and covariance matrix [15]. Other examples include structured pruning through the use of eigenvalues of the Hessian [26], approximate Bayesian inference through the use of the FIM [12], and natural gradient descent which also depends on the FIM [2]. Therefore, if we can somehow reduce the enormous computational overhead of calculating these large matrices, this could facilitate significant advances in generalization, hyperparameter prediction, pruning, continual learning, and optimization of deep neural networks. It is worth noting that these different applications require the computation of these matrices at different frequencies, so the requirement on the overhead of computing them also differs for each application. For example, if one only requires the Hessian for characterizing the generalization after the training, it only needs to be computed once. On the other hand, if one wants to use the FIM to precondition the first-order gradient during training, it needs to be computed for every parameter update. In the present work, we will tackle this most challenging application of using the FIM to precondition the gradient, *i.e.* natural gradient descent. Our goal is not to propose the natural gradient descent as an alternative optimizer to stochastic gradient descent (SGD), but rather to use this optimizer as a benchmark to demonstrate our capability to calculate the FIM with minimum overhead. If we can compute the second-order information during training with sufficient accuracy and minimum overhead, it would meet the requirements of all other use cases of the second-order information.

Martens *et al.* [14] have demonstrated that the FIM can be approximated through layer-wise block-diagonalization and Kronecker factorization while reducing the computational cost by orders of magnitude. Osawa *et al.* have shown that data-parallel distributed computing can further reduce the overhead of computing the FIM

Table 1: Step-by-step effect of performance optimizations on time per K-FAC update for training ResNet-50 on ImageNet-1K classification. Osawa *et al.* 's result [19] is on 1024 GPUs, and other results are with 128 GPUs (NVIDIA Tesla V100.)

	Distrib. FIM inverse Sec. 3.3.1	Symm. comm. of FIM Sec. 3.3.2	float21 comm. of FIM Sec. 3.3.3	Overlap comm. Sec. 4.1	Hier. Comm. Sec. 4.1	Stale FIM Sec. 4.2	SGD Time/upd.	K-FAC Time/upd.	K-FAC/SGD ratio
Osawa *et al.* [19]	✓	✓				✓[1]	-	340 ms	-
Tsuji *et al.* [23]	✓	✓		✓	✓		85 ms	315 ms	3.71
Osawa *et al.* [18]	✓	✓		✓	✓		-	236 ms	-
	✓	✓		✓	✓	✓	-	178 ms	-
This work	✓						**57** ms	251 ms	4.40
	✓	✓						199 ms	3.49
	✓	✓	✓					182 ms	3.19
	✓	✓	✓	✓				221 ms[2]	3.88
	✓	✓	✓	✓	✓			192 ms	3.37
	✓	✓	✓	✓	✓	✓		**108** ms	**1.89**

Table 2: Time per SGD update for distributed training of ResNet-50 on ImageNet-1K classification.

	Hardware	Software	Time/upd.
Goyal *et al.*[5]	Tesla P100	Caffe2	255 ms
You *et al.*[29]	KNL	Intel Caffe	341 ms
Akiba *et al.*[1]	Tesla P100	Chainer	255 ms
Jia *et al.*[10]	Tesla P40	TensorFlow	220 ms
Tsuji *et al.*[23]	Tesla V100	Chainer	85 ms
Ying *et al.*[28]	TPU v3	TensorFlow	**37** ms
Mikami *et al.*[17]	Tesla V100	NNL	**57** ms
Yamazaki *et al.*[27]	Tesla V100	MXNet	**50** ms
This work (SGD)	Tesla V100	Chainer	**57** ms

[19]. Moreover, contrary to prior belief, they showed that K-FAC can achieve the same Top-1 validation accuracy as the other leader-board momentum SGD methods for training ResNet-50 on ImageNet-1K classification. They also showed that this accuracy can be obtained in one-third of the number of epochs (35 epochs) compared to SGD (90 epochs). However, the time per K-FAC update was still about an order of magnitude larger for K-FAC, causing the total training time to reach 10 min. on 1024 GPUs, whereas other SGD methods were able to train 90 epochs in 1.2 min. on 2048 GPUs [27]. To accelerate K-FAC update, Tsuji *et al.* [23] first analyzed the computational and communication bottlenecks required for K-FAC training and proposed some performance optimizations that were evaluated on 128 GPUs. Combining Tsuji *et al.* 's performance optimizations with more aggressive approximations to the FIM, Osawa *et al.* [18] further reduced the total time for training ResNet-50 on ImageNet-1K classification with K-FAC to 5.5 min. on 1024 GPUs.

Their efforts have indeed advanced practical second-order information computation even in large-scale deep learning. Still, there is room for improving these studies in terms of the scientific understanding of the actual overhead of second-order information computation for various neural networks. For example, as described in Table 2, the time per SGD update varies widely depending on hardware and software implementation, and this makes it less clear how much overhead the computation of the second-order information has compared to merely computing the first-order gradients. To better understand the actual overhead, we introduce a systematic performance analysis of second-order optimization using K-FAC for training ResNet-50 on ImageNet-1K classification.

Our contributions are:

(1) We show a step-by-step performance optimization of SGD training of ResNet-50 on ImageNet-1K, which includes mixed-precision training and reduction of memory access and CPU overhead (Table 2). We show how one can achieve state-of-the-art throughput by starting from a standard baseline.
(2) We perform a systematic performance analysis of K-FAC and compare its performance to state-of-the-art SGD (Table 1). We also introduce a custom 21-bit floating point format for collective communication among GPUs (Figure 6), which is only necessary for communicating Kronecker factors that require slightly higher precision than 16-bit.
(3) We reduce the total training time of ResNet-50 on ImageNet-1K dataset using K-FAC to 2 min. on 2048 GPUs (Table 3). By using stale Fisher Information Matrix, we reduce K-FAC overhead to 1.89x compared to our state-of-the-art SGD implementation.

Since the distributed training of ResNet-50 on ImageNet has been a popular leader-board benchmark for quite some time, our reference of comparison could not be more challenging. The fact that we can reduce the training time of K-FAC to match that of SGD while also matching the Top-1 validation accuracy not only debunks the claim that natural gradient descent is prone to overfitting, but also, for the first time, demonstrate that second-order information can be computed with negligible overhead even during the training of fairly large models. The present work only calculates the gradient's non-central covariance matrix (also known as empirical Fisher) with Kronecker factorization, but techniques like KFRA and KFLR [3] allow the same Kronecker factorization framework to be extended to the exact FIM — which is equivalent to the generalized Gauss-Newton matrix for several loss functions in deep learning. In the Kronecker factorization framework, every kind of second-order information computation shares the same dominant time cost in communicating and inverting Kronecker factors, not in building it.

