Scalable K-FAC Training for Deep Neural Networks with Distributed Preconditioning

Lin Zhang, Shaohuai Shi, *Member, IEEE*, Wei Wang, *Member, IEEE*, Bo Li, *Fellow, IEEE*

Abstract—The second-order optimization methods, notably the D-KFAC (Distributed Kronecker Factored Approximate Curvature) algorithms, have gained traction on accelerating deep neural network (DNN) training on GPU clusters. However, existing D-KFAC algorithms require to compute and communicate a large volume of second-order information, i.e., Kronecker factors (KFs), before preconditioning gradients, resulting in large computation and communication overheads as well as a high memory footprint. In this paper, we propose DP-KFAC, a novel distributed preconditioning scheme that distributes the KF constructing tasks at different DNN layers to different workers. DP-KFAC not only retains the convergence property of the existing D-KFAC algorithms but also enables three benefits: reduced computation overhead in constructing KFs, no communication of KFs, and low memory footprint. Extensive experiments on a 64-GPU cluster show that DP-KFAC reduces the computation overhead by 1.55×-1.65×, the communication cost by 2.79×-3.15×, and the memory footprint by 1.14×-1.47× in each second-order update compared to the state-of-the-art D-KFAC methods. Our codes are available at https://github.com/lzhangbv/kfac_pytorch.

Index Terms—Distributed deep learning, second-order, K-FAC, performance optimization

1 Introduction

In distributed DNN (deep neural network) training, data parallelism is used as a common practice where the training model is replicated on multiple workers and updated in parallel with local data samples. In the data-parallel settings, a popular training method is distributed synchronous stochastic gradient descent (S-SGD) algorithm [1], [2], [3], [4], [5]. However, S-SGD only utilizes the first-order gradients to update model parameters, and often requires a large number of iterations to converge [3], [6], [7], [8], making it slow when training a large model.

In light of the inefficiency of S-SGD algorithms, second-order optimization methods have been proposed recently to *precondition* gradients using the Fisher Information Matrix (FIM), which are proven effective with evidences in both theory [9], [10], [11], [12] and experiments [13], [14], [15], [16], [17], [18]. Preconditioning gradient with second-order information often leads to a faster convergence due to the alleviated effect of pathological curvature [19]. By using Kronecker Factored Approximate Curvature (K-FAC) algorithms [20], [21], [22], [23], the FIM can be approximated layer-wise for DNNs in a compute-efficient way. Existing works [13], [14], [15], [17], [18], [24] have empirically verified that training DNNs with distributed K-FAC (D-KFAC) can converge in a much smaller number of iterations than that of SGD algorithms. Notably, the ResNet-50 [25] model reaches the target accuracy of 75.9% on the ImageNet dataset [26] in 45 epochs using D-KFAC [14], [17], [18], whereas SGD requires 68 epochs with carefully tuned hyper-parameters according to MLPerf [27]. Our experimental result (Fig. 1) in training a ResNet-110 [25] model on the Cifar-10 dataset [28] also shows that D-KFAC requires 40% fewer epochs to reach the target 93.5% validation accuracy [25] compared to SGD.

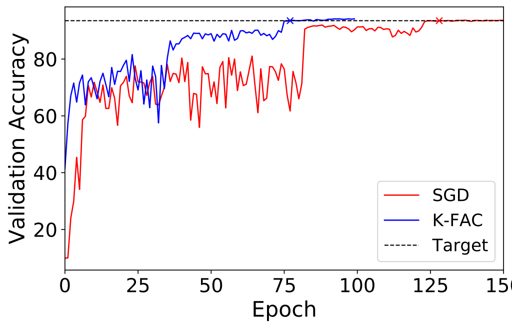


Fig. 1: Training the ResNet-110 model on the Cifar-10 dataset with SGD and K-FAC to achieve the target accuracy of 93.5%.

Despite the reduced number of iterations, existing D-KFAC algorithms incur expensive overheads in computation, communication, and memory footprint of distributed optimization per iteration, resulting in low scaling efficiency on GPU clusters. Specifically, in DNN training with SGD, it first samples the data to do the feed-forward (FF) computations to calculate the loss $\mathcal{L}$, and then computes the first-order gradient ($\nabla\mathcal{L}_i$ at layer i) w.r.t. the model parameters through backward propagation (BP), as shown in Fig. 2 (middle). Using K-FAC optimization, it needs to compute the Kronecker factors (A_{i-1} during FF and G_i during BP), as shown in Fig. 2 (side), to estimate FIMs by $F_i = A_{i-1} \otimes G_i$, where F_i is the approximate FIM at layer i of a DNN model and $\otimes$ is the Kronecker product. F_i is then inverted to precondition the gradients, i.e., $F_i^{-1}\nabla\mathcal{L}_i$. When applying data parallelism with multiple workers, the

- *Lin Zhang, Wei Wang, and Bo Li are with the Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, Hong Kong (email: lzhangbv@connect.ust.hk, {weiwa, bli}@cse.ust.hk).*
- *Shaohuai Shi is the corresponding author and with the School of Computer Science and Technology, Harbin Institute of Technology, Shenzhen (email: shaohuais@hit.edu.cn).*

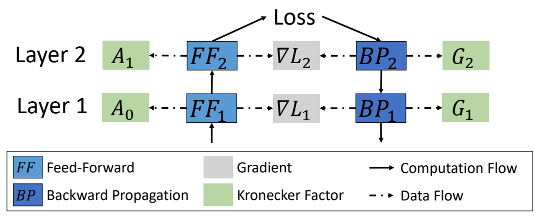


Fig. 2: An example of Kronecker factors (A and G) computation in a 2-layer DNN.

Kronecker factors (KFs) should be constructed locally and aggregated among workers to generate global KFs which are then inverted to precondition the gradients. This process takes a large portion of the training time in each iteration (more details in §3).

In [13], [14], [15], [16], the authors propose model parallelism (MPD-KFAC) methods to distributively compute the inverses of KFs, where different GPUs calculate the inverses of different KFs in parallel. From the system's point of view, one can either compute all preconditioned gradients in all workers (which results in minimum communication, called COMM-OPT [13], [14], [15], [17]) or distributively precondition gradients (which results in minimum temporary results storage, called MEM-OPT [16], [17]) to reduce the iteration time of D-KFAC optimization. Some scheduling algorithms are also proposed to further reduce the iteration time in D-KFAC. For example, SPD-KFAC [29] is proposed to pipeline the computations and communications of KFs. In [17], KAISA is proposed to dynamically determine COMM-OPT and MEM-OPT for particular cluster configurations. In [18], the KFs update interval can be dynamically determined by their traces. However, all these algorithms require every worker to calculate all layers' KFs which are further aggregated among all workers. Such a requirement introduces significant computation and communication overheads as well as high memory consumption to store KFs. Due to the layer-wise structure of DNNs and layer-wise FIM approximation of K-FAC [20], in this work, we propose a novel *distributed preconditioning* scheme, named DP-KFAC, by preconditioning different layers' gradients using local KFs (partial batch KFs computation) to accelerate distributed training with little impact on the model convergence.

In DP-KFAC, we distribute the computing tasks of preconditioning at different layers to different workers, which means each worker only constructs and inverts a part of KFs. Therefore, the per-worker computation workload and memory consumption is significantly reduced, and the communications of KFs among all workers are totally eliminated. We conduct experiments on a 64-GPU cluster connected with 100Gb/s InfiniBand on different applications including CNNs and Transformers. The experimental results show that our DP-KFAC preserves the convergence performance with existing D-KFAC algorithms and it achieves much more efficient training speed than existing state-of-the-arts KAISA [17] and SPD-KFAC [29]. Specifically, on our 64-GPU cluster, DP-KFAC saves 1.55-1.65× computational costs, 2.79-3.15× communication costs, 1.14-1.47× memory consumption, and 1.13-2.27× iteration time compared to existing state-of-the-art D-KFAC methods KAISA and SPD-KFAC. The main contributions of this paper are summarized as follows:

- We systemically analyze the computation and communication overheads of the existing D-KFAC implementations. It is observed that the system bottleneck is caused by the large amount of KFs to be constructed and communicated in distributed optimization.
- We propose a novel distributed preconditioning scheme, named DP-KFAC, to accelerate training by reducing the computation and communication overheads of constructing KFs, and it requires less memory consumption.
- We implement our DP-KFAC system atop existing popular deep learning frameworks PyTorch and Horovod. Users only need to add several lines of code to use DP-KFAC.
- We conduct extensive experiments with modern DNNs on a 64-GPU cluster with a 100Gb/s InfiniBand interconnect. The experimental results show that our DP-KFAC is much more efficient in iteration time and consumes less memory storage than state-of-the-art algorithms while maintaining almost the same validation accuracy.

The rest of the paper is organized as follows. We introduce some background and related work in §2. We then analyse the system bottlenecks of the existing D-KFAC implementations in §3. We present our DP-KFAC design in §4, followed by the system implementation in §5. We show experimental studies in §6. Finally, we conclude the paper in §8.

2 Background and Related Work

In DNN training, the target is to minimize a loss function $\mathcal{L}(\mathbf{w}, D)$, where $\mathbf{w}$ is the model parameters and D is the training data by iteratively updating $\mathbf{w}$. It can be optimized with its first-order gradient using SGD or the preconditioned gradient using second-order information. We summarize some frequently used notations as follows:

TABLE 1: Notations.

Name	Description
$\mathbf{w}$	model parameter vector
$\nabla\mathcal{L}$	gradient of a loss w.r.t. parameter
F	fisher information matrix (FIM)
A & G	Kronecker factors (KFs)
α	learning rate
P	the number of workers

2.1 Distributed SGD

The mini-batch SGD algorithm updates the model parameters iteratively by using the first-order gradient, i.e.,

$$\mathbf{w}_i^{(t+1)} = \mathbf{w}_i^{(t)} - \alpha^{(t)} \nabla\mathcal{L}_i(\mathbf{w}^{(t)}, \mathcal{B}^{(t)}), \qquad (1)$$

where $\mathbf{w}_i^{(t)}$ and $\nabla\mathcal{L}_i(\mathbf{w}^{(t)}, \mathcal{B}^{(t)})$ represent the model parameter and gradient of layer i at iteration t respectively, for $i = 1, \cdots, L$ in an L-layer DNN. $\alpha^{(t)} > 0$ is the learning rate, and $\mathcal{B}^{(t)}$ is a mini-batch of data randomly sampled from the training dataset.

