Convolutional Neural Network Training with Distributed K-FAC

J. Gregory Pauloski[‡], Zhao Zhang[*], Lei Huang[*], Weijia Xu[*], Ian T. Foster[¶]
[*]Texas Advanced Computing Center
Email: zzhang, huang, xwj@tacc.utexas.edu
[‡]University of Texas at Austin
Email: jgpauloski@utexas.edu
[¶]University of Chicago & Argonne National Laboratory
Email: foster@uchicago.edu

***Abstract*—Training neural networks with many processors can reduce time-to-solution; however, it is challenging to maintain convergence and efficiency at large scales. The Kronecker-factored Approximate Curvature (K-FAC) was recently proposed as an approximation of the Fisher Information Matrix that can be used in natural gradient optimizers. We investigate here a scalable K-FAC design and its applicability in convolutional neural network (CNN) training at scale. We study optimization techniques such as layer-wise distribution strategies, inverse-free second-order gradient evaluation, and dynamic K-FAC update decoupling to reduce training time while preserving convergence. We use residual neural networks (ResNet) applied to the CIFAR-10 and ImageNet-1k datasets to evaluate the correctness and scalability of our K-FAC gradient preconditioner. With ResNet-50 on the ImageNet-1k dataset, our distributed K-FAC implementation converges to the 75.9% MLPerf baseline in 18–25% less time than does the classic stochastic gradient descent (SGD) optimizer across scales on a GPU cluster.**

***Index Terms*—optimization methods, neural networks, scalability, high performance computing**

I. Introduction

Deep learning (DL) methods are having transformative impacts on many areas of society, not least in science and engineering where they are enabling exciting new approaches to problems in many disciplines. Convolutional neural networks (CNNs) are widely used for classification, regression, object detection, segmentation, and other tasks. As the powerful memory and communication architectures of high-performance computing (HPC) systems have been shown to support DL applications well, there is growing interest in both the use of HPC for DL [1–6] and the use of DL for scientific applications [7–9].

Leveraging the massive computing resources of supercomputers to dramatically reduce training time is challenging, especially under the constraint of convergence (e.g., validation accuracy) [10]. To this end, researchers have examined the scaling properties of first-order methods such as stochastic gradient descent (SGD) [1, 3–5, 11, 12] and shown promising scaling results for ResNet-50 [13] and BERT [14] applications, albeit via the use of application- or architecture-specific techniques such as distributed batch normalization and communication optimization. A second direction is to explore second-order information such as the Fisher Information Matrix (FIM) to reduce the required number of training iterations. Scientists have also examined the natural gradient method (a category of second-order methods) with Kronecker-factored Approximate Curvature (K-FAC) [6, 15, 16]. Although previous K-FAC research has shown significant reductions in training iterations and high scalability (up to 1024 Nvidia GPUs), the K-FAC implementation used in these works cannot converge to the acceptance performance of MLPerf [17], e.g., 75.9% validation accuracy for ResNet-50, or compete with the faster training times of SGD.

Equations 1 and 2 show the update rules for SGD and the iterative second-order method used in K-FAC, respectively. $w^{(k)}$ is the weight at iteration k, $\alpha^{(k)}$ is the learning rate at iteration k, n is the mini-batch size, $\nabla L_i(w^{(k)})$ is the gradient of the loss function L_i for the i^{th} example with regard to $w^{(k)}$, and $F^{-1}(w^{(k)})$ is the inverse of the Fisher information matrix (FIM).

$$\text{SGD: } w^{(k+1)} = w^{(k)} - \frac{\alpha^{(k)}}{n}\sum_{i=1}^{n}\nabla L_i(w^{(k)}) \tag{1}$$

$$\text{K-FAC: } w^{(k+1)} = w^{(k)} - \frac{\alpha^{(k)}F^{-1}(w^{(k)})}{n}\sum_{i=1}^{n}\nabla L_i(w^{(k)}) \tag{2}$$

We report in this paper on methods for improving the correctness and scalability of K-FAC based optimizers. We introduce a distributed K-FAC optimization strategy that leverages the independence between layer inputs to make efficient use of a large number of processors. We study the impact of various explicit and implicit matrix inverse algorithms on training convergence and cost to evaluate $F^{-1}(w^{(k)})\sum_{i=1}^{n}\nabla L_i(w^{(k)})$. Approximating the FIM as a preconditioner to the gradient is expensive; thus, we exploit a conventional method in L-BFGS [18] that decouples the approximation from variable update. This approach allows us to determine the best preconditioner update frequency to reduce training time while preserving convergence.