[1] They skipped the FIM update at a constant frequency.
[2] Overlap has a bad effect without Hier. Comm. For details, please refer to section 4.1.

2 PRELIMINARIES

2.1 Mini-batch Stochastic Learning

Let us consider a classification problem, where $\boldsymbol{x}$ is the training sample, $\boldsymbol{t}$ is a one-hot vector of the corresponding label, and the dataset is defined as a pair of the two $\mathcal{D} = \{(\boldsymbol{x}, \boldsymbol{t})\}$. For a mini-batch $\mathcal{B} \subset \mathcal{D}$, the cost function $\mathcal{L}$ can be written as

$$\mathcal{L}(\mathcal{B}; \boldsymbol{\theta}) = \frac{1}{|\mathcal{B}|} \sum_{(\boldsymbol{x},\boldsymbol{t})\in\mathcal{B}} \ell(f(\boldsymbol{x}; \boldsymbol{\theta}), \boldsymbol{t}) \tag{1}$$

where ℓ is the sample-wise loss function.

Stochastic Gradient Descent minimizes the above cost function using its gradient $\nabla\mathcal{L}$

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \eta\nabla\mathcal{L}(\mathcal{B}; \boldsymbol{\theta}^{(t)}) \tag{2}$$

where the mini-batch $\mathcal{B}$ is chosen randomly from the entire dataset $\mathcal{D}$, and η is the learning rate.

2.2 Natural Gradient Descent

When natural gradient descent (NGD) [2] is used instead of first order methods such as stochastic gradient descent (SGD), the gradient is preconditioned by multiplying the inverse of the FIM.

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \eta\boldsymbol{F}^{-1}\nabla\mathcal{L}(\boldsymbol{\theta}^{(t)}) \tag{3}$$

where the FIM $\boldsymbol{F}$ is defined as

$$\boldsymbol{F} = \mathop{\mathbb{E}}_{\boldsymbol{x}\sim q}\left[\mathop{\mathbb{E}}_{\boldsymbol{y}\sim p_{\boldsymbol{\theta}}}\left[\nabla \log p_{\boldsymbol{\theta}}(\boldsymbol{y}|\boldsymbol{x})\nabla \log p_{\boldsymbol{\theta}}(\boldsymbol{y}|\boldsymbol{x})^T\right]\right] \tag{4}$$

where $p_{\boldsymbol{\theta}}(\boldsymbol{y}|\boldsymbol{x})$ is the output of softmax. To reduce the amount of computation, we approximate the FIM with the empirical FIM.

$$\boldsymbol{F} \approx \mathop{\mathbb{E}}_{(\boldsymbol{x},\boldsymbol{t})\in\mathcal{B}}\left[\nabla \log p_{\boldsymbol{\theta}}(\boldsymbol{t}|\boldsymbol{x})\nabla \log p_{\boldsymbol{\theta}}(\boldsymbol{t}|\boldsymbol{x})^T\right] \tag{5}$$

The number of elements in the FIM grows quadratically against the number of parameters, and the cost of computing the inverse of the FIM grows cubically. Therefore, the storage and computation of the FIM becomes infeasible even for CNNs of moderate size.

2.3 K-FAC approximations

In order to reduce the storage and computational cost of the FIM, Kronecker-factored approximate curvature (K-FAC) [14] uses the following approximations.

(1) Layer-wise block-diagonal approximation.
By assuming that the correlation of the parameters between layers are weak, we can approximate the FIM in a block-diagonal form

$$\boldsymbol{F} \approx diag(\boldsymbol{F}_1, \boldsymbol{F}_2, \cdots, \boldsymbol{F}_L) \tag{6}$$

where its inverse can be calculated by inverting each block separately

$$\boldsymbol{F}^{-1} \approx diag(\boldsymbol{F}_1^{-1}, \boldsymbol{F}_2^{-1}, \cdots, \boldsymbol{F}_L^{-1}). \tag{7}$$

(2) Kronecker factorization.
For a fully connected layer, the pre-activation can be calculated from the activation of the previous layer as

$$\boldsymbol{s}_l = \boldsymbol{W}_l\boldsymbol{a}_{l-1}. \tag{8}$$

where $\boldsymbol{s}_l \in \mathbb{R}^{d_l}, \boldsymbol{a}_{l-1} \in \mathbb{R}^{d_{l-1}}, \boldsymbol{W}_l \in \mathbb{R}^{d_l\times d_{l-1}}$, and d_l is the dimension of the l-th layer. The bias is not explicitly shown here in order to simplify the discussion that follows.

The Kronecker factorization of the FIM for the l-th layer can be written as

$$\begin{aligned}\boldsymbol{F}_l &= \mathop{\mathbb{E}}_{(\boldsymbol{x},\boldsymbol{t})\in\mathcal{B}}\left[(\nabla_{\boldsymbol{s}_l}\log p_{\boldsymbol{\theta}}(\boldsymbol{t}|\boldsymbol{x}))\boldsymbol{a}_{l-1}) \otimes (\nabla_{\boldsymbol{s}_l}\log p_{\boldsymbol{\theta}}(\boldsymbol{t}|\boldsymbol{x}))\boldsymbol{a}_{l-1})^T\right] \\ &\approx \boldsymbol{B}_l \otimes \boldsymbol{A}_{l-1}\end{aligned} \tag{9}$$

$$\boldsymbol{B}_l = \mathop{\mathbb{E}}_{(\boldsymbol{x},\boldsymbol{t})\in\mathcal{B}}\left[(\nabla_{\boldsymbol{s}_l}\log p_{\boldsymbol{\theta}}(\boldsymbol{t}|\boldsymbol{x}))(\nabla_{\boldsymbol{s}_l}\log p_{\boldsymbol{\theta}}(\boldsymbol{t}|\boldsymbol{x}))^T\right] \tag{10}$$

$$\boldsymbol{A}_{l-1} = \mathop{\mathbb{E}}_{(\boldsymbol{x},\boldsymbol{t})\in\mathcal{B}}\left[\boldsymbol{a}_{l-1}\boldsymbol{a}_{l-1}^T\right]. \tag{11}$$

Note that the gradients $\nabla_{\boldsymbol{s}_l}$ are with respect to the pre-activation for each layer. The approximation happens when the expectation $\mathbb{E}_{(\boldsymbol{x},t)\in\mathcal{B}}$ is taken before the Kronecker product, whereas the exact FIM takes the Kronecker product before the expectation.
The inverse of a matrix can be calculated by taking Kronecker product of the inverse of its Kronecker factors

$$\boldsymbol{F}_l^{-1} = \boldsymbol{B}_l^{-1} \otimes \boldsymbol{A}_{l-1}^{-1}. \tag{12}$$

Kronecker factorization can be extended convolutional layers and other layers as well [6].