When exploiting data parallelism to train a single model with multiple workers, the distributed synchronized SGD (S-SGD) algorithm updates the model by

$$\mathbf{w}_i^{(t+1)} = \mathbf{w}_i^{(t)} - \alpha^{(t)} \frac{1}{P} \sum_{p=1}^{P} \nabla \mathcal{L}_i(\mathbf{w}^{(t)}, \mathcal{B}^{(t),p}), \qquad (2)$$

where $\nabla \mathcal{L}_i(\mathbf{w}^{(t)}, \mathcal{B}^{(t),p})$ is the local gradient computed on the p-th worker with its locally sampled data $\mathcal{B}^{(t),p}$. Since the local gradients are located among P workers, it is necessary to aggregate the gradients (i.e., summation) before applied to update the model parameters. In the centralized architecture, a parameter server is required to pull local gradients (and push parameters) from (and to) all workers [30], while in the decentralized architecture, the *all-reduce* collective communication is used to aggregate the gradients among all workers [1].

2.2 Distributed K-FAC

The second-order algorithm with the natural gradient is to use FIMs to precondition the first-order gradient with the following update formula:

$$\mathbf{w}_i^{(t+1)} = \mathbf{w}_i^{(t)} - \alpha^{(t)} F_i^{-1} \nabla \mathcal{L}_i(\mathbf{w}^{(t)}, \mathcal{B}^{(t)}), \qquad (3)$$

where F_i is the FIM of layer i. The preconditioned gradient, i.e., $F_i^{-1}\nabla \mathcal{L}_i$, often leads to a faster convergence [10], [20], which means it can converge to a solution in fewer number of iterations than SGD.

To construct the FIM in a more efficient way, K-FAC approximates it as the Kronecker product of two smaller matrices:

$$F_i \approx \mathbb{E}[a_{i-1}a_{i-1}^T] \otimes \mathbb{E}[g_i g_i^T] \triangleq A_{i-1} \otimes G_i, \qquad (4)$$

where a_{i-1} and g_i are the input of layer i (i.e., output at layer $i-1$) and the gradient w.r.t. the pre-activation output of layer i, respectively. A_{i-1} and G_i are called Kronecker factors (KFs). As shown in Fig. 2, the Kronecker factors $A_{i-1} = \mathbb{E}[a_{i-1}a_{i-1}^T]$ and $G_i = \mathbb{E}[g_i g_i^T]$ can be calculated (estimated) during feed-forward and back-propagation with the same mini-batch of data, respectively [20]. The update rule of K-FAC in Eq. (3) can be further represented by

$$\mathbf{w}_i^{(t+1)} = \mathbf{w}_i^{(t)} - \alpha^{(t)} (A_{i-1}^{-1} \otimes G_i^{-1}) \nabla \mathcal{L}_i(\mathbf{w}^{(t)}, \mathcal{B}^{(t)}). \qquad (5)$$

Compared to SGD, K-FAC needs to construct and invert KFs for preconditioning.

When exploiting distributed K-FAC (D-KFAC) with data parallelism, the update rule can be represented as follows:

$$\mathbf{w}_i^{(t+1)} = \mathbf{w}_i^{(t)} - \alpha^{(t)} (\bar{A}_{i-1}^{-1} \otimes \bar{G}_i^{-1}) \nabla \bar{\mathcal{L}}_i, \qquad (6)$$

where $\nabla \bar{\mathcal{L}}_i = \frac{1}{P}\sum_{p=1}^{P} \nabla \mathcal{L}_i(\mathbf{w}^{(t)}, \mathcal{B}^{(t),p})$ is the aggregated gradient at layer i just like S-SGD. $\bar{A}_{i-1}$ and $\bar{G}_i$ are the distributed version of KFs used for preconditioning, and their values depend on particular algorithms.

In the centralized architecture with parameter server (PS) [30], the distributed K-FAC algorithm (PS-KFAC) [24] assigns additional workers to calculate and invert KFs locally, as shown in Fig. 3(a). This means that the computations of gradients and KFs are independent with each other using different data. Once the PS receives inverted KFs from additional workers, they can be directly served as $\bar{A}_{i-1}^{-1}$ and $\bar{G}_i^{-1}$ to precondition the aggregated gradient $\nabla \bar{\mathcal{L}}_i$.

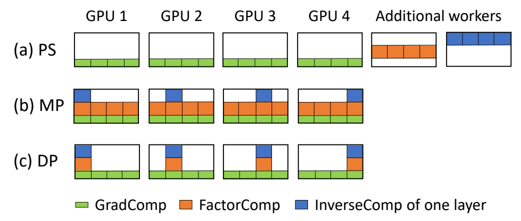


Fig. 3: An example of comparing the per-worker computational workloads of gradient computation (GradComp), KFs computation (FactorComp), and inverse computation (InverseComp) with three different D-KFAC implementations (i.e., PS, MP, and our DP), where we specify the number of GPUs as $P = 4$ and the number of DNN layers as $L = 4$. Any colour chunk denotes a specific computing task of one layer.

In the decentralized architecture, which is more widely used in practise [13], [15], [17], [18], each worker computes both local gradients and local KFs of all layers; and then the local gradients and KFs are both aggregated via *all-reduce* communications. After *all-reduce*, each worker has the identical aggregated gradient $\nabla \bar{\mathcal{L}}_i$ and global KFs, i.e., $\bar{A}_{i-1} = \frac{1}{P}\sum_{p=1}^{P} A_{i-1}^p$ and $\bar{G}_i = \frac{1}{P}\sum_{p=1}^{P} G_i^p$, where A_{i-1}^p and G_i^p are local KFs of layer i constructed on worker p. Since the inverse computations of different layers are independent from each other, MPD-KFAC [13], [14], [15] adopts the concept of model parallelism (MP) to partition the inverse tasks of different layers to different workers, as shown in Fig. 3(b). As each worker has only a part of inverse results used for preconditioning, the inverses should be either all-gathered among all workers for preconditioning (i.e., COMM-OPT in [17]) or used to precondition the gradients locally and then broadcasting the preconditioned gradients to all other workers for model updates (i.e., MEM-OPT in [17]). Note that most existing D-KFAC algorithms [14], [15], [16], [29] have followed this MP strategy.

There are also some communication scheduling strategies [18], [29] being proposed to alleviate the communication overheads in D-KFAC through pipelining techniques [29] or using dynamic update strategies [18] to improve the scaling efficiency of the distributed system. In this paper, our work is aligned with this direction.

3 Analysis of D-KFAC Algorithms

In this section, we provide an in-depth analysis to the existing D-KFAC algorithms. Specifically, we are interested in the data amount of gradients and KFs, which are directly related to the computation and communication efficiency.

Let N_g^i and N_f^i denote the number of elements of the gradient and KFs of layer i ($i = 1, \cdots, L$), respectively. If layer i is a simple linear transformation as follows[1]:

$$s_i = W_i a_{i-1} \quad \text{and} \quad a_i = \phi(s_i), \qquad (7)$$

1. Without loss of generality, a convolutional layer can also be formed as a linear transformation [21].

where $a_{i-1} \in \mathbb{R}^{d_{i-1}}$ is the input vector, $s_i \in \mathbb{R}^{d_i}$ is the pre-activation output vector, $W_i \in \mathbb{R}^{d_i \times d_{i-1}}$ is the associated parameter matrix, and ϕ is an element-wise activation function. Obviously, the gradient $\nabla \mathcal{L}_i$ at this layer has the same number of elements as the parameter W_i, that is, $N_g^i = d_i \times d_{i-1}$. According to Eq.(4), KFs A_{i-1} and G_i are two square matrices with the dimensions of d_{i-1} and d_i, respectively. Thus, the total number of elements of KFs in a layer is $N_f^i = d_{i-1}^2 + d_i^2$. Therefore, $\frac{N_f^i}{N_g^i} \geq 2$.

3.1 Computation Overheads

In the highly optimized D-KFAC design, MPD-KFAC, there are mainly three computing tasks: gradient computation (GradComp), factor computation (FactorComp), and inverse factor computation (InverseComp) as shown in Fig. 3(b). In the InverseComp stage, each GPU only needs to invert a part of KFs making the overhead of InverseComp be reduced (by P times on average) on a P-worker cluster. However, GradComp and FactorComp are still very time-consuming. This is because every worker needs to construct all layers' gradients and KFs, with the total data volume of $N_g = \sum_{i=1}^{L} N_g^i$ and $N_f = \sum_{i=1}^{L} N_f^i$, respectively. As $N_f > N_g$, FactorComp typically requires higher workloads than GradComp (including feed-forward and back-propagation).

3.2 Communication Overheads

In terms of the communication tasks, MPD-KFAC consists of gradient communication (GradComm), factor communication (FactorComm), and preconditioned gradient communication (PredComm). In the GradComm and FactorComm stages, the amount of data to be communicated via *all-reduce* operations is $2(P-1) \times N_g$ and $2(P-1) \times N_f$, for aggregating all local gradients and KFs, respectively. In the PredComm stage, because the preconditioned results of gradients are distributed on different workers, the data amount for broadcasting to all other workers is $(P-1) \times N_g$. Therefore, the FactorComm stage has introduced the most expensive communication overheads due to the largest data to be communicated. For example, the ResNet-50 [25] model has $N_f = 153.9$M but $N_g = 25.6$M.

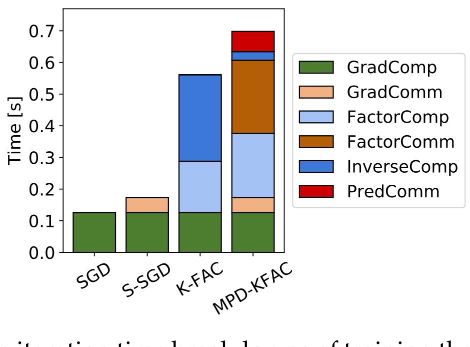


Fig. 4: The iteration time breakdowns of training the ResNet-50 model with existing algorithms.

To illustrate the algorithm efficiency empirically, we provide the iteration time breakdowns of SGD and K-FAC and their distributed versions (S-SGD and MPD-KFAC [13]) by training the ResNet-50 model on a 64-GPU cluster (the details of the system configurations can be found in §6). The results are shown in Fig. 4. Since the preconditioning time (PredComp) after the KFs have been inverted is very small compared to parts, we exclude PredComp in Fig. 4. It is seen that K-FAC runs more than four times slower than SGD at each iteration due to the high costs of constructing and inverting the KFs. MPD-KFAC reduces the inverse computation time vastly using multiple GPUs, but the factor computation and communication overheads become the new bottleneck.