Specifically, we design the distribution strategies and study their scaling properties. The core idea is to distribute per-layer K-FAC calculations to individual processors, then aggregate the results as a preconditioner F^{-1} to the gradient $\nabla L(w^{(k)})$ for the iterative second-order method as shown in Equation 2. We examine two methods of computing the inverse of F, explicit inverse and implicit eigen decomposition, and select the latter algorithm as it preserves training convergence at scale. We also integrate a set of techniques including dynamic K-FAC update decoupling, damping decay, and K-FAC update frequency decay to reduce training time while preserving model convergence. Details are presented in §IV and §V.

We prototype our solution in the widely adopted PyTorch DL framework [19] and Horovod distributed training framework [20]. We use ResNet-34, ResNet-50, ResNet-101, and ResNet-152 as example applications and train on various numbers of Nvidia V100 GPUs on the TACC Frontera supercomputer. We show that with our new method, ResNet-50 on the ImageNet-1k [21] dataset converges to above or equal to the 75.9% baseline performance required by MLPerf [17] at all scales. Across scales, our K-FAC based optimization is 18–25% faster than SGD.

This work makes four contributions:

- A distributed K-FAC optimization strategy;
- A study of explicit/implicit matrix inverse algorithms on its impact to K-FAC and training convergence;
- An empirically optimal preconditioner update frequency for example applications;
- An open source implementation of the proposed algorithm using PyTorch [19] and Horovod [20].

The rest of this paper is organized as follows: §II discusses parallel DL training, SGD, and K-FAC. We review and summarize previous work in scalable DL training in §III. §IV introduces key system design options and decisions. §V discusses technical decisions in details. §VI presents the experiment results, analysis, and discussion. We draw conclusions in §VII.

II. Background

Neural network training is usually done through an iterative procedure. Most existing methods exploit the batch optimization method [11], where a mini-batch of training data is fed to neural network to derive an averaged loss and then used to update variables using corresponding rules. Equations 1 and 2 show the update rules of SGD and K-FAC.

In this section, we review distributed training strategies and explain in detail the applicability of the data parallel approach in supporting SGD and K-FAC.

A. Data Parallelism

There are three common strategies to distribute DL training across multiple processors: *data parallel*, *model parallel*, and *hybrid parallel*. The data parallel approach replicates the model across processors and distributes a mini-batch of training data to these processors in each iteration. The model parallel approach distributes a large model, usually exceeding the capacity of the host memory of a processor, to multiple processors. A hybrid approach mixes the two above approaches either by applying different parallel approaches on different layers, or by partitioning processors into groups,

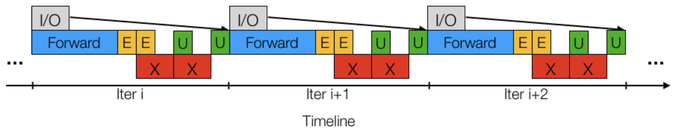

Fig. 1: An overview of the synchronous SGD iteration. I/O: Training programs read files. Forward: Forward computation. E: Gradient evaluation. X: Gradient exchange. U: Trainable variable update. Gradient evaluation and exchange are commonly referred together as back-propagation.

within which model parallelism is exploited while data parallelism is used across groups.

In practice, the data parallel approach is dominant in DL training at large scale, as it achieves better machine utilization compared to the model parallel method. Numerous effort such as Intel MLSL [22], Horovod [20], TensorFlow [23], and PyTorch [19] have native support for data parallel training via NCCL, Gloo [24], or MPI collective operations [25].

B. SGD

The implementation of SGD via data parallelism falls into two categories: synchronous [26] and asynchronous [27–31], depending on whether all variables are updated in every iteration. Asynchronous SGD methods have been proven to have a non-linear slowdown compared to synchronous SGD [32]. Thus we only consider synchronous methods in this paper.