Since the inverse operations can be performed independently for the $\boldsymbol{A}$ and $\boldsymbol{B}$ matrix for each layer, the overhead for computing the inverse decreases proportional to the number of GPUs.

2.4 Distributed K-FAC

The combination of layer-wise block-diagonal approximation and Kronecker factorization reduces the cost of computing large FIM. In this section, we introduce a strategy to further reduce the overhead of computing large FIM by use of distributed computing [19].

The algorithm for our distributed K-FAC optimizer is shown in Algorithm 1. The difference between a distributed SGD optimizer is shown in bold. Kronecker factors for the activation $\boldsymbol{A}$ are calculated at each layer during the forward propagation, while the Kronecker factors for the gradient $\boldsymbol{B}$ are calculated at each layer during the backward propagation. At this point, each GPU keeps the expectation of $\boldsymbol{A}$ and $\boldsymbol{B}$ over the local mini-batch, but for all layers. The expectation of these Kronecker factors are calculated for the global mini-batch by calling a Reduce-ScatterV collective communication, which computes the sum among the different GPUs while scattering each layer among them. After this communication, each GPU keeps the expectation of $\boldsymbol{A}$ and $\boldsymbol{B}$ over the global mini-batch, but for a few layers. Therefore, each one can work on a unique set of layers, and calculate the inverse of $\boldsymbol{A}$ and $\boldsymbol{B}$ for those layers independently. Once the inverse (Cholesky factor) is calculated for a subset of layers, we can precondition the corresponding gradients for those layers, and use that to update the corresponding weights for them. At this point, each GPU keeps a mutually exclusive set of updated weights. Finally, an AllGatherV collective communication is called to gather these mutually exclusive sets of weights so that all GPUs have updated them for all layers.

When the number of GPUs is the same as the number of layers, each GPU needs to handle only two matrices ($\boldsymbol{A}$ and $\boldsymbol{B}$ for a given layer). For example, if we consider a 10-layer network with the somewhat uniform number of parameters for each layer, using 10 GPUs will reduce the overhead of the FIM calculation by 10x. Note that we are considering a data-parallel setting, hence the

computation of SGD does not decrease when more GPUs are used. Therefore, only the overhead of computing the FIM decreases and eventually becomes comparable to the SGD computation time, as we will show in section 3.3.

Algorithm 1: Distributed K-FAC Optimizer

while *not converge* **do**
  foreach $l = 1, \cdots, L$ **do**
    forward l-th layer
    compute Kronecker factor A_{l-1}
  foreach $l = L, \cdots, 1$ **do**
    compute Kronecker factor B_l
    backward l-th layer
  Reduce+ScatterV$(\nabla L_{1:L}, A_{0:L-1}, B_{1:L})$
  for $l = 1, \cdots, L$ **do in parallel**
    compute inverse of each factor B_l^{-1}, A_{l-1}^{-1}
    compute preconditioned grad $\mathcal{G}_l$
    update parameter θ_l using $\mathcal{G}_l$
  AllGatherV(θ)
return θ

3 PERFORMANCE OPTIMIZATIONS

In this section, we will describe the strategies for performance optimization of our implementation of the distributed K-FAC method. In the end, we were able to reduce the per-step computation time of K-FAC to only twice that of SGD. Previous work by Osawa *et al.* showed that it was possible to train ResNet-50 on ImageNet-1K[21] in 23 epochs using K-FAC, whereas SGD typically takes 90 epochs. Therefore, being able to reduce the computation time per-step to twice that of SGD will yield an overall training time of K-FAC that is faster than SGD. The fact that we can calculate the FIM almost exactly in the same amount of time as a single SGD step has further implications. A theoretical analysis using higher-order information such as the Hessian and its generalized Gauss-Newton approximation is now within reach even for the largest models.

In the remainder of this section, we will show a step-by-step analysis of the performance optimizations that eventually leads to the significant reduction of the overhead of computing the FIM. We perform before-after comparisons for each optimization technique we introduce. In section 3.1 we describe the basic framework of our implementation. Then, in section 3.2 we describe that the performance optimizations can be applied to both K-FAC and SGD. Finally, in section 3.3 we describe the performance optimizations that are unique to the distributed K-FAC method.

3.1 Implementation

K-FAC was initially implemented in TensorFlow by Martens *et al.*[14] Based on the code[3] provided by [19], we have developed our implementation using Chainer since it provides a simpler interface to linear algebra CUDA kernels through the CuPy/NumPy interface, and a highly scalable NCCL interface for the collective communication we heavily depend on. However, we do not have a strong inclination towards a particular framework, and Chainer is similar enough to PyTorch that porting the code can be done fairly easily. For example, `cupy.ndarray` can be converted to `torch.Tensor` without time-consuming memory copy by using `__cuda_array_interface__`[4].

In the course of optimizing our implementation of K-FAC, we have introduced changes that would also benefit SGD. To distinguish these general techniques from the ones that only benefit K-FAC, and also to provide a competitive baseline for measuring the overhead of our K-FAC implementation, we will first describe these general techniques in the following subsection.

3.2 SGD Performance Optimization

Before describing the performance optimization techniques that bring the overhead of K-FAC down to competitive value, we first describe the optimization techniques for the forward and backward propagation that benefit both SGD and K-FAC.

In the present work, we choose the training of ResNet-50 on ImageNet-1K as the target for our performance optimization, since it has been used as a common benchmark for distributed training [1, 5, 28], and has highly optimized references on MLPerf.

Figure 1 shows the breakdown of the per-step training time of ResNet-50 on ImageNet-1K when different performance optimization techniques are implemented in succession.

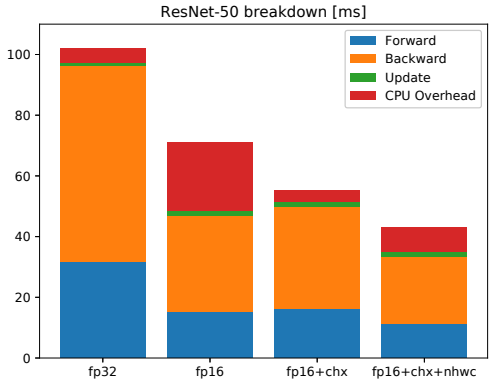

Figure 1: Breakdown of ResNet-50 training on ImageNet-1K. Image is cropped to 224×224 resolution. Batch size is 32.