In summary, existing highly optimized D-KFAC algorithms suffer from the system inefficiency due to the extensive overheads of constructing and communicating KFs of all layers for preconditioning. As a compromise, stale FIM construction and communication [13], [18] that reduce the frequency of KFs' expensive computations and communications can alleviate the inefficiency. However, skipping FIM updates by utilizing the stale statistics could bring potential negative effects on the convergence performance [16], [17], [20]. In this work, we focus on the per-iteration optimization to tackle the computation and communication challenges in D-KFAC algorithms.

4 Distributed Preconditioning

As discussed in the previous section, in the FactorComp stage of D-KFAC, each worker (say worker p) needs to construct local KFs (A_{i-1}^p during feed-forward) and (G_i^p during back-propagation) with the local sampled data, where $i = 1, 2, ..., L$, for an L-layer model. Then the communications are required in the FactorComm stage to aggregate the global KFs $\bar{A}_{i-1} = \frac{1}{P}\sum_{p=1}^{P} A_{i-1}^p$ and $\bar{G}_i = \frac{1}{P}\sum_{p=1}^{P} G_i^p$. This design is based on the assumption that the FIMs should be iteratively estimated according to the current mini-batch of data which is distributed to all participated workers in data parallelism. Thus, all local KFs should be aggregated to generate the global estimates of KFs. On the other hand, in the PS architecture, PS-KFAC [24] constructs KFs (A_{i-1} and G_i) with the sampled data different from the data used for calculating the first-order gradients in workers. The results in [24] show that PS-KFAC converges similarly with the original K-FAC.

Motivated by PS-KFAC, we propose a *distributed preconditioning* (DP) scheme (i.e., DP-KFAC) for D-KFAC, in the decentralized architecture using data parallelism, which uses a partial view of mini-batch for factor computation. In our DP-KFAC , each worker only constructs several layers' KFs using its locally sampled data to generate the estimate of empirical FIMs. Then the local KFs are inverted directly on that worker to precondition the corresponding layers' gradients. In other words, KFs of all layers are constructed and inverted distributively instead of using additional workers to calculate and invert KFs in PS-KFAC or aggregating KFs in MPD-KFAC.

Formally, given an L-layer DNN and a P-worker cluster, we use $S = \{l_1, l_2, \cdots, l_L\}$ to denote the set of all layers whose gradients need to be preconditioned before updating model parameters. In DP-KFAC, worker p ($p = 1, 2, ..., P$) only constructs KFs in a subset of layers $S_p \subset S$ that satisfies

$$S_1 \cup S_2 \cup ... \cup S_P = S, \tag{8}$$

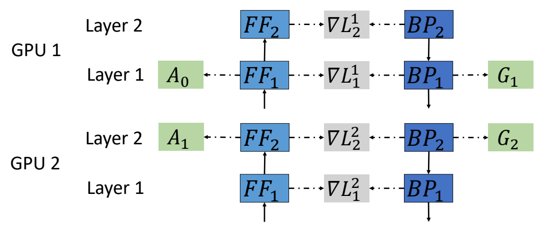


Fig. 5: An example of partitioning the tasks of KFs computations to two GPUs in DP-KFAC.

and

$$S_i \cap S_j = \varnothing \text{ for } i \neq j. \tag{9}$$

After KFs in S_p have been constructed at worker p, the worker directly inverts the KFs and then preconditions the first-order gradients on the layers in S_p. This means that in the FactorComp and InverseComp stages, worker p only computes and inverts the KFs of layers in S_p instead of S. As the processes of GradComp and GradComm remain the same as S-SGD in DP-KFAC, each worker has identical aggregated gradients (i.e., $\nabla \bar{\mathcal{L}}_i$ for $i = 1, 2, ...L$) after the *all-reduce* communications. Since worker p only inverts the KFs in S_p, the aggregated gradients of layers in S_p are also preconditioned on this worker. Finally, the distributed preconditioned gradients of layers in S_p are broadcast to other workers in the PredComm stage. After that, all workers have the identical preconditioned gradients which are used to update the model parameters. In summary, our DP-KFAC distributes the preconditioning which contains FactorComp and InverseComp stages at different layers to different GPUs (as shown in Fig. 3(c)) so that the computation and communication overheads can be significantly reduced.

An example with a 2-layer DNN running on two GPUs with DP-KFAC is shown in Fig. 5. The KFs (A_0 and G_1) in layer 1 are calculated on GPU 1, and the KFs (A_1 and G_2) in layer 2 are computed on GPU 2. Note that both GPU 1 and GPU 2 compute the local gradients of two layers. This means each GPU only needs to compute two-layer gradients and one-layer KFs. As GPU 1 has the results of A_0 and G_1 and GPU 2 has the results of A_1 and G_2, they further invert the KFs in parallel. At the same time, the local gradients are communicated via the *all-reduce* collective so that each GPU has the identical aggregated gradients of all layers. After that, GPU 1 can use the results of A_0^{-1} and G_1^{-1} to precondition the aggregated gradients of layer 1, while the results of A_1^{-1} and G_2^{-1} are used to precondition the gradients of layer 2 on GPU 2. Therefore, GPU 1 and GPU 2 construct and invert parts of all KFs in parallel.

The pseudo-code of DP-KFAC is shown in Algorithm 1. At the beginning of training, each worker is assigned with particular layers (line 1) for preconditioning. Lines 2-5 compute and communicate the gradients that are the same with S-SGD. Lines 6-13 precondition the gradients of all layers distributively. In lines 14-16, the distributed preconditioned gradients are broadcast among all workers. Finally, in lines 17-19, each worker updates the model parameters using the identical preconditioned gradients.

Algorithm 1 DP-KFAC

```
1: distribute layers to unique workers in a circular order;
2: for each worker $p$ do
3:     compute the gradients $\nabla\mathcal{L}_{1:L}$;
4: end for
5: communicate the gradients via all-reduce($\nabla\mathcal{L}_{1:L}$);
6: for each worker $p$ do
7:     for each layer $i$ assigned to $p$ do
8:         compute the Kronecker factors $A_{i-1}$ and $G_i$;
9:         Running average of $A_{i-1}$ and $G_i$;
10:        compute the eigen-decompositions or inverses of $A_{i-1}$ and $G_i$;
11:        precondition the gradients;
12:    end for
13: end for
14: for each layer $i$ assigned to $p$ do
15:     broadcast the preconditioned gradient from root worker $p$;
16: end for
17: for each worker $p$ do
18:     update weights using the preconditioned gradients;
19: end for
```

4.1 Running Average of KFs

In MPD-KFAC, KFs are computed through the global mini-batch size for the purpose of approximating the empirical FIMs over a long-term run [16]. That is,

$$A_{i-1}^{(t)} = \xi A_{i-1}^{(t)} + (1-\xi)A_{i-1}^{(t-1)}, \tag{10}$$

$$G_{i}^{(t)} = \xi G_{i}^{(t)} + (1-\xi)G_{i}^{(t-1)}, \tag{11}$$

where ξ is the running average and typically $\xi \in [0.9, 1)$. Therefore, when using the local mini-batch size in DP-KFAC to compute KFs over a large number of iterations, the expectation of KFs become the same as that with a global mini-batch size. Consequently, different local mini-batch sizes may have little impact on the convergence performance. We further conduct experiments to examine this problem when training with different local mini-batch sizes in §6.

4.2 Complexity analysis

Our DP-KFAC is mainly composed of three computation stages (GradComp, FactorComp, and InverseComp) and two communication stages (GradComm and PredComm). The GradComp and GradComm stages are the same with S-SGD. Each worker computes N_g elements of gradients, and the amount of data to be communicated is $2(P-1)\times N_g$. For other computation overheads, each worker only needs to construct and invert a fraction of KFs (i.e., $\frac{N_f}{P}$ on average) in the FactorComp and InverseComp stages. In the PredComm stage, the data amount to be broadcast is $(P-1)\times N_g$, which is the same as MPD-KFAC. Therefore, the computation overhead is significantly alleviated by reducing the per-worker workload of FactorComp by P times over MPD-KFAC, and the communication of KFs is totally eliminated. Overall, the efficiency of DP-KFAC with a amount of GPUs becomes comparable to S-SGD.

In terms of memory consumption, in DP-KFAC, each worker only needs to store $2 \times \frac{N_f}{P}$ elements of KFs of the assigned layers plus their inverse results. Therefore, the memory requirement in DP-KFAC is much less than MPD-KFAC.

The comparison of complexity between different algorithms is shown in Table 2.

TABLE 2: Complexity comparison of different algorithms. For MPD-KFAC and DP-KFAC, the complexity is measured on the second-order update.

Algorithm	S-SGD	MPD-KFAC	DP-KFAC
GradComp	N_g	N_g	N_g
FactorComp	0	N_f	N_f/P
InverseComp	0	N_f/P	N_f/P
GradComm	$2(P-1)N_g$	$2(P-1)N_g$	$2(P-1)N_g$
FactorComm	0	$2(P-1)N_f$	0
PredComm	0	$(P-1)N_g$	$(P-1)N_g$
Memory	N_g	$2(N_g+N_f)$	$2(N_g+N_f/P)$

5 Implementation

We implement our DP-KFAC atop PyTorch [31] and Horovod [32] as PyTorch is a popular deep learning framework and Horovod is widely used for distributed training. To make a minimal change of user code to use our DP-KFAC, the gradient computations and communications are kept unchanged, which are the same with S-SGD implemented in Horovod. Built on the KAISA preconditioner implementation [17], we implement a preconditioner (named "DP_KFAC") to perform distributed preconditioning to the gradient before updating model parameters. Thus, we wrap our DP-KFAC implementation with a PyTorch optimizer class, "DP_KFAC". Unlike KAISA, we support two types of damping (§5.3), matrix-inversion (with "inv_type=inverse") and eigen-decomposition (with "inv_type=eigen") when initializing an instance of "DP_KFAC".

The idea of distributed preconditioning can be implemented in other deep learning frameworks such as TensorFlow [33]. However, the difficulty of implementing DP-KFAC is how to support Kronecker factorization for many different DNN layers, which is challenging for all K-FAC algorithms. Similar to KAISA [17], our implementation now supports preconditioning on Linear and Conv2D layers.