Synchronous SGD is an iterative procedure of five steps: 1) I/O, 2) forward compute, 3) gradient evaluation, 4) gradient exchange, and 5) variable update. The distributed training program reads a fixed batch of training items from a storage system and may preprocess them. The training items are then fed to the neural network for forward computation to determine the defined loss. Given the loss, the gradient evaluation step computes the gradient for each trainable variable. All processors then communicate to exchange the gradients (e.g., to compute the average in SGD). Finally, the trainable variables are updated with the corresponding gradients using a specific rule, e.g., gradient descent. An iteration completes when all variables are updated. Most modern DL frameworks implement this procedure by using a streamlined parallelism with I/O and gradient exchange being performed asynchronously to the other steps, as shown in Figure 1. This approach allows for high utilization of available hardware such as CPUs, GPUs, and interconnects.

With SGD, the only data that is communicated are the initial model weights before training starts and the gradients in each iteration. Collective operations such as *broadcast()* and *allreduce()* suffice the requirements for SGD.

C. K-FAC

K-FAC is a method for efficiently approximating the natural gradient. In natural gradient descent optimization, the Fisher information matrix (FIM) is used to represent the curvature of the loss function. While computing the FIM is complex,

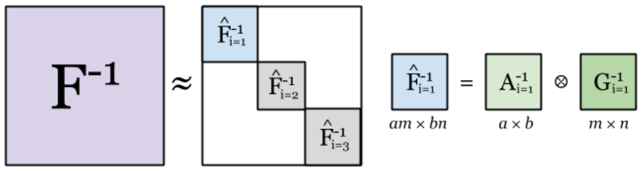

Fig. 2: The K-FAC approximation of the Fisher information matrix. $\otimes$ is the Kronecker product.

K-FAC approximates the FIM as Kronecker products of smaller matrices. These smaller matrices are more efficiently invertable.

The FIM can be interpreted as the negative expected Hessian of a log-likelihood and is given by

$$F = \mathbb{E}[\nabla \log p(y|x;\theta) \nabla \log p(y|x;\theta)^T]. \tag{3}$$

K-FAC approximates F as $\hat{F}$, a diagonal block matrix where each block represents one layer in a network of L layers:

$$\hat{F} = \mathtt{diag}(\hat{F}_1, ..., \hat{F}_i, ..., \hat{F}_L) \tag{4}$$

where

$$\hat{F}_i = a_{i-1}a_{i-1}^T \otimes g_i g_i^T = A_{i-1} \otimes G_i. \tag{5}$$

This is a Kronecker-factorization of $\hat{F}_i$ where the Kronecker factors, often referred to as the covariance matrices, are $A_{i-1} = a_{i-1}a_{i-1}^T$ and $G_i = g_i g_i^T$. The activation of the $i-1$th layer and the gradient of the output of the ith layer are represented as a_{i-1} and g_i respectively.

The Kronecker product $A \otimes B$ where A has size $m \times n$ and B has size $p \times q$ is:

$$A \otimes B = \begin{bmatrix} a_{11}B & \dots & a_{1n}B \\ \vdots & \ddots & \vdots \\ a_{m1}B & \dots & a_{mn}B \end{bmatrix}. \tag{6}$$

For example:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \otimes \begin{bmatrix} 5 & 6 \\ 7 & 8 \\ 9 & 0 \end{bmatrix} = \begin{bmatrix} 1\times5 & 1\times6 & 2\times5 & 2\times6 \\ 1\times7 & 1\times8 & 2\times7 & 2\times8 \\ 1\times9 & 1\times0 & 2\times9 & 2\times0 \\ 3\times5 & 3\times6 & 4\times5 & 4\times6 \\ 3\times7 & 3\times8 & 4\times7 & 4\times8 \\ 3\times9 & 3\times0 & 4\times9 & 4\times0 \end{bmatrix} \tag{7}$$

The inverse of $\hat{F}_i$ can be computed as the Kronecker product of the inverse of the factors A_{i-1} and G_i, as shown in Figure 2, using the property $(A \otimes B)^{-1} = A^{-1} \otimes B^{-1}$.

$$\hat{F}_i^{-1} = A_{i-1}^{-1} \otimes G_i^{-1} \tag{8}$$

We can then use $\hat{F}_i^{-1}$ to precondition the gradient, ∇L, of the parameters $w_i^{(k)}$ for layer i and iteration k.

$$w_i^{(k+1)} = w_i^{(k)} - \alpha^{(k)} \hat{F}_i^{-1} \nabla L_i(w_i^{(k)}) \tag{9}$$