`Forward` is the time it takes to execute the GPU kernel for the forward propagation. This also includes the time of data augmentation – our case Mixup [30]. We use mixup not only because its regularization effect helps K-FAC from overfitting, but also because it helps SGD to achieve a slightly higher validation accuracy. Therefore, we consider it as an effective regularization technique that benefits both SGD and K-FAC. `Backward` shows the time it takes to execute the GPU kernel for the backward propagation. `Update` is the execution time of the GPU kernel that performs the update of the parameters. `Overhead` is computed by measuring the actual wall clock time per step and subtracting the execution time of the three GPU kernels above. This includes the Python overhead to call the CUDA backend (cuDNN) and manage a computational graph that could not be overlapped with the GPU kernels.

3.2.1 Baseline. Our baseline uses 32-bit floating point (`float32`) for all variables, for which the training time per step is shown in Figure 1. For this case, the `Forward` and `Backward` takes up most of the time.

[3] https://github.com/tyohei/chainerkfac

[4] https://numba.pydata.org/numba-doc/dev/cuda/cuda_array_interface.html

3.2.2 fp16: Half-precision computation. We now introduce the use of 16-bit floating point (`float16`) for all variables, which is shown as the second bar in Figure 1. Although the use of `float16` may not yield stable and accurate convergence for some models, it has been shown in previous work that the training of ResNet-50 can be done with `float16` [10] and `bfloat16` [28]. Following the work [10, 16], we use `float16` for gradients, but we store the model parameters copy in `float32`. We also perform loss scaling and weight normalization to restrict the dynamic range of the variables.

The training time per step when `float16` is used is shown in Figure 1. When compared with the time for `float32` shown in the same figure, it can be seen that `float16` accelerates the `Forward` by 2.1x and the `Backward` by 2.04x. However, the `Overhead` increases when `float16` is used, so the overall speedup is only 1.43x. The increase in overhead could be caused by the parts that are computed on the CPU such as the construction of the computational graph, which was originally overlapped with the GPU computation.

The following solutions can be used to reduce this overhead.

- Increase the batch size so that the GPU kernels take longer to compute, thus hiding the CPU overhead once again. This will result in a larger global batch size and is not preferable in a data-parallel distributed setting.
- Accelerate the construction of the computation graph. Define-by-Run execution model by Chainer requires the computation graph to be constructed on-the-fly, which has a non-negligible overhead when implemented in Python.

3.2.3 Optimizing CPU-side code with ChainerX. In Chainer, the construction and backward propagation of the computation graph is implemented in Python. ChainerX[5] implements this part in C++, hence the CPU overhead is greatly reduced.

When computing second order information for each sample, we can use the `hook` in Chainer to extract this information for each layer during `Forward` and `Backward` [4, 19]. However, ChainerX does not have `hooks`, therefore we had to implement this capability on our own. By doing so, we were able to add this capability to ChainerX in a non-intrusive way that works for any model.

The per step training time for the use of ChainerX is shown in Figure 1 as `fp16+chx`. When compared to the Chainer implementation shown as `fp16`, the `Overhead` has been reduced to 16.9%. The effect of using `float16` is now reflected in the total training time because the `Overhead` is no longer the dominant component.

3.2.4 NHWC tensor structure. Now that we have confirmed the benefit of using `float16`, we consider the efficient use of TensorCores that are available on modern NVIDIA GPUs. The standard layout for the tensors in convnets is `NCHW`, where N is the batch size, C is the number of channels, and H and W are the height and width of the filter. For the efficient use of TensorCores, this layout must be changed to `NHWC`[6], in order to remove the need to transpose the feature map before the TensorCore computation.

In addition to the restructuring of the tensor, we also fuse some of the GPU kernels. The batch norm layers and activations have a lower Flop/Byte ratio compared to the linear or convolution layers. Therefore, their performance is limited by the memory bandwidth rather than the arithmetic throughput of the GPUs[11]. cuDNN 7.4.1 onwards provides an API where the batch norm, activation, and addition are fused.

The training time per step when using the `NHWC` layout and fused kernels is shown in Figure 1 as `fp16+chx+nhwc`. The `Forward` and `Backward` time are now 2.81x and 2.90x faster than the baseline, respectively. As a result, we are now able to train ResNet-50 on ImageNet-1K at 744.8 images / sec (without communication). Also by utilizing NCCL's AllReduce collective communication for distributed gradient accumulation, we can perform distributed training with 128 GPUs at throughput 561.4 images / sec. Table 2 shows the distributed SGD update time per step of ResNet-50 on ImageNet-1K classification. By using mixed-precision training, and also reducing memory access and CPU overhead, our SGD training and other training in NVIDIA Tesla V100 are comparable throughputs.

3.3 K-FAC optimizations

In this section, we will describe the performance optimization techniques that are unique to distributed K-FAC training. Figures 2 and 3 show the training time per step of our K-FAC implementation with and without the performance optimization, respectively. The `Forward`, `Backward`, and `Update` times are taken from the optimized version after all the performance optimization techniques mentioned in the previous section are implemented. The `Overhead` of K-FAC is simply calculated by subtracting the SGD time from the K-FAC time, as it was difficult to measure the exact time for the K-FAC overhead directly.

The objective of this section is to show step-by-step, how we are able to achieve the performance shown in Figure 3 starting from Figure 2. The detailed step-by-step performance optimization is shown in Figure 4.

3.3.1 Distributed computation of the FIM inverse. It can be seen from Figure 3 that the time for the `inverse` computation decreases inversely proportional to the number of GPUs. All of our experiments use 1 GPU per process. When only 1 GPU is used, the inversion of the block-diagonal FIM is computed by inverting each block separately, where we loop over all the layers and call a GPU kernel for Cholesky decomposition of each block. Since the expectation of the FIM is taken across all GPUs before the inversion, there is no parallelism to be extracted across the samples at this point. We therefore distribute each layer across the GPUs to extract the parallelism across the different layers. Contrary to the case of layer-parallelism in the forward and backward propagation, layer-parallelism for the inverse computation does not result in any communication, nor does it have any dependencies between the layers that merit pipelining. When the number of GPUs is smaller than the number of layers, each GPU handles multiple layers. When the number of GPUs exceeds the number of layers, some GPUs will remain idle during the matrix inversion step.