5.1 Workload Distribution

In our implemented DP-KFAC, We apply a round-robin schedule to distribute all supported layers to different workers in a circular order. For example, given an L-layer DNN and a P-worker cluster, layers in $S_p = \{l_p, l_{p+P}, l_{p+2P}, \cdots\}$ are assigned to worker p. This means that worker p is only required to construct and invert the KFs for preconditioning on a subset of layers S_p. In particular, the KFs A and G of layers in S_p are constructed on the worker p during the feed-forward and back-propagation passes by registering the specific hook functions in PyTorch's "register_forward_pre_hook" and "register_backward_hook" APIs, respectively.

Finally, after the preconditioned gradients have been calculated (§5.3) distributively, we use the *broadcast* collective to collect all results in the PredComm stage before they are used to update the model parameters in all workers.

A load-balancing workload distribution (e.g., according to the number of floating-point operations per layer) could further improve the training performance. However, its improvement could be much smaller than the time reduced by eliminating the all-reduce operations in aggregating KFs. Thus, we will study this problem in our further version.

5.2 Supported Factor Calculation

We support factor calculation for Linear and Conv2D layers. Given that layer i has been assigned to worker p, the corresponding KFs A_{i-1} and G_i are estimated using the partial data on the p-th worker. Start from the linear layer, the KFs are given, derived from Eq. 4, as follows:

$$A_{i-1} = \frac{1}{|\mathcal{B}^p|}\sum_{j\in\mathcal{B}^p} a_{i-1,j}a_{i-1,j}^T, \quad G_i = \frac{1}{|\mathcal{B}^p|}\sum_{j\in\mathcal{B}^p} g_{i,j}g_{i,j}^T, \tag{12}$$

where $\mathcal{B}^p$ is the mini-batch on the p-th worker, and $a_{i-1,j}$ and $g_{i,j}$ are the activation and pre-activation gradient vectors, respectively, w.r.t. the j-th data sample of $\mathcal{B}^p$. For the linear layer in the Transformer model [34], as each data sample is not a vector but a sequence of vectors, we first build $a_{i-1,j}$ and $g_{i,j}$ by taking average over the sequence, and then use them to calculate KFs over the local mini-batch.

For the Conv2D layer, we follow the work of [14], [21], and calculate KFs as below:

$$A_{i-1} = \frac{1}{|\mathcal{B}^p|}M_{a_{i-1}}^T M_{a_{i-1}}, \quad G_i = \frac{1}{|\mathcal{B}^p|}\frac{1}{h_i w_i}M_{g_i}^T M_{g_i}, \tag{13}$$

where $M_{a_{i-1}}$ is the unfolding activation matrix in the shape of (batch size $\times h_i w_i$, $c_{i-1}k_i^2$), and M_{g_i} is the reshaped pre-activation gradient matrix of (batch size $\times h_i w_i$, c_i). c_{i-1} and c_i are input and output channel numbers, k_i is the kernel size, and h_i and w_i are output image height and width. .

5.3 Supported Inverse Calculation

In the InverseComp stage, the damping strategy [20] is often required for preconditioning the gradients with $(F_i+\gamma I)^{-1}$, where I is an identity matrix and $\gamma > 0$ is the damping value. The inverse can be computed directly using matrix-inversion by some efficient techniques like Cholesky [13], [18] or using eigen-decompositions. These two kinds of computations are different in the precondition update formula.

Matrix-inversion. For the matrix-inversion based damping technique [20], the update formula is given:

$$(F_i+\gamma I)^{-1}\nabla\bar{\mathcal{L}}_i \approx (G_i + \frac{\sqrt{\gamma}}{\pi_i}I)^{-1}\nabla\bar{\mathcal{L}}_i(A_{i-1}+\pi_i\sqrt{\gamma}I)^{-1}, \tag{14}$$

where A_{i-1} and G_i are KFs, $\nabla\bar{\mathcal{L}}_i$ is the aggregated gradient at layer i, and γ is the damping value. It means that we add KFs by a scaled damping value and then we invert the damped KFs for preconditioning. The scalar constant is given as $\pi_i = \sqrt{\mathrm{Tr}(A_{i-1})/\mathrm{Dim}(A_{i-1})}/\sqrt{\mathrm{Tr}(G_i)/\mathrm{Dim}(G_i)}$, which minimizes the approximation error of applying damping into KFs, and works better than the choice of $\pi_i = 1$ in practice.

Eigen-decomposition. Regarding the eigen-decomposition based damping technique [16], [21], the update formula should be changed to:

$$(F_i+\gamma I)^{-1}\nabla\bar{\mathcal{L}}_i = Q_G\frac{Q_G^T\nabla\bar{\mathcal{L}}_i Q_A}{v_G v_A^T+\gamma}Q_A^T, \tag{15}$$

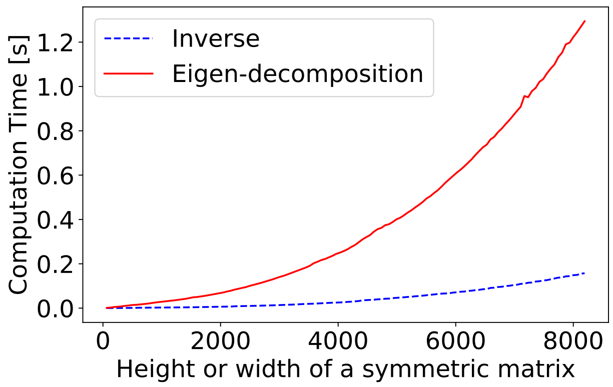


Fig. 6: Computation time comparison between matrix inverse and eigen-decomposition on an Nvidia RTX2080Ti GPU.

where $A_{i-1} = Q_A \Lambda_A Q_A^T$ and $G_i = Q_G \Lambda_G Q_G^T$ are the eigen-decompositions of KFs, v_A and v_G are the diagonals of Λ_A and Λ_G, and $\nabla \bar{\mathcal{L}}_i$ is the aggregated gradient of layer i to be preconditioned.

In our implementation, we support these two kinds of computations in the preconditioning. By default, we use the eigen-decomposition based damping method to conduct our experiments in the next section because it could provide more stable results [16], [17], [21]. We use the cholesky inverse and symmetric eigen-decomposition functions in PyTorch's linalg library. In practice, it is found that eigen-decomposition is more expensive than matrix inversion, as shown in 6, especially for large matrices.

6 Experimental Studies

In this section, we present the experimental studies with several parts. First, we clarify our experimental settings. Second, we compare the convergence performance of DP-KFAC with baselines including S-SGD and MPD-KFAC which is highly optimized in KAISA [17], on different applications. Third, we dive into the per-iteration time efficiency and memory consumption of our DP-KFAC compared to state-of-the-arts. Fourth, we discuss the effects of different aspects including the mini-batch size, the number of GPUs, and the frequency of FIM updates.

6.1 Experimental Settings

System Configuration. We conduct our experiments on a 64-GPU cluster. It consists of 16 nodes connected 100Gb/s InfiniBand, and each node has 4 Nvidia RTX2080Ti GPUs (11GB RAM) connected by two Intel(R) Xeon(R) Gold 6230 CPUs, 512GB memory, and PCIe3.0x16. We use some common software including PyTorch-1.10.0, Horovod-0.21.0, CUDA-10.2, cuDNN-7.6, and NCCL-2.6.4. When running 4-worker experiments, we only use one node of the cluster.

Baselines. For the first-order algorithms, we use the well-tuned optimizers for particular models, i.e., SGD with momentum for CNNs and Adam [35] for Transformers. For the D-KFAC variants, we compare our DP-KFAC with KAISA[2] with COMM-OPT (KAISA-CO), KAISA with MEM-OPT (KAISA-MO) [17], SPD-KFAC [29] with pipelining, and THOR [18] with dynamically determining the FIM update frequency. By default, we use KAISA as KAISA-CO if not particularly specified.

2. The code is adopted from its official implementation at https://github.com/gpauloski/kfac_pytorch.

TABLE 3: Hyper-parameter settings. α is the base LR, BS is the per-GPU batch size, P is the number of workers, and WU is the number of LR warmup iterations. At decay epochs, LR is decayed by 10 times.

Model	α	BS	P	WU	Decay Epochs	Epoch
ResNet-110	0.1	128	4	98	35, 75, 90	100
VGG-16	0.1	128	4	98	35, 65, 80, 90	100
ResNet-50	0.0125	32	64	3130	25, 35, 40, 45, 50	55
Transformer	1e-6	128	8	4000	-	200
BERT	5e-6	4	8	0	-	3

DNN Models. To verify the convergence performance, we choose two modern types of DNNs including CNNs and Transformers similar to the works of KAISA [17] and THOR [18]. In CNNs, we choose three representative CNN models and datasets, that is ResNet-110 on the Cifar-10 [28] dataset, VGG-16 [36] on the Cifar-100 [28] dataset and ResNet-50 on the ImageNet [26] dataset. The Cifar-10 (or Cifar-100) dataset has 10 (or 100) classes with 50,000 training images and 10,000 validation images, and the ImageNet dataset spans 1,000 classes and contains ~1.3M training images and 50,000 validation images. In Transformers, we choose a Transformer [34] model on the Multi-30k dataset [37] and a BERT (finetune) [38] model on the SQuAD v1.1 [39] dataset. The Multi-30k dataset consists of a pair of images and their German and English captions with 29,000 training sentences and 1,014 validation sentences. As this dataset is smaller than the one used in [34], we instead use a shallow Transformer model with only two hidden layers as [8] to achieve better German-to-English translation results. The SQuAD represents a standard question answering dataset with more than 100K crowdsourced question-answer pairs. Since training BERT from scratch is extremely time-consuming, we choose to finetune a pre-trained BERT model on SQuAD using the Transformers library [40].

Hyper-parameters. For all algorithms (S-SGD and D-KFAC), we adopt the learning rate (LR) scheduling strategy from [1]. Specifically, we use the linear scaling rule that multiplies a base LR (α) by the number of workers (P) when applied data parallelism to train DNN models. In early stages of training (say warmup), we start from the base LR of α and enlarge it by a constant amount at each iteration such that it reaches $P \times \alpha$. After warmup, the LR decays by $1/10$ multiple times at given epochs for better convergence. All hyper-parameters that relate to LR are listed in Table 3. For other hyper-parameters, we use moving average parameters for gradients (say momentum) and KFs [20] of 0.9 and 0.95, respectively. We use S-SGD+ to denote the SGD (or Adam) algorithm that runs more epochs to reach the target accuracy, i.e., we train ResNet-110 and VGG-16 for 165 epochs (LR decays at epochs 82 and 123), and ResNet-50 for 90 epochs (LR decays at epochs 30, 60, and 80), Transformer for 400 epochs, and BERT for 6 epochs. For D-KFAC related algorithms, we use the damping γ with 0.03 for different models, except 0.002 for ResNet-50.