Using the relation $(A \otimes B)\vec{c} = B^T \vec{c} A$, the preconditioned gradient $\hat{F}_i^{-1} \nabla L_i(w_i^{(k)})$ can be efficiently computed as

$$\hat{F}_i^{-1} \nabla L_i(w_i^{(k)}) = G_i^{-1} \nabla L_i(w_i^{(k)}) A_{i-1}^{-1}. \tag{10}$$

The FIM approximation $\hat{F}_i$ can be ill-conditioned for inverting so to compensate for inherent inaccuracies, a Tikhonov regularization technique is applied where γI is added to $\hat{F}_i$ [6, 33]. We call γ the damping parameter, and γI can be added to the Kronecker factors such that instead of computing $\hat{F}_i^{-1}$, we compute $(\hat{F}_i + \gamma I)^{-1}$ as:

$$(\hat{F}_i + \gamma I)^{-1} = (A_{i-1} + \gamma I)^{-1} \otimes (G_i + \gamma I)^{-1}. \tag{11}$$

Thus, the final update step for the parameters at iteration k is:

$$w_i^{(k+1)} = w_i^{(k)} - \alpha^{(k)} (G_i + \gamma I)^{-1} \nabla L_i(w_i^{(k)}) (A_{i-1} + \gamma I)^{-1} \tag{12}$$

D. Frameworks

We choose PyTorch [19] and Horovod [20] to prototype the K-FAC optimizer. PyTorch is an widely adopted DL framework with comprehensive support of various neural network architectures and high performance. At its backend, PyTorch uses C++ runtime for performance. At its frontend, PyTorch exposes a linear algebra interface and a neural network interface, which can be used to implement certain matrix operations and compose neural networks, respectively. PyTorch has an imperative programming interface which eases our development compared to the symbolic programming interface of TensorFlow [23]. PyTorch has its own distributed training support with collective communication in MPI, NCCL, or Gloo [24]. We choose Horovod [20] instead, as Horovod supports PyTorch, TensorFlow, and MXNet [34] and we plan to extend our K-FAC optimization to other major DL frameworks.

Horovod is a general communication framework that can call MPI, NCCL, or IBM Distributed Deep Learning Library (DDL) primitives. It inherits MPI concepts such as *size*, *rank*, and *local rank*, and exposes a limited communication interface with *allreduce()*, *allgather()*, and *broadcast()* primitives. In particular, its *allreduce()* operation is implemented by using the scatter-reduce algorithm, which is bandwidth optimal in the ring topology under the assumption that DL gradient exchange involves large enough data to be bandwidth bound [35]. *Allreduce* in Horovod can be synchronous or asynchronous; users can specify the size of fusion buffer that accumulates data before communication until reaching the buffer size. The fusion buffer is usually set as 16 MB or 32 MB to guarantee that each *allreduce()* is bandwidth dominated. Thus the communication in Horovod can make highly efficient use of the underlying interconnect.

III. Related Work

Much recent research has focused on scaling DL training via the use of large batch sizes. For example, with the classic image classification problem, a batch size of 32K is considered large for convolutional neural network training with the ImageNet-1k dataset. Large batches make it possible to distribute enough data to a large number of processors to maintain high utilization. However, they lead to high communication costs at large scale, as the size of the gradients are large enough that the communication overhead is dominated by bandwidth and is proportional to the number of processors.

In this section, we summarize previous work of DL training at large scale and justify the fundamental difference between our work and previous efforts.

A. Scaling results of SGD

Previous research such as LARS [1] and LAMB [12] propose SGD variants with layer-wise adaptive learning rate and learning rate schedules to enable large batch size training without losing model convergence. The ResNet-50 training with ImageNet-1k dataset is shown to converge to 74.9% baseline at the time in 20 minutes on 2048 Intel Xeon Platinum 8160 processors [1]. Subsequent work [3–5] leverages the same optimizer with dedicated optimizations for the interconnect architecture, e.g., 2D torus, and processor architecture optimizations such as GPU and TPU. Researchers were able to bring the training time to ∼2.2 minutes on 1024 TPUs [4], with a model that converges to 76.3% validation accuracy, which is above the MLPerf [17] baseline of 75.9%. Some techniques such as mixed precision, interconnect-aware *allreduce()*, and distributed batch normalization are specific to certain hardware and applications. In contrast, our work focuses on general optimizations that can be applied regardless of processor architecture, interconnect topology, or application.