For example, for ResNet-34 the number of layers that can be computed independently is 38. Therefore, using more than 38 GPUs will not result in any more speedup. On the other hand, the forward and backward propagation will keep scaling due to data-parallelism, until the global batch size becomes too large and generalization starts to degrade. For ResNet-50 the number of matrices is 110 and

[5] https://chainer.org/announcement/2018/12/03/chainerx.html
[6] https://docs.nvidia.com/deeplearning/sdk/cudnn-developer-guide/index.html#tensor-ops-tensor-transformations-conversion

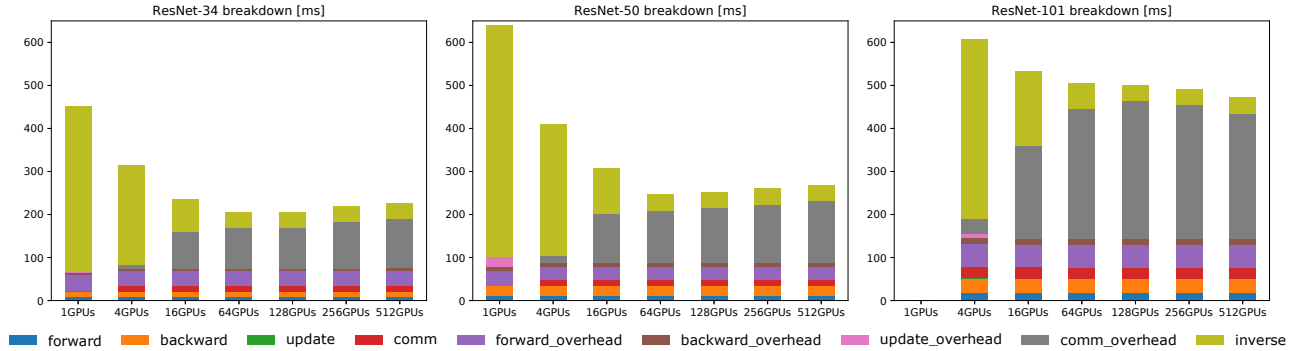

Figure 2: ResNets breakdowns with various GPUs, without K-FAC specific performance optimizations. Increasing the number of GPUs significantly reduces the time to invert the Kronecker factors (`inverse`). On the other hand, communication costs (`comm_overhead`) due to the increased number of GPUs cannot be ignored. Due to the large amount of space occupied by the Kronecker factors, ResNet-101 could not be trained using K-FAC on one GPU.

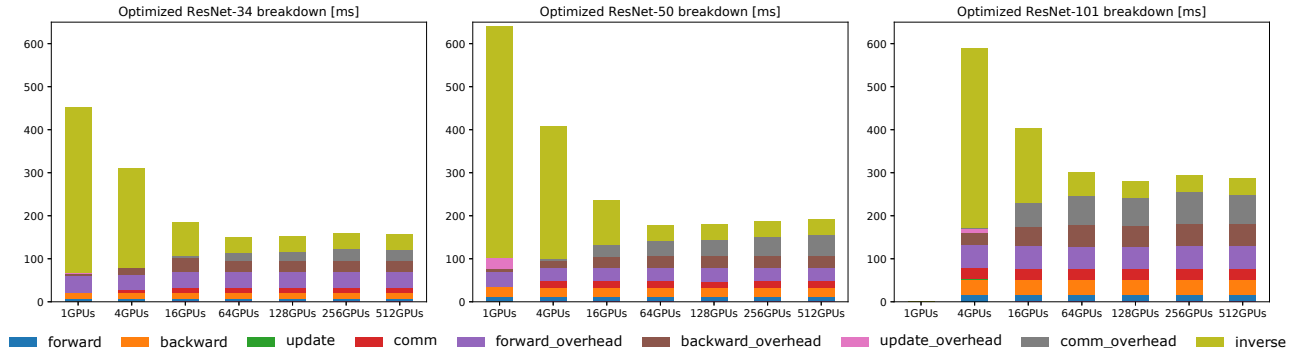

Figure 3: ResNets breakdowns with various GPUs, with K-FAC specific performance optimizations. The performance optimizations listed in the Table 1 significantly reduce the computation and communication overhead of multiple GPU training for all ResNet architectures.

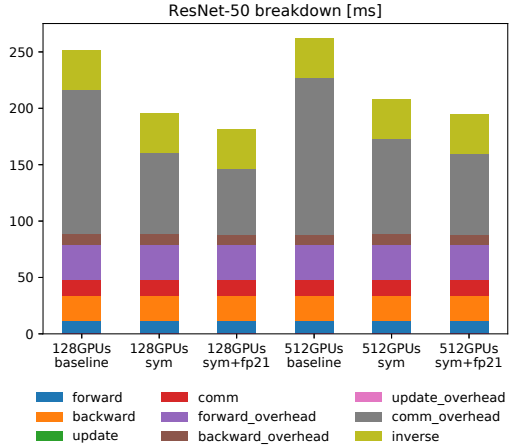

Figure 4: Step-by-step performance analysis of K-FAC optimizations listed in the Table 1.

for ResNet-101 it is 212. This means the amount of parallelism is larger for deeper models. One thing to keep in mind is that the number of parameters varies across layers, as well as the size of the matrices to be inverted.

The load-balance between the different GPUs for ResNet-34 on 128 GPUs is shown in Figure 5. On a single GPU, the inversion takes 384 ms, and on 128 GPUs the largest layer takes approximately 36 ms. ResNet-34 has only 110 matrices so when 128 GPUs are used some of them remain idle during the inversion. When the number of GPUs is much smaller than the number of layers, we can balance the workload among the GPUs, but as the number of GPUs increases the load-balancing becomes difficult. In Figure 5, the processes with smaller workload seem like they have a larger communication time, but this is actually caused by these processes having to wait until the bulk-synchronous communication can be performed.

This load-imbalance is less of a problem for Transformers [25], where the number of parameters does not vary between layers. Therefore, we anticipate that the overhead of K-FAC could be reduced even further for Transformers.

3.3.2 Communication optimization of symmetric matrix. When comparing K-FAC and SGD in Figure 2, we see that the K-FAC communication is quite large. K-FAC needs to communicate the Kronecker factors $\boldsymbol{A}$ and $\boldsymbol{B}$, which is typically larger than the gradient vector that the SGD communicates.

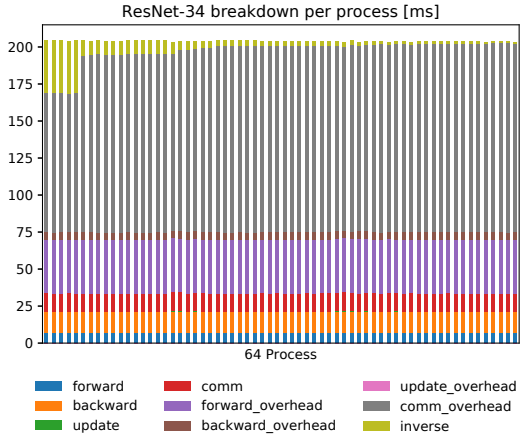

Figure 5: ResNet-34 breakdowns per each process. Distributed FIM inverse is only applied.