6.2 Convergence Performance

We compare the convergence performance of our DP-KFAC with KAISA[3] and S-SGD (with momentum) or Adam. For each application, the preconditioners are applied in convolution layers and linear layers in both KAISA and DP-KFAC.

CNNs. The convergence curves (top-1 validation accuracy vs. epochs) on CNNs are shown in Fig. 7(a) and (b). The results indicate that DP-KFAC converges almost the same with KAISA, and they both outperform S-SGD at the same number of epochs. In VGG-16 running 100 epochs, both DP-KFAC and KAISA achieve the target accuracy of 73% at around 90 epochs, while S-SGD only converges to 70.58%. In ResNet-50, DP-KFAC and KAISA achieve the target accuracy of 75.9% at epoch 45, while S-SGD only converges to 74.31%. As expected, since the second-order methods can converge faster than the first-order methods, S-SGD needs to run more iterations to achieve the target accuracy than the K-FAC methods. The final converged accuracy is shown in Table 4, where S-SGD, KAISA, and DP-KFAC run with the same number of epochs as shown in Table 3 while S-SGD+ runs extra epochs (165 epochs on ResNet-110 and VGG-16, and 90 epochs on ResNet-50). The results show that our DP-KFAC achieves very close model accuracy with KAISA, and both of them can achieve the target accuracy in a much fewer number of iterations than SGD.

Transformers. We report the convergence curves (validation accuracy vs. epochs, and validation loss vs. iterations) on Transformer and BERT models in Fig. 7(c) and Fig. 7(d), respectively. The results validate that DP-KFAC converges as fast as KAISA on Transformer-based models for different NLP tasks, while Adam converges slower than K-FAC algorithms, and achieves poorer performance with the same training iterations. For final converged results, we use the BLEU score and F1 score to measure the quality of model predictions in translation and question answering tasks, respectively. As shown in Table 4, DP-KFAC and KAISA achieve very close BLEU scores (38.66 and 38.47) on the Multi-30k de-en dataset, both of which are higher than the results of Adam (36.69) and S-SGD+ (37.15). On the SQuAD dataset, DP-KFAC and KAISA have almost the same F1 scores (87.86 and 87.73), which is better than Adam (86.05) and S-SGD+ (87.16). From above results, it has shown that K-FAC algorithms can achieve better convergence performance than the first-order counterpart in both CV and NLP domains, and our DP-KFAC algorithm can preserve the model accuracy compared to the KAISA without distributed preconditioning.

6.3 Training Efficiency

As we have shown the convergence performance of our DP-KFAC, we would like to demonstrate its per-iteration training efficiency and memory consumption compared to existing highly optimized solutions including KAISA-CO, KAISA-MO [17], and SPD-KFAC [29]. Specifically, KAISA-CO and SPD-KFAC broadcast the inverses of KFs and then compute the preconditioned gradients in all workers, while KAISA-MO and DP-KFAC precondition the gradients distributively and then broadcast the preconditioned gradients among all workers. Except the models listed in Table 3, we select two more popular CNNs, DenseNet-201 [41] and Inception-v4 [42], on the ImageNet dataset and the per-GPU mini-batch size is set as 16 to fully utilize GPU memory.

3. Since KAISA and THOR are with the same computation and communication schemes, and their main difference is the update interval of the FIMs, they should have similar convergence performance as shown in [18]. So we only run KAISA to compare the convergence.

TABLE 4: Best validation accuracy of different optimization algorithms under the configured epochs as shown in Table 3. S-SGD+ indicates that the models are trained with S-SGD (or Adam) using more epochs to improve model accuracy. The top-1 accuracy (%) is used to measure CNNs, while BLEU and F1 metrics are used to measure Transformer and BERT, respectively. We measure 3 times for each experiment and report their mean and std.

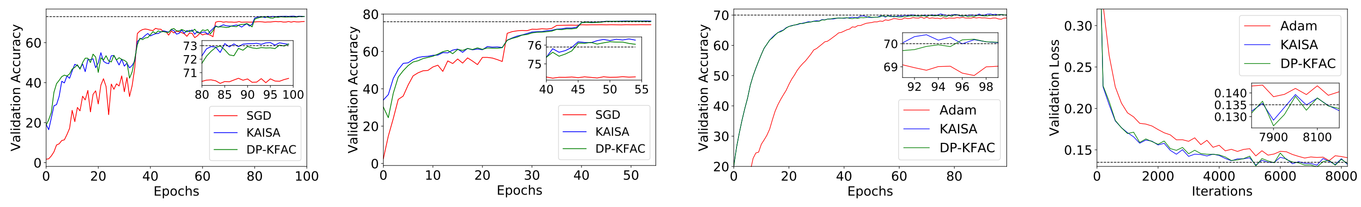
Model	S-SGD	S-SGD+	KAISA	DP-KFAC
ResNet-110	93.25±0.08	93.66±0.09	93.89±0.12	**93.99±0.10**
VGG-16	70.44±0.13	**73.23±0.10**	73.21±0.09	73.12±0.08
ResNet-50	74.31±0.10	**76.42±0.03**	76.10±0.06	76.34±0.12
Transformer	36.69±0.05	37.15±0.05	38.47±0.26	**38.66±0.18**
BERT	86.05±0.03	87.16±0.06	87.73±0.24	**87.86±0.07**

The per-iteration time and GPU memory consumption are shown in Table 5. The results show that our DP-KFAC outperforms all the other state-of-the-art algorithms in per-iteration time and GPU memory consumption in all tested models. Particularly, on the relatively small models, ResNet-110 and VGG-16, our DP-KFAC runs 14%-68% faster than KAISA (including both KAISA-CO and KAISA-MO) and SPD-KFAC. On the large-scale models with ImageNet, our DP-KFAC significantly reduces the iteration time by 1.5×-2.3× compared to KAISA and SPD-KFAC on the 64-GPU cluster. Regarding the Transformers, DP-KFAC runs 21%-50% faster than KAISA and SPD-KFAC. In terms of occupied GPU memory, DP-KFAC consumes 26%, 15%, and 42% less memory than KAISA-CO, KAISA-MO, and SPD-KFAC. Note SPD-KFAC consumes the most memory with tensor fusion to reduce the iteration time. In addition, we notice KAISA-MO outperforms KAISA-CO in both time and memory costs, this is because we consider the per-iteration optimization, in which the iterative communication overhead of KAISA-MO (PredComm) is smaller than KAISA-CO (InverseComm). Because KAISA-CO is motivated by using stale FIM, which reduces the frequency of InverseComm, we will study the effects of the update frequency of KFs in the discussions.

6.4 Time Breakdowns

Time Performance. To understand the time consumption in different D-KFAC algorithms, We dive into the time breakdowns of one training iteration between KAISA-MO, SPD-KFAC, and DP-KFAC, on three representative models: ResNet-50, DenseNet-201, and BERT. The results are given in Fig. 8.

First, the feed-forward and back-propagation computations (FF&BP) and gradient communications (GradComm) are the same on different algorithms, because the S-SGD part is implemented based on Horovod and it is independent from the K-FAC preconditioner. Second, in terms of KF



(a) VGG-16 on Cifar-100 (b) ResNet-50 on ImageNet (c) Transformer on Multi-30k (d) BERT on SQuAD

Fig. 7: Convergence performance comparison between DP-KFAC and SGD/KAISA on different DNN models and datasets. The black dashed line represents the target accuracy.

TABLE 5: Average second-order iteration time (in seconds) and occupied GPU memory (in GB) measured by 100 iterations.

Model	Dataset	# GPUs	KAISA-CO		KAISA-MO		SPD-KFAC		DP-KFAC	
			Time	Mem	Time	Mem	Time	Mem	Time	Mem
ResNet-110	Cifar-10	4	0.865±0.071	3.3	0.840±0.027	3.2	0.743±0.008	3.4	**0.514±0.007**	**3.2**
VGG-16	Cifar-100	4	1.394±0.007	4.3	1.305±0.008	4.0	1.207±0.006	4.9	**1.062±0.005**	**3.5**
ResNet-50	ImageNet	64	1.375±0.010	8.8	1.204±0.002	8.0	1.125±0.005	9.8	**0.744±0.002**	**7.0**
DenseNet-201			2.030±0.011	9.3	1.624±0.011	8.1	1.609±0.005	10.7	**0.894±0.008**	**6.6**
Inception-v4			1.703±0.005	7.5	1.402±0.011	6.5	1.305±0.008	8.8	**0.803±0.003**	**5.1**
Transformer	Multi-30k	8	1.390±0.016	7.6	1.274±0.017	7.3	1.348±0.028	8.3	**1.049±0.033**	**6.8**
BERT	SQuAD	8	1.812±0.009	8.1	1.644±0.005	7.1	1.703±0.009	8.9	**1.211±0.007**	**6.1**

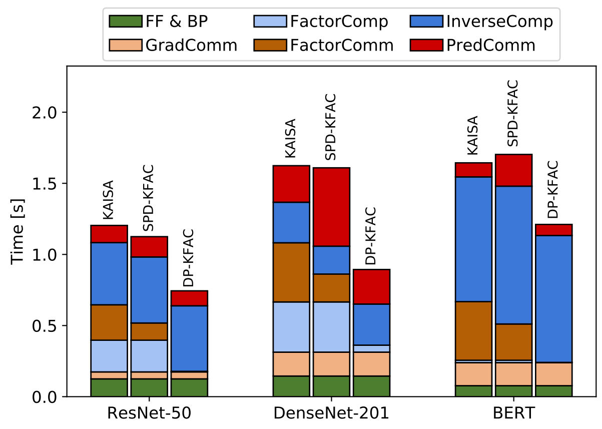


Fig. 8: Time breakdown comparison between different D-KFAC algorithms with eigen-decomposition based damping.

computations (FactorComp), the FactorComp cost in DP-KFAC is significantly reduced compared to KAISA and SPD-KFAC. Note that the cost of FactorComp on the BERT is small on all algorithms since the batch size of BERT for estimating KFs is only 4. Third, for KF communications (FactorComm), though SPD-KFAC can reduce some FactorComm overheads by overlapping the computation and communication tasks, our DP-KFAC can totally eliminate FactorComm. Hence, DP-KFAC has zero FactorComm cost and small FactorComp cost, which contribute to performance improvement of DP-KFAC. However, the InverseComp cost could be the bottleneck even when the eigen-decomposition computations are distributively performed in DP-KFAC. One alternative is to use the matrix-inversion damping strategy instead, as inverting damped KFs is more time efficient than eigen-decomposition computations.