B. Scaling results of K-FAC

K-FAC [15] has been shown to take fewer steps to converge for image classification [6, 36] and language processing tasks [37]. However, each step in K-FAC runs significantly slower than in SGD, as the FIM must be approximated. In previous work, an asynchronous distributed K-FAC using a doubly-factored Kronecker approximation was used to achieve a time-per-iteration comparable to standard SGD training on ImageNet-1k [36] . While this work reports a $2\times$ improvement in overall training times due to faster convergence at the start of training, the implementation only converges to a final 70% validation accuracy. Researchers have also published a distributed implementation of K-FAC that distributes a block-diagonal approximation (with each block associated to one layer) across GPUs [6]. Even though the work reports that their K-FAC implementation takes ∼10 minutes to reach the 74.9% validation accuracy baseline in just 978 iterations, there is no training time comparison of K-FAC to SGD. In addition, the model convergence is obviously 1% lower than the MLPerf [17] baseline, which is considered as an acceptance test for DL researchers and practitioners. In our work, we prioritize training correctness and make such system design decisions when facing performance tradeoffs. Further, we compare K-FAC training time to SGD and design strategies to reduce training time without losing model convergence.

TABLE I: CIFAR-10 ResNet-32 validation accuracy for inverse vs. eigen decomposition K-FAC updates

Batch Size	256	512	1024
SGD	92.77%	92.58%	92.69%
K-FAC w/ Inverse	92.58%	92.36%	91.71%
K-FAC w/ Eigen-decomp.	92.76%	92.90%	92.92%

IV. Design

Recent works have found promising results in reducing the overhead of K-FAC by assigning each worker, e.g., GPU, to compute the FIM approximation for a single layer such that the K-FAC update for each layer can be computed in parallel [6]. While this method is effective when the number of layers in the network is equal to or greater than the number of workers, the scaling performance decreases as workers are left idle when there are more workers than layers to compute. Our distributed K-FAC design seeks to improve on this method by reducing the frequency of communication, increase the granularity at which the K-FAC computations can be distributed, and ensuring our design achieves MLPerf baselines on benchmarks. We also design our K-FAC algorithm to act as a gradient preconditioner such that K-FAC can be used in-place with any standard optimizer, such as Adam, LARS, or SGD.

A. Matrix Inversion

A standard K-FAC update step for one layer requires inverting two matrices, $(A_{i-1} + \gamma I)$ and $(G_i + \gamma I)$, as shown in Equation (11). However, it has been shown that $(\hat{F}_i + \gamma I)^{-1}$ can be computed implicitly using an alternative method based on the eigendecompostion of $\hat{F}_i$ [33]. Let $A_{i-1} = Q_G \Lambda_G Q_G$ and $G_i = Q_A \Lambda_A Q_A$ be the eigen decompositions of the factors A_{i-1} and G_i. Then, we can compute the preconditioned gradient as:

$$V_1 = Q_G^T L_i(w_i^{(k)}) Q_A \tag{13}$$

$$V_2 = V_1 / (v_G v_A^T + \lambda) \tag{14}$$

$$(\hat{F}_i + \gamma I)^{-1} \nabla L_i(w_i^{(k)}) = Q_G V_2 Q_A^T \tag{15}$$

where v_A and v_G are vectors of the eigenvalues of A_{i-1} and G_i, i.e. the diagonals of Λ_A and Λ_G. For the full proof of this eigen decomposition expansion, see Appendix A.2 in [33].

In our design, we use the eigen decomposition expansion of $(\hat{F}_i + \gamma I)^{-1}$ in Equations (13)–(15) to compute the preconditioned gradient. In Table I, we compare the final validation accuracy on CIFAR-10 with ResNet-32 between the explicit inverse method in Equation (11) and the implicit eigen decomposition method. We consider the baseline for acceptable accuracy to be 92.49% [13]. As the batch size increases, the inverse K-FAC update performs worse on the validation data and drops below the acceptable baseline whereas both SGD and K-FAC with eigen decomposition perform above baseline performance at all batch sizes.