Here, each Kronecker factor $\boldsymbol{A}, \boldsymbol{B}$ is symmetric matrix. Therefore to calculate the mini-batch average of the Kronecker factors, only the upper triangular matrix is communicated, then the upper triangular part will be extended to the lower remaining part. Under the assumption that the parameter size of each Kronecker factor is m, the communication volume is reduced to $\frac{m(m+1)}{2}$ from m^2.

3.3.3 Low precision communication of Kronecker factors. When comparing optimized K-FAC and SGD in Figure 4, we see that the K-FAC communication is still a hot spot. To reduce the communication volume, we use low precision communication for Kronecker factors as the gradient vector has communicated in half-precision.

However, half-precision communication could not keep a training curve at the level of the single-precision training can produce. We attribute that half-precision communication breaks its small eigenvalues due to rounding.

To tackle this problem, we employ custom floating precision format — `float21` [9]. Data representation of `float21` is shown in Figure 6. This format can be implemented in NCCL easily thanks to its truncated `float32` format (Figure 7). We packed three `float21` in 64-bit data type to preserve memory alignment.

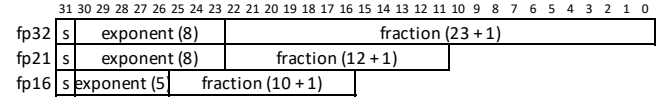

Figure 6: Data representation of 21-bit floating point format.

```
class FP21 {
  uint64_t a : 21, b : 21, c : 21;
};
void packFP21(FP21 *p, float a, float b, float c) {
  p->a = (*reinterpret_cast<uint32_t *>(&a)) >> 11;
  // Also, packing of b and c.
}
void unpackFP21(const FP21 p, float *a, float *b, float *c) {
  *reinterpret_cast<uint32_t *>(a) = (p.a << 11) | (1 << 10);
  // Also, unpacking of b and c.
}
```

Figure 7: Implementation of 21-bit floating point format.

3.3.4 Further performance optimization. Further performance optimizations are as follows:

- Overlapping communication and computation.
  As the equation (9) shows there are two types of Kronecker factors, $\boldsymbol{A}$ and $\boldsymbol{B}$. Factor $\boldsymbol{A}$ is constructed from the activations, which is calculated in Forward propagation. Therefore, we can perform $\boldsymbol{A}$'s Reduce-ScatterV and Backward propagation simultaneously.
  Tsuji *et al.* [23] reported that using overlap improves performance, but NCCL's communication kernel utilizes Streaming Multiprocessor (SM) of GPUs. Backward also utilizes GPU's SMs so its arithmetic throughput may be affected.
- Skipping FIM updates by utilizing the stale statistics.
  Osawa *et al.* [18] introduced a fast second-order optimization approach, which adaptively reduces the frequency of updating statistics (inverse of the FIM) based on changes in values of the stale statistics during training. This approach has the potential to reduce the overhead of constructing and transmitting Kronecker factors significantly.

Since estimating the effectiveness of each improvement in micro-benchmark is difficult, we evaluate them in actual training.

4 APPLICATION TO IMAGENET TRAINING

In this section, we apply performance optimizations to actual training — ResNet-50 training on ImageNet-1K dataset. This training task is often studied in previous work [1, 5, 28]. Also, we benchmark further optimizations described in section 3.3.4. We conduct all experiments on the ABCI (AI Bridging Cloud Infrastructure). which has 1088 nodes with four NVIDIA Tesla V100 GPUs per node.

4.1 How overlap of communication and computation works in training

As described in section 3.3.4, overlap of communication and computation works effectively in K-FAC method [23]. We empirically showed that it does not work because of NCCL's design. NCCL uses the Streaming Multiprocessor (SM) for communication progress but also uses it for the reduction. We can control how NCCL uses them by changing the following environment variables[7]:

- `NCCL_MAX_NRINGS`: The number of rings for Ring-AllReduce[20].
- `NCCL_NTHREADS`: The number of threads for each ring.

Figure 8 shows the per step update time of K-FAC using overlap techniques. Firstly, we can see `sym+fp21+overlap` is slower than `sym+fp21` due to its heavy `backward_overhead`. In this situation, NCCL automatically uses four rings to utilize NVLink and InfiniBand effectively. To reduce SM usage of NCCL, we restricted the number of rings but the communication time became longer (`sym+fp21+overlap`, `rings=2`). Using hierarchical AllReduce [24] can save SM resources because it splits the communication kernels into multiple kernels, for intra-node communication and inter-node communication (`sym+fp21+overlap`, `hier`). In our experiment, NCCL by default uses twelve rings and two rings for intra- and inter-node communication, respectively. Therefore, the SM usage of hierarchical version lies between `ring=2` and `ring=4` but the performance improvement is insignificant.

[7] https://docs.nvidia.com/deeplearning/sdk/nccl-developer-guide/docs/env.html

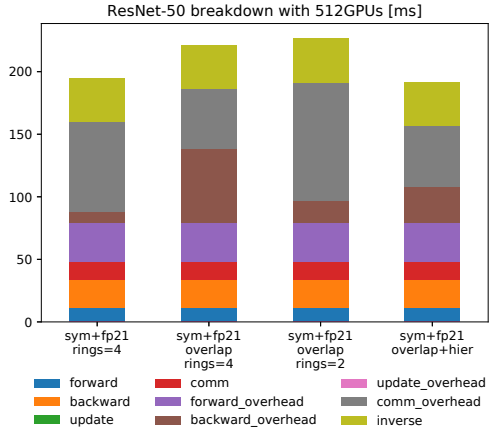

Figure 8: Performance of overlapped communication.

However, we empirically observed that using hierarchical communication increases the tolerance for low precision communication of Kronecker factors, which otherwise would not converge. Thus, we employ overlap and hierarchical communication in the following subsection.

4.2 Training ResNet-50 on ImageNet-1K

We apply the performance optimization to training of ResNet-50[7] on ImageNet-1K classification. We follow the K-FAC training settings described in [18]. Unlike their settings, we adopt ResNet-D[8].

Table 3 shows our training result. Without the use of stale FIMs, we achieved 76.1% Top-1 validation accuracy in 38.6 min. with the mini-batch size of 4K (128 GPUs), and also 76.0% Top-1 validation accuracy in 10.5 min. with a mini-batch size of 16K (512 GPUs). The per step update time is 212, 217 and 230 ms for 128, 256 and 512 GPUs, respectively. These training configurations have not suffered from the communication latency of large-scale training.

With the use of stale FIMs, the Top-1 validation accuracy in each configuration still keep competitive accuracy ($> 75.0\%$) but the speedup is 1.9x compared to the case without the use of stale FIMs.