Therefore, we provide time breakdowns of matrix-inversion damping based D-KFAC algorithms in Fig. 9 with the same settings. It shows that the matrix-inversion based damping can in average reduce the iteration time of DP-KFAC by 2.2x compared to those using eigen-decomposition based damping. Besides, DP-KFAC outperforms KAISA and SPD-KFAC as it can also reduce the FactorComp cost and remove the FactorComm overhead. Furthermore, using matrix-inversion in InverseComp makes the iteration time of DP-KFAC be much closer to that of S-SGD than using eigen-decomposition. Specifically, the running time of the second-order iteration of DP-KFAC is around 1.7×-2× slower than the first-order iteration time of S-SGD.

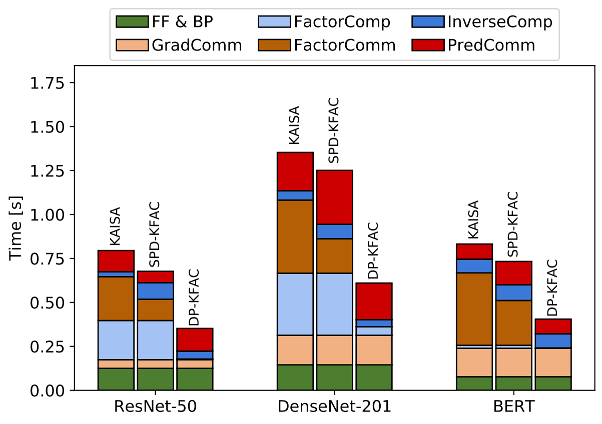


Fig. 9: Time breakdown comparison between different D-KFAC algorithms with matrix-inversion based damping.

Convergence Performance. Besides, we provide the validation accuracy of KAISA (inv) and DP-KFAC (inv) with matrix-inversion based damping on different datasets following the same configurations as shown in Table 3. The results are given in Table 6. It shows that DP-KFAC (inv) can maintain the validation accuracy over different datasets compared to KAISA (inv) without distributed preconditioning. As DP-KFAC (inv) can achieve the target accuracy

TABLE 6: Best validation accuracy of KAISA and DP-KFAC with eigen-decomposition or matrix-inversion based damping.

Algorithm	ResNet-110	VGG-16	Transformer	BERT
KAISA (eigen)	93.89	73.21	38.47	87.73
DP-KFAC (eigen)	93.99	73.12	38.66	87.86
KAISA (inv)	93.83	73.26	38.17	87.75
DP-KFAC (inv)	**94.35**	**73.27**	**38.90**	**88.00**

as well as DP-KFAC (eigen), it is suggested to use DP-KFAC (inv) to further accelerate the training process with lower computation cost. The matrix-inversion based K-FAC algorithms achieved worse validation accuracy compared to using eigen-decomposition [16], [17], but in this work, we find that the scalar π_i (see Eq. 14) is helpful to stabilize the training process. Apart from that, we tune the running average value ξ (see Eq. 10), e.g., set $\xi = 0.05$ on Cifar-100, for better performance.

6.5 Discussions

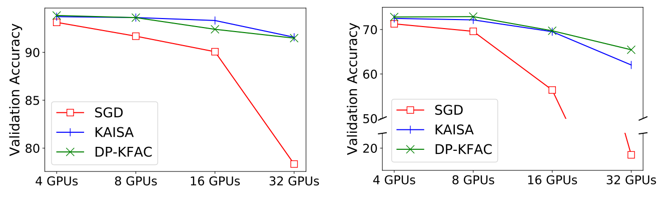
In this subsection, we would like to discuss the learning properties of our DP-KFAC algorithm with different aspects, including the per-GPU batch size, the number of GPUs, the damping value, and the frequency of FIM updates. We finally compare the end-to-end training time of DP-KFAC to SGD and existing D-KFAC algorithms by training the ResNet-50 model with stale FIMs.

Effects of local mini-batch sizes. In MPD-KFAC, KFs are computed through the global mini-batch size for the purpose of approximating the empirical FIMs over a long-term run. Intuitively, when using the local mini-batch size in DP-KFAC to compute KFs over a large number of iterations, the expectation of KFs become the same as that with a global mini-batch size. Consequently, different local mini-batch sizes may have little impact on the convergence performance. We further conduct experiments (100 epochs) to examine this problem when training with different local mini-batch sizes compared with SGD and KAISA as shown in Table 7. The results show that: 1) DP-KFAC achieves comparable convergence performance with KAISA under different mini-batch sizes; 2) Both DP-KFAC and KAISA significantly outperform SGD with a fixed number of iterations, especially on VGG-16; 3) With an increased mini-batch size, the convergence performance tends to decrease in all algorithms, which could be related to the generalization gap in large-batch training [6], [43]. In distributed training, data may be partitioned across multiple nodes, which makes each node have only a part of data for calculating gradients [44]. It may also have some impacts on the convergence on SGD [44] and KFAC algorithms. We will leave this as our future study.

Effects of the number of GPUs in convergence. The key communication and computation workloads reduction in our DP-KFAC is to estimate the KFs with local mini-batch data and construct the FIMs distributively without factor communications, while MPD-KFAC algorithms estimate the KFs with global mini-batch data. Since the global mini-batch size is P times larger than local mini-batch size, we hereby report the training performance when the number of workers P changes. We run SGD, KAISA, and DP-KFAC

TABLE 7: Validation accuracy (%) with different local mini-batch sizes in SGD, KAISA, and DP-KFAC running on 4 GPUs. We measure 3 times for each experiment and report their mean and std. Bold texts are the best validation accuracy among the compared algorithms.

Model	Algorithm	Local mini-batch size			
		32	64	128	256
ResNet-110	SGD	93.78±.14	93.43±.12	93.04±.48	91.31±.37
	KAISA	93.84±.11	93.95±.14	94.0±.14	**93.79±.16**
	DP-KFAC	**94.03±.17**	**94.04±.17**	**94.02±.08**	93.79±.20
VGG-16	SGD	72.14±.12	71.67±.45	70.32±.49	69.54±.51
	KAISA	72.94±.39	**73.5±.27**	**73.21±.09**	72.67±.21
	DP-KFAC	**73.02±.30**	73.34±.20	73.12±.08	**72.82±.48**



(a) ResNet-110 on Cifar-10 (b) VGG-16 on Cifar-100

Fig. 10: Validation accuracy comparison between DP-KFAC and SGD/KAISA on ResNet-110 and VGG-16 with different number of GPUs.

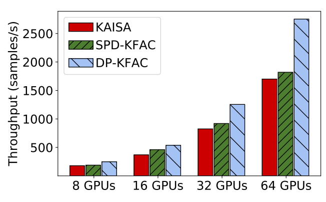
on ResNet-110 and VGG-16 models, with different number of GPUs. We use a damping γ of 0.001 to avoid possible divergence, and keep all other hyper-parameters fixed. The validation accuracy results are shown in Figure 10. The results indicate that our DP-KFAC achieves comparable performance to KAISA on two models with different number of GPUs. As expected, they both outperform SGD in all tested cases. As the number of GPUs increases, the performance of all algorithms with the same epochs drops in different degrees. This is because the number of iterations is also reduced with large data parallelism [1]. The empirical results show that D-KFAC could scale better to larger batch size than SGD.

Effects of the number of GPUs in throughput. To validate the scalability of DP-KFAC when the number of GPUs changes, we also compare the system throughput on ResNet-50 and DenseNet-201 models. The results are shown in Figure 11, indicating that our DP-KFAC can achieve the best system throughput in all tested cases, and it scales much better than others when the number of GPUs increases. It is seen that with more GPUs, our DP-KFAC has higher benefits than others since the time cost of data aggregation of KFs is proportional to the number of workers in both SPD-KFAC and KAISA, but DP-KFAC has no need to communicate KFs.

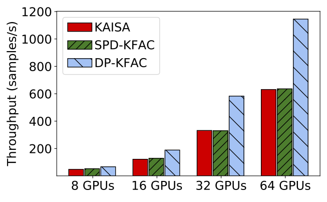
To be specific, we provide the time breakdowns of training ResNet-50, DenseNet-201, and Inception-v4 models with 16 GPUs in Figure 12. It shows that DP-KFAC can outperform KAISA and SPD-KFAC by reducing the FactorComp cost and removing the FactorComm stage.

.

Effects of damping and the update frequency of KFs. Damping and the update frequency of KFs are two critical hyper-parameters that need to be tuned in K-FAC algo-



(a) ResNet-50



(b) DenseNet-201

Fig. 11: System throughput comparison between DP-KFAC and KAISA/SPD-KFAC on the ImageNet dataset.

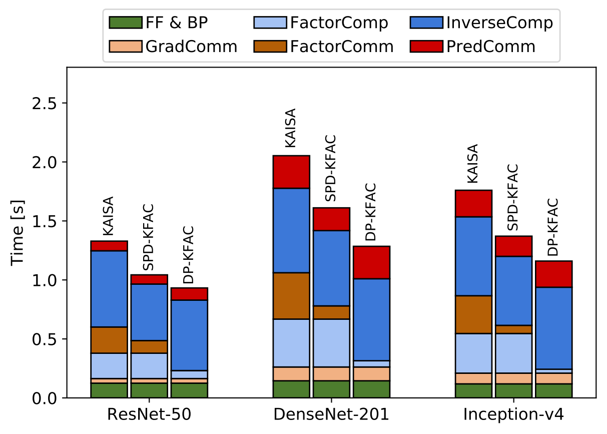
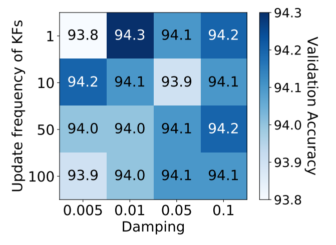
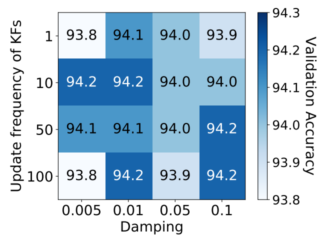


Fig. 12: Time breakdown comparison between different D-KFAC algorithms with 16 GPUs.