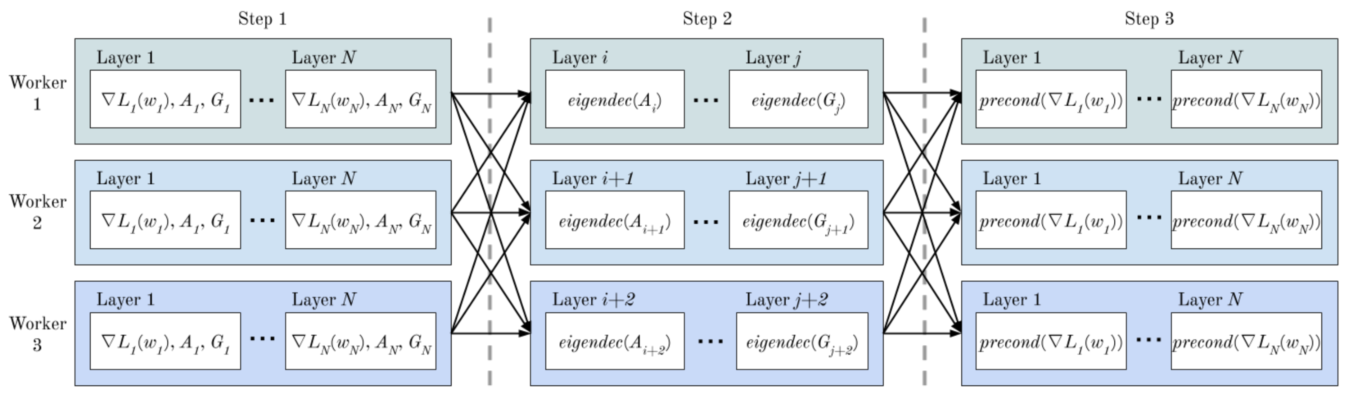

Fig. 3: Overview of our layer-wise factor distribution scheme. In step 1, each worker computes the gradient, $\nabla L_i(w_i)$, and the factors, A_{i-1} and G_i, for each layer using the worker's mini-batch. Before step 2, the gradients and factors are *allreduced*, and each factor is assigned to a worker in a round-robin fashion. In step 2, each worker computes the eigen decomposition of each factor it was assigned. The results from step 2 are gathered across all workers before step 3 where the preconditioned gradient is computed locally and in-place for all layers following Equations (13) to (15). In step 2, we use indices i for A and j for G to denote that the eigen decomposition for A_{i-1} and G_i can occur on different workers. The pseudocode for each step is provided in Algorithm 1.

Algorithm 1: Distributed K-FAC preconditioner

```
   /* Compute Gradients */
1  foreach worker do
2      Compute forward and backward pass
3  end
   /* Step 1: Compute Factors */
4  allreduce($\nabla L_{1:L}(w_{1:L})$)
5  foreach worker do
6      Compute $A_{0:L-1}$ and $G_{1:L}$
7  end
8  allreduce($A_{0:L-1}$,$G_{1:L}$)
9  Assign factors $A_{0:L-1}$ and $G_{1:L}$ to unique workers
   /* Step 2: Compute Decompositions */
10 foreach worker $w$ do
11     foreach $A_i$ assigned to $w$ do
12         $Q_{A_i}, \Lambda_{A_i} = eigendecompose(A_i)$
13     end
14     foreach $G_j$ assigned to $w$ do
15         $Q_{G_j}, \Lambda_{G_j} = eigendecompose(G_j)$
16     end
17 end
18 allgather($Q_{A_{0:L-1}}, \Lambda_{A_{0:L-1}}, Q_{G_{1:L}}, \Lambda_{G_{1:L}}$)
   /* Step 3: Precondition Gradient */
19 foreach worker do
20     precondition($\nabla L_{1:L}(w_{1:L})$)
21 end
   /* Update Weights with SGD */
22 foreach worker do
23     Update weights using the preconditioned gradients
24 end
```

B. Parallelism

During each iteration, the forward and backward passes are computed on each worker using the worker's local mini-batch. Hooks are registered to the input and output of each layer to save the activation of the previous layer and gradient with respect to the output of the current layer. Then using Horovod's *allreduce()*, the gradients computed in the backward pass are averaged across all workers.

Each worker computes the Kronecker factors A_{i-1} and G_i using the saved activations and gradients of the output for each layer, and we maintain a running average of the Kronecker factors computed from each mini-batch.

$$A_{i-1}^{(k)} = \xi A_{i-1}^{(k)} + (1-\xi)A_{i-1}^{(k-1)} \tag{16}$$

$$G_i^{(k)} = \xi G_i^{(k)} + (1-\xi)G_i^{(k-1)} \tag{17}$$

ξ is the running average hyper-parameter typically in the range $[0.9, 1)$. Using *allreduce()*, the updated running averages of the factors are averaged across workers. This process of calculating the gradients and factors locally and averaging the results is shown in step 1 of Algorithm 1 and Figure 3.