In the mini-batch size of $64K$ (2048 GPUs), we achieved 75.6% Top-1 validation accuracy in 2.7 min. with stale FIMs. Also, we achieved 75.0% Top-1 validation accuracy in 2.0 min.

5 DISCUSSION AND CONCLUSIONS

In this work, we conducted a step-by-step performance analysis of the overhead of computing the Fisher information matrix (FIM) during the training of ResNet-50 on ImageNet-1K. We use a block-diagonal approximation and Kronecker factorization (K-FAC) to approximate the FIM. Our hybrid data-model-parallel distributed implementation achieved 75.0% Top-1 validation accuracy in 2 min. with a mini-batch size of 81K on 2048 V100 GPUs. Our step-by-step performance optimization involved a combination of the use of mixed-precision arithmetic, reordering the tensor layout, taking advantage of the symmetry of the matrix during communication, 21-bit communication, overlapping computation and communication, hierarchical collective communication, and the use of stale FIM.

In order to distinguish the performance optimizations that benefit SGD training from the ones that only benefit K-FAC, we first showed a step-by-step performance optimization of SGD. In the end, our SGD matches the performance of other state-of-the-art implementations of SGD, and we are able to show the incremental steps to achieve this performance. Using this highly optimized SGD as a baseline, we conducted a step-by-step performance optimization of K-FAC. The step-by-step analysis revealed that overlapping computation and communication can have a detrimental effect. When the backward propagation computation is not optimized and the GPU is not being fully utilized overlapping works well, but after optimizing the backward propagation, the use of GPU resources by the NCCL communication interferes with the computation and slows down the backpropagation when overlapped. Being able to distribute the computation of the inverse FIM results in over an order of magnitude speedup, but it does not speed up proportional to the number GPUs due to the load-imbalance between layers. Note that in data-parallel training the time per step remains constant, so the speed of K-FAC gets closer and closer to SGD as the number of GPUs increases. With all the performance optimizations for K-FAC, we were able to reduce the per step training time of K-FAC to 1.89 times that of SGD. This number may seem large, but being able to compute the FIM and its inverse with less than double the time of a single SGD step has significant implications. To be precise the FIM we use in K-FAC is the empirical FIM, which can be very different from the exact FIM [13, 22]. However, techniques like KFRA [3] make it possible to compute the exact FIM with the same computational cost as the empirical FIM. Similar techniques can also be used to compute the Jacobian and Hessian with a similar computational cost. Therefore, the fact that we are computing the empirical FIM does not change our conclusion that these information matrices can be computed within the same time frame as a single SGD step. As can be seen from the breakdown plots in our experiments, the dominant portion of the computation is the `inverse` calculation and not the computation of the matrices themselves.

6 ACKNOWLEDGMENTS

This work is supported by JST CREST Grant Number JPMJCR19F5, Japan. Part of this work is conducted as research activities of AIST-Tokyo Tech Real World Big-Data Computation Open Innovation Laboratory (RWBC-OIL). Computational resource of AI Bridging Cloud Infrastructure (ABCI) was awarded by "ABCI Grand Challenge" Program, National Institute of Advanced Industrial Science and Technology (AIST). This work is supported by "Joint Usage/Research Center for Interdisciplinary Large-scale Information Infrastructures" and "High Performance Computing Infrastructure" in Japan (Project ID: jh200023-NAHI and jh200037-NAH). This research used computational resources of the HPCI system provided by Tokyo Institute of Technology through the HPCI System Research Project (Project ID:hp200030).

REFERENCES

[1] Takuya Akiba, Shuji Suzuki, and Keisuke Fukuda. 2017. Extremely Large Minibatch SGD: Training ResNet-50 on ImageNet in 15 Minutes. *arXiv:1711.04325 [cs]* (Nov. 2017). http://arxiv.org/abs/1711.04325

[2] Shun-ichi Amari. 1998. Natural Gradient Works Efficiently in Learning. *Neural Computation* 10, 2 (Feb. 1998), 251–276. https://doi.org/10.1162/089976698300017746

[3] Aleksandar Botev, Hippolyt Ritter, and David Barber. 2017. Practical Gauss-Newton optimisation for deep learning. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70 (ICML'17)*. JMLR.org, Sydney, NSW, Australia, 557–565.

Table 3: Training time and Top-1 validation accuracy of ResNet-50 on ImageNet-1K classification with NVIDIA Tesla V100 GPUs. Time for SGD is simulated for 90 epochs for each mini-batch size using the SGD performance optimizations described in Sec. 3.2. Time for K-FAC is measured with all the performance optimizations described in Table 1.

# of GPUs	Mini-batch size	Optimizer	Stale FIM	# of Updates	Time	Accuracy
128	4,096	SGD	-	28,151	26.7 min.	-
		K-FAC	not applied	10,920	38.6 min.	76.1 %
		K-FAC	✓	10,920	**21.5** min.	75.7 %
256	8,192	SGD	-	14,076	13.4 min.	-
		K-FAC	not applied	5,460	19.8 min.	76.0%
		K-FAC	✓	5,460	**9.4** min.	75.5%
512	16,384	SGD	-	7,038	6.7 min.	-
		K-FAC	not applied	2,730	10.5 min.	76.0 %
		K-FAC	✓	2,730	**5.5** min.	74.9 %
2048	65,536	K-FAC	✓	1,178	2.7 min.	75.6 %
	81,920		✓	795	2.0 min.	75.0 %

[4] Felix Dangel, Frederik Kunstner, and Philipp Hennig. 2019. BackPACK: Packing more into backprop. *arXiv:1912.10985 [cs, stat]* (Dec. 2019). http://arxiv.org/abs/1912.10985

[5] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. *arXiv:1706.02677 [cs]* (June 2017). http://arxiv.org/abs/1706.02677

[6] Roger Grosse and James Martens. 2016. A Kronecker-factored approximate Fisher matrix for convolution layers. *arXiv:1602.01407 [cs, stat]* (May 2016). http://arxiv.org/abs/1602.01407

[7] K. He, X. Zhang, S. Ren, and J. Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 770–778. https://doi.org/10.1109/CVPR.2016.90

[8] Tong He, Zhi Zhang, Hang Zhang, Zhongyue Zhang, Junyuan Xie, and Mu Li. 2018. Bag of Tricks for Image Classification with Convolutional Neural Networks. *arXiv:1812.01187 [cs]* (Dec. 2018). http://arxiv.org/abs/1812.01187

[9] Tsuyoshi Ichimura, Kohei Fujita, Takuma Yamaguchi, Akira Naruse, Jack C. Wells, Thomas C. Schulthess, Tjerk P. Straatsma, Christopher J. Zimmer, Maxime Martinasso, Kengo Nakajima, Muneo Hori, and Lalith Maddegedara. 2018. A fast scalable implicit solver for nonlinear time-evolution earthquake city problem on low-ordered unstructured finite elements with artificial intelligence and transprecision computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '18)*. IEEE Press, Dallas, Texas, 1–11.