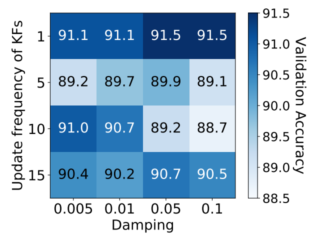
	0.005	0.01	0.05	0.1
1	93.8	94.3	94.1	94.2
10	94.2	94.1	93.9	94.1
50	94.0	94.0	94.1	94.2
100	93.9	94.0	94.1	94.1

(a) KAISA, 4 GPUs



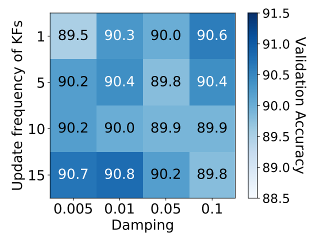
	0.005	0.01	0.05	0.1
1	93.8	94.1	94.0	93.9
10	94.2	94.2	94.0	94.0
50	94.1	94.1	94.0	94.2
100	93.8	94.2	93.9	94.2

(b) DP-KFAC, 4 GPUs



	0.005	0.01	0.05	0.1
1	91.1	91.1	91.5	91.5
5	89.2	89.7	89.9	89.1
10	91.0	90.7	89.2	88.7
15	90.4	90.2	90.7	90.5

(c) KAISA, 32 GPUs



	0.005	0.01	0.05	0.1
1	89.5	90.3	90.0	90.6
5	90.2	90.4	89.8	90.4
10	90.2	90.0	89.9	89.9
15	90.7	90.8	90.2	89.8

(d) DP-KFAC, 32 GPUs

Fig. 13: The effects of damping and the update frequency of KFs on ResNet-110 with KAISA and DP-KFAC.

rithms [20]. Thus, we study their effects on both KAISA and DP-KFAC of training ResNet-110 with 4 GPUs and 32 GPUs. The hyper-parameter tuning spaces and their corresponding validation accuracy are given in Fig. 13. The results show that DP-KFAC can maintain very close performance to KAISA even over different hyper-parameter settings. Specifically, over the 16 runs for both KAISA and DP-KFAC, their average accuracies are 94.07% and 94.05% on 4 GPUs, and 90.28% and 90.17% on 32 GPUs, respectively. Using 32 GPUs, it shows that the performance drop exists over different hyper-parameters caused by the effects of large-batch training. Furthermore, we find that both KAISA and DP-KFAC can maintain very close accuracy with stale FIM and even large update frequency of KFs (e.g., updating KFs every 100 iterations). This contradicts the observation that skipping FIM updates will affect the convergence performance [16], and we contribute it to that the possible performance drop of using stale FIM depends on datasets and models. As it seems that no simple correlation between model accuracy and hyper-parameters (damping and the update frequency of KFs) is exhibited at least for training ResNet-110 on Cifar-10, we leave it as a future work that is worth to be explored.

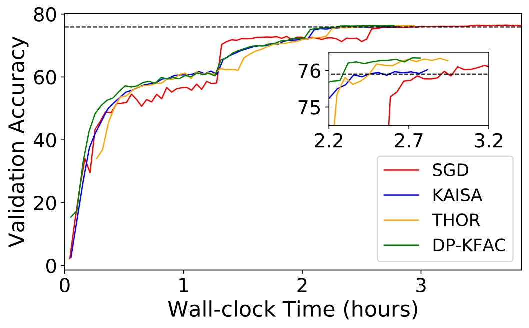


Fig. 14: Validation accuracy/loss vs. wall-clock time with stale FIMs on ResNet-50. The black dashed line represents the target accuracy (i.e., 75.9%).

Training with Stale FIM. In practice, it is common to reduce the frequency of FIM updates in K-FAC algorithms [13], [16], [17], [18], [20] to accelerate the training process. We hereby verify our DP-KFAC using the stale FIM update with the same configuration as KAISA [17] using $K\_freq = 500$ (the recompute frequency of eigen-decomposition) and $F\_freq = 50$ (the update frequency of KFs). We keep the same hyper-parameter settings in all runs as Table 3. We also reproduce THOR [18] for fair comparison on our 64-GPU cluster in training ResNet-50, which utilizes dynamic update intervals (F_freq and K_freq are both dynamically increased from 1 to 5000 in the first 10 epochs and $K\_freq = F\_freq = 5000$ in the later epochs). The convergence results (top-1 validation accuracy vs. wall-clock time) using stale FIMs are shown in Fig. 14[4]. It is seen that our DP-KFAC achieves faster convergence performance than both KAISA and THOR in the end-to-end training with stale FIMs. Specifically, DP-KFAC reaches the target accuracy of 75.9% in just 8350 seconds, which is about 500, 500, and 2450 seconds faster than KAISA, THOR, and SGD, respectively. Therefore, our DP-KFAC is 23% faster than SGD, while achieving 6% improvement over KAISA and THOR by reducing the frequency of FIM updates. Though the improvement of DP-KFAC over other D-KFAC algorithms is somehow weakened in this case, we argue

4. We also conducted experiments with LARS [45] for better convergence, which requires 70 epochs to reach the target accuracy.

TABLE 8: Throughput comparison of different K-FAC algorithms with stale FIM. The results are measured on ImageNet using 64 GPUs. Each algorithm maximizes the local batch size (BS) to fully utilize the GPU memory and updates KFs every 50 iterations (i.e., $K\_freq = F\_freq = 50$).

Model	KAISA		DP-KFAC		Speedup
	BS	Throughput	BS	Throughput	
ResNet-50	50	7729.5	60	9099.5	1.18x
DenseNet-201	24	2160.3	32	3245.6	1.50x
Inception-v4	40	3956.7	50	4984.4	1.26x

that the distributed preconditioning technique are of significance for two reasons: (1) the per-iteration training time optimization is necessary in the cases when skipping KF updates adversely affects the model accuracy, and (2) it can save the GPU memory for storing KF results distributively and the saved memory allows DP-KFAC to train with larger local batch size than KAISA, which can further improve its throughput.

To verify the benefit of memory saving of DP-KFAC, we compare the system throughput of KAISA (MO) and DP-KFAC (eigen) using different local batch sizes to fully utilize the GPU memory. The results are given in Table 8. It shows that DP-KFAC can use larger local batch size than KAISA with reduced memory footprint, and DP-KFAC achieves an average of 31.3% speedup over KAISA in the end-to-end training with stale FIM.

7 Related Work

Recent years there have been some attempts trying to make second-order optimizations in DNN training be practical [10], [20], [46], [47], [48], [49], among which the K-FAC based [20] algorithms have been successfully applied in large-scale training [13], [15], [17], [18]. We mainly introduce the K-FAC related studies.

7.1 K-FAC Algorithms

The second-order algorithms utilize the curvature information to precondition the first-order gradient, which often leads to faster optimizations [10]. However, the preconditioning matrix like FIM is too big to construct and invert in DNNs. In [20], the K-FAC algorithm is introduced to approximate the FIMs layer-wisely using Kronecker factors, which provides a relatively efficient way to approximate FIMs [20], [21], [22]. In [23], [50], alternative factored methods have been proposed to further reduce the approximation error of FIMs. Alternatively, in [51], [52], the authors propose more efficient approximations with faster inversion. These works do not focus on the distributed optimization, but they are orthogonal to our work.

7.2 D-KFAC Algorithms

The distributed optimization algorithms using K-FAC (D-KFAC) [13], [14], [15], [16], [17] enable the second-order optimization to converge faster than the SGD counterpart on distributed GPU clusters. The implementations of D-KFAC algorithms span from the centralized architecture [24] to the decentralized architecture [13], [14], [15], [16], [17], [18]. In the decentralized architecture, it typically uses model parallelism to invert or eigen-decompose [16], [17] KFs of different layers on different workers, and it is able to train the large-scale models faster than SGD in the the end-to-end wall-clock time on GPU clusters. There are also some communication scheduling strategies [18], [29] being proposed to alleviate the communication overheads in D-KFAC through pipelining techniques [29] or using dynamic update strategies [18] to improve the scaling efficiency of the distributed system. In this paper, our work is aligned with this direction.

8 Conclusion

In this paper, we first analysed the computation and communication bottlenecks in the existing distributed K-FAC (D-KFAC) algorithms. We then introduced our DP-KFAC with distributed preconditioning (DP) that distributes the preconditioning tasks of different layers of a DNN to different GPUs to accelerate training. DP-KFAC enjoys the similar convergence property as the existing D-KFAC algorithms, which is well verified through extensive experiments. With DP-KFAC, the computation, communication, and memory costs can be reduced without sacrificing the convergence performance. We conducted extensive experiments with various modern CNNs and Transformers on different datasets on a 64-GPU cluster. The experimental results showed that our DP-KFAC is much more time- and-memory efficient than state-of-the-art methods without affecting the model accuracy.

Acknowledgment

We thank the reviewers for their valuable comments. The research was supported in part by RGC RIF grant R6021-20 and RGC GRF grant under contracts 16209120 and 16200221. Wei Wang was supported by RGC GRF under contracts 16213120 and 16202121.

References

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

[2] H. Zhang, Z. Zheng, S. Xu, W. Dai, Q. Ho, X. Liang, Z. Hu, J. Wei, P. Xie, and E. P. Xing, "Poseidon: An efficient communication architecture for distributed deep learning on GPU clusters," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 2017, pp. 181–193.

[3] X. Jia, S. Song, S. Shi, W. He, Y. Wang, H. Rong, F. Zhou, L. Xie, Z. Guo, Y. Yang, L. Yu, T. Chen, G. Hu, and X. Chu, "Highly scalable deep learning training system with mixed-precision: Training ImageNet in four minutes," in *Proc. of Workshop on Systems for ML and Open Source Software, collocated with NeurIPS 2018*, 2018.

[4] Y. Peng, Y. Zhu, Y. Chen, Y. Bao, B. Yi, C. Lan, C. Wu, and C. Guo, "A generic communication scheduler for distributed DNN training acceleration," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 16–29.

[5] Y. You, J. Li, S. Reddi, J. Hseu, S. Kumar, S. Bhojanapalli, X. Song, J. Demmel, K. Keutzer, and C.-J. Hsieh, "Large batch optimization for deep learning: Training BERT in 76 minutes," in *International Conference on Learning Representations*, 2020.

[6] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang, "On large-batch training for deep learning: Generalization gap and sharp minima," in *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*, 2017.

[7] E. Hoffer, I. Hubara, and D. Soudry, "Train longer, generalize better: closing the generalization gap in large batch training of neural networks," in *Advances in Neural Information Processing Systems*, 2017, pp. 1731–1741.

[8] C. J. Shallue, J. Lee, J. Antognini, J. Sohl-Dickstein, R. Frostig, and G. E. Dahl, "Measuring the effects of data parallelism on neural network training," *Journal of Machine Learning Research*, vol. 20, pp. 1–49, 2019.

[9] S.-I. Amari, "Natural gradient works efficiently in learning," *Neural computation*, vol. 10, no. 2, pp. 251–276, 1998.

[10] L. Bottou, F. E. Curtis, and J. Nocedal, "Optimization methods for large-scale machine learning," *Siam Review*, vol. 60, no. 2, pp. 223–311, 2018.

[11] G. Zhang, J. Martens, and R. B. Grosse, "Fast convergence of natural gradient descent for over-parameterized neural networks," in *NeurIPS*, 2019.

[12] J. Martens, "New insights and perspectives on the natural gradient method," *Journal of Machine Learning Research*, vol. 21, pp. 1–76, 2020.

[13] K. Osawa, Y. Tsuji, Y. Ueno, A. Naruse, R. Yokota, and S. Matsuoka, "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*, 2019, pp. 12 359–12 367.

[14] K. Osawa, Y. Tsuji, Y. Ueno, A. Naruse, C. Foo, and R. Yokota, "Scalable and practical natural gradient for large-scale deep learning," *IEEE Transactions on Pattern Analysis & Machine Intelligence*, 2020.

[15] Y. Ueno, K. Osawa, Y. Tsuji, A. Naruse, and R. Yokota, "Rich information is affordable: A systematic performance analysis of second-order optimization using K-FAC," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2020, pp. 2145–2153.

[16] J. G. Pauloski, Z. Zhang, L. Huang, W. Xu, and I. T. Foster, "Convolutional neural network training with distributed K-FAC," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020, pp. 1–14.

[17] J. G. Pauloski, Q. Huang, L. Huang, S. Venkataraman, K. Chard, I. Foster, and Z. Zhang, "Kaisa: An adaptive second-order optimizer framework for deep neural networks," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021.

[18] M. Chen, K. Gao, X. Liu, Z. Wang, N. Ni, Q. Zhang, L. Chen, C. Ding, Z. Huang, M. Wang *et al.*, "THOR, trace-based hardware-driven layer-oriented natural gradient descent computation," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, no. 8, 2021, pp. 7046–7054.

[19] S.-i. Amari, J. Ba, R. B. Grosse, X. Li, A. Nitanda, T. Suzuki, D. Wu, and J. Xu, "When does preconditioning help or hurt generalization?" in *International Conference on Learning Representations*, 2021.

[20] J. Martens and R. Grosse, "Optimizing neural networks with kronecker-factored approximate curvature," in *International conference on machine learning*, 2015, pp. 2408–2417.

[21] R. Grosse and J. Martens, "A kronecker-factored approximate fisher matrix for convolution layers," in *International Conference on Machine Learning*, 2016, pp. 573–582.

[22] J. Martens, J. Ba, and M. Johnson, "Kronecker-factored curvature approximations for recurrent neural networks," in *International Conference on Learning Representations*, 2018.

[23] T. George, C. Laurent, X. Bouthillier, N. Ballas, and P. Vincent, "Fast approximate natural gradient descent in a kronecker factored eigenbasis," *Advances in Neural Information Processing Systems*, vol. 31, pp. 9550–9560, 2018.

[24] J. Ba, R. Grosse, and J. Martens, "Distributed second-order optimization using kronecker-factored approximations," in *International Conference on Learning Representations*, 2017.

[25] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[26] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 2009, pp. 248–255.

[27] "Mlperf training v1.1 results," https://mlcommons.org/en/news/mlperf-training-v11/, accessed: 2022-03-15.

[28] A. Krizhevsky, "Learning multiple layers of features from tiny images," *Citeseer*, 2009.

[29] S. Shi, L. Zhang, and B. Li, "Accelerating distributed k-fac with smart parallelism of computing and communication tasks," in *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2021.

[30] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server," in *11th USENIX OSDI*, 2014, pp. 583–598.

[31] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[32] A. Sergeev and M. Del Balso, "Horovod: fast and easy distributed deep learning in tensorflow," *arXiv preprint arXiv:1802.05799*, 2018.

[33] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, 2016, pp. 265–283.

[34] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[35] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

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

[37] D. Elliott, S. Frank, K. Sima'an, and L. Specia, "Multi30k: Multilingual english-german image descriptions," in *5th Workshop on Vision and Language*. Association for Computational Linguistics (ACL), 2016, pp. 70–74.

[38] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, 2019, pp. 4171–4186.

[39] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang, "Squad: 100,000+ questions for machine comprehension of text," in *EMNLP*, 2016.

[40] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, and J. Brew, "Transformers: State-of-the-art natural language processing," in *EMNLP*, 2020.

[41] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 4700–4708.

[42] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, "Inception-v4, inception-resnet and the impact of residual connections on learning," in *Proc. of The 31st AAAI*, 2017.

[43] T. Lin, L. Kong, S. Stich, and M. Jaggi, "Extrapolation for large-batch training in deep learning," in *International Conference on Machine Learning*. PMLR, 2020, pp. 6094–6104.

[44] T. T. Nguyen, F. Trahay, J. Domke, A. Drozd, E. Vatai, J. Liao, M. Wahib, and B. Gerofi, "Why globally re-shuffle? revisiting data shuffling in large scale deep learning," in *IEEE International Parallel & Distributed Processing Symposium*, 2022.

[45] Y. You, Z. Zhang, C.-J. Hsieh, J. Demmel, and K. Keutzer, "ImageNet training in minutes," in *Proceedings of the 47th International Conference on Parallel Processing*, 2018, pp. 1–10.

[46] J. Martens *et al.*, "Deep learning via hessian-free optimization." in *ICML*, vol. 27, 2010, pp. 735–742.

[47] X.-T. Yuan and P. Li, "On convergence of distributed approximate newton methods: Globalization, sharper bounds and beyond." *J. Mach. Learn. Res.*, vol. 21, pp. 206–1, 2020.

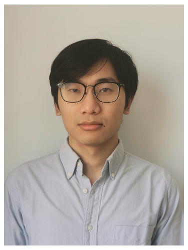
[48] V. Thomas, F. Pedregosa, B. Merriënboer, P.-A. Manzagol, Y. Bengio, and N. Le Roux, "On the interplay between noise and curvature and its effect on optimization and generalization," in *International Conference on Artificial Intelligence and Statistics*. PMLR, 2020, pp. 3503–3513.

[49] Z. Yao, A. Gholami, S. Shen, M. Mustafa, K. Keutzer, and M. Mahoney, "Adahessian: An adaptive second order optimizer for machine learning," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, no. 12, 2021, pp. 10 665–10 673.

[50] K. Gao, X. Liu, Z. Huang, M. Wang, Z. Wang, D. Xu, and F. Yu, "A trace-restricted kronecker-factored approximation to natural gradient," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, no. 9, 2021, pp. 7519–7527.

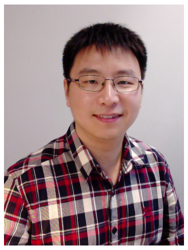
[51] Z. Tang, F. Jiang, M. Gong, H. Li, Y. Wu, F. Yu, Z. Wang, and M. Wang, "Skfac: Training neural networks with faster kronecker-factored approximate curvature," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2021, pp. 13 479–13 487.

[52] S. Soori, B. Can, B. Mu, M. Gürbüzbalaban, and M. M. Dehnavi, "Tengrad: Time-efficient natural gradient descent with exact fisher-block inversion," *arXiv preprint arXiv:2106.03947*, 2021.



Lin Zhang received the B.S. degree at School of Electrical Engineering and Automation from Zhejiang University in 2018. He is currently pursuing the Ph.D. degree in the Department of Computer Science and Engineering at the Hong Kong University of Science and Technology. His research interests include machine learning systems and applications, with a special focus on second-order optimization methods, and unsupervised learning on graphs.

Shaohuai Shi received a B.E. degree in software engineering from South China University of Technology, P.R. China, in 2010, an MS degree in computer science from Harbin Institute of Technology, P.R. China in 2013, and a Ph.D. degree in computer science from Hong Kong Baptist University in 2020. He is currently an assistant professor in the School of Computer Science and Technology at Harbin Institute of Technology, Shenzhen. His research interests include GPU computing and machine learning systems. He is a member of the IEEE.



Wei Wang received his B.Engr. and M.Engr. degrees from the Department of Electrical Engineering, Shanghai Jiao Tong University, China, in 2007 and 2010, respectively and his Ph.D. degree from the Department of Electrical and Computer Engineering, University of Toronto, Canada, in 2015. Since 2015, he has been with the Department of Computer Science and Engineering at the Hong Kong University of Science and Technology (HKUST), where he is currently an Associate Professor. He is also affiliated with the HKUST Big Data Institute. Dr. Wang's research interests cover the broad area of distributed systems, with focus on serverless computing, machine learning systems, and cloud resource management. He published extensively in the premier conferences and journals of his fields. His research has won the Best Paper Runner Up awards of IEEE ICDCS 2021 and USENIX ICAC 2013.

Bo Li is a professor in the Department of Computer Science and Engineering, Hong Kong University of Science and Technology. He holds the Cheung Kong chair professor in Shanghai Jiao Tong University. Prior to that, he was with IBM Networking System Division, Research Triangle Park, North Carolina. He was an adjunct researcher with Microsoft Research Asia-MSRA and was a visiting scientist in Microsoft Advanced Technology Center (ATC). He has been a technical advisor for China Cache Corp. (NASDAQ CCIH) since 2007. He is an adjunct professor with the Huazhong University of Science and Technology, Wuhan, China. His recent research interests include: large-scale content distribution in the Internet, Peer-to-Peer media streaming, the Internet topology, cloud computing, green computing and communications. He is a fellow of the IEEE for "contribution to content distributions via the Internet". He received the Young Investigator Award from the National Natural Science Foundation of China (NSFC) in 2004. He served as a Distinguished lecturer of the IEEE Communications Society (2006-2007). He was a corecipient for three Best Paper Awards from IEEE, and the Best System Track Paper in ACM Multimedia (2009).