[10] Xianyan Jia, Shutao Song, Wei He, Yangzihao Wang, Haidong Rong, Feihu Zhou, Liqiang Xie, Zhenyu Guo, Yuanzhou Yang, Liwei Yu, Tiegang Chen, Guangxiao Hu, Shaohuai Shi, and Xiaowen Chu. 2018. Highly Scalable Deep Learning Training System with Mixed-Precision: Training ImageNet in Four Minutes. *arXiv:1807.11205 [cs, stat]* (July 2018). http://arxiv.org/abs/1807.11205

[11] Wonkyung Jung, Daejin Jung, and Byeongho Kim, Sunjung Lee, Wonjong Rhee, and Jung Ho Ahn. 2018. Restructuring Batch Normalization to Accelerate CNN Training. *arXiv:1807.01702 [cs]* (July 2018). http://arxiv.org/abs/1807.01702

[12] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A. Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, Demis Hassabis, Claudia Clopath, Dharshan Kumaran, and Raia Hadsell. 2017. Overcoming catastrophic forgetting in neural networks. *arXiv:1612.00796 [cs, stat]* (Jan. 2017). http://arxiv.org/abs/1612.00796

[13] Frederik Kunstner, Lukas Balles, and Philipp Hennig. 2019. Limitations of the Empirical Fisher Approximation for Natural Gradient Descent. *arXiv:1905.12558 [cs, stat]* (Dec. 2019). http://arxiv.org/abs/1905.12558

[14] James Martens and Roger Grosse. 2015. Optimizing Neural Networks with Kronecker-factored Approximate Curvature. *arXiv:1503.05671 [cs, stat]* (March 2015). http://arxiv.org/abs/1503.05671

[15] Sam McCandlish, Jared Kaplan, Dario Amodei, and OpenAI Dota Team. 2018. An Empirical Model of Large-Batch Training. *arXiv:1812.06162 [cs, stat]* (Dec. 2018). http://arxiv.org/abs/1812.06162

[16] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. 2018. Mixed Precision Training. *arXiv:1710.03740 [cs, stat]* (Feb. 2018). http://arxiv.org/abs/1710.03740

[17] Hiroaki Mikami, Hisahiro Suganuma, Pongsakorn U-chupala, Yoshiki Tanaka, and Yuichi Kageyama. 2018. ImageNet/ResNet-50 Training in 224 Seconds. (Nov. 2018). https://arxiv.org/abs/1811.05233

[18] Kazuki Osawa, Yohei Tsuji, Yuichiro Ueno, Akira Naruse, Chuan-Sheng Foo, and Rio Yokota. 2020. Scalable and Practical Natural Gradient for Large-Scale Deep Learning. *arXiv:2002.06015 [cs, stat]* (Feb. 2020). http://arxiv.org/abs/2002.06015

[19] Kazuki Osawa, Yohei Tsuji, Yuichiro Ueno, Akira Naruse, Rio Yokota, and Satoshi Matsuoka. 2019. Large-Scale Distributed Second-Order Optimization Using Kronecker-Factored Approximate Curvature for Deep Convolutional Neural Networks. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 12351–12359. https://doi.org/10.1109/CVPR.2019.01264

[20] Pitch Patarasuk and Xin Yuan. 2009. Bandwidth Optimal All-reduce Algorithms for Clusters of Workstations. *J. Parallel and Distrib. Comput.* 69, 2 (Feb. 2009), 117–124. https://doi.org/10.1016/j.jpdc.2008.09.002

[21] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. 2015. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision* 115, 3 (Dec. 2015), 211–252. https://doi.org/10.1007/s11263-015-0816-y

[22] Valentin Thomas, Fabian Pedregosa, Bart van Merriënboer, Pierre-Antoine Mangazol, Yoshua Bengio, and Nicolas Le Roux. 2019. Information matrices and generalization. *arXiv:1906.07774 [cs, stat]* (June 2019). http://arxiv.org/abs/1906.07774

[23] Yohei Tsuji, Kazuki Osawa, Yuichiro Ueno, Akira Naruse, Rio Yokota, and Satoshi Matsuoka. 2019. Performance Optimizations and Analysis of Distributed Deep Learning with Approximated Second-Order Optimization Method. In *Proceedings of the 48th International Conference on Parallel Processing: Workshops (ICPP 2019)*. Association for Computing Machinery, Kyoto, Japan, 1–8. https://doi.org/10.1145/3339186.3339202

[24] Yuichiro Ueno and Rio Yokota. 2019. Exhaustive Study of Hierarchical AllReduce Patterns for Large Messages Between GPUs. In *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. 430–439. https://doi.org/10.1109/CCGRID.2019.00057

[25] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. *arXiv:1706.03762 [cs]* (June 2017). http://arxiv.org/abs/1706.03762

[26] Chaoqi Wang, Roger Grosse, Sanja Fidler, and Guodong Zhang. 2019. EigenDamage: Structured Pruning in the Kronecker-Factored Eigenbasis. *arXiv:1905.05934 [cs, stat]* (May 2019). http://arxiv.org/abs/1905.05934

[27] Masafumi Yamazaki, Akihiko Kasagi, Akihiro Tabuchi, Takumi Honda, Masahiro Miwa, Naoto Fukumoto, Tsuguchika Tabaru, Atsushi Ike, and Kohta Nakashima. 2019. Yet Another Accelerated SGD: ResNet-50 Training on ImageNet in 74.7 seconds. *arXiv:1903.12650 [cs, stat]* (March 2019). http://arxiv.org/abs/1903.12650

[28] Chris Ying, Sameer Kumar, Dehao Chen, Tao Wang, and Youlong Cheng. 2018. Image Classification at Supercomputer Scale. *arXiv:1811.06992 [cs, stat]* (Nov. 2018). http://arxiv.org/abs/1811.06992

[29] Yang You, Zhao Zhang, Cho-Jui Hsieh, James Demmel, and Kurt Keutzer. 2018. ImageNet Training in Minutes. In *Proceedings of the 47th International Conference on Parallel Processing (ICPP 2018)*. ACM, New York, NY, USA, 1:1–1:10. https://doi.org/10.1145/3225058.3225069

[30] Hongyi Zhang, Moustapha Cisse, Yann N. Dauphin, and David Lopez-Paz. 2018. mixup: Beyond Empirical Risk Minimization. *arXiv:1710.09412 [cs, stat]* (April 2018). http://arxiv.org/abs/1710.09412