Current distributed K-FAC implementations [6] choose to assign each worker to compute A_{i-1}^{-1}, G_i^{-1}, and the final preconditioned gradient $(\hat{F}_i + \gamma I)^{-1}\nabla L_i(w_i^{(k)})$ before communicating the per layer preconditioned gradients to all workers such that each worker can update its local weights. Our design takes a different approach where we assign each worker a single factor to eigen decompose in a round-robin fashion. This is shown in step 2 of Algorithm 1 and Figure 3 where each worker computes the eigen decomposition of the subset of factors A_{i-1} and G_i assigned to itself. This approach decouples the eigen decomposition updates from the preconditioning of the gradients.

The eigen decompositions for the factors are then communicated between all workers so that each worker can compute the preconditioned gradients locally using Equations (13)–(15). The preconditioned gradients are computed in place so that any standard optimizer, e.g., SGD, can be used to update the weights.

C. Communication

Our design introduces communication in three places: 1) averaging the gradients, 2) averaging the Kronecker factors, and 3) gathering the eigen decompositions for each factor. By partitioning the communication into these three parts, we can take advantage of the fact that preconditioned gradients are computed locally to reduce communication in iterations where the factors are not updated.

A common strategy to improve training times when using K-FAC is to only update the Kronecker factors every n iterations. In iterations where the Kronecker factors are not updated, we can skip the communication for (2) and (3) and use the stale Kronecker factors from previous iterations that are already stored locally on each worker to compute the preconditioned gradients. This reduction in communication is possible because we decouple the eigen decomposition of the factors from the preconditioning of the gradients. In Algorithm 1, this would result in skipping lines 5–18. As training progresses, the FIM becomes more stable and impact of using stale eigen decompositions decreases.

In practice, the factors can be updated every tens or hundreds of iterations without loss in performance. Since the K-FAC updates happen infrequently, the majority of training iterations require no extra communication compared to conventional model parallel training with SGD which only requires communicating the gradients.

By decoupling the Kronecker factor calculations from the preconditioned gradient calculation, we are also able to compute the eigen decompositions for A_{i-1} and G_i on different workers and achieve double the worker utilization compared to existing distributed K-FAC implementations that use a layer-wise distribution scheme [6].

V. Implementation

Our implementation is based on an existing open-source K-FAC optimizer written with PyTorch that has no support for distributed training [38, 39]. We modified this implementation to support our distributed K-FAC design and to act as a preconditioner for standard PyTorch optimizers. Our implementation supports K-FAC updates for Linear and Conv2D layers. All unsupported layers are ignored by the K-FAC preconditioner and updated normally using the user's choice of optimizer, such as SGD. Careful consideration was made to ensure that our K-FAC implementation could be used with minimal changes to existing PyTorch scripts that use Horovod for distributed training.

A. Communication with Horovod

Existing open-source K-FAC implementations that support distributed training use TensorFlow's parameter server model

```
1  ...
2
3  optimizer = optim.SGD(model.parameters(), ...)
4  optimizer = hvd.DistributedOptimizer(optimizer, ...)
5  preconditioner = KFAC(model, ...)
6
7  ...
8
9  for i, (data, target) in enumerate(train_loader):
10   optimizer.zero_grad()
11   output = model(data)
12   loss = criterion(output, target)
13   loss.backward()
14
15   optimizer.synchronize()
16   preconditioner.step()
17   with optimizer.skip_synchronize():
18     optimizer.step()
19
20   ...
```

Listing 1: Example K-FAC usage.

[40]. Parameter servers introduce a bottleneck that inhibits performance at large scale. For this reason, we use Horovod for communication because Horovod uses a decentralized approach that scales well to thousands of compute nodes [20].

Horovod's PyTorch implementation offers support for synchronous and asynchronous communication operations. All communications operations we used are done asynchronously to take advantage of parallelism between computation and communication. Using Horovod, handles are registered to communication operations such that we can register *allreduce()* operations as we compute factors and eigen decompositions across workers and wait to do the communication in batches.

B. K-FAC Interface

Our K-FAC implementation is designed to be easily inserted into existing training scripts using Horovod. An example of how to incorporate K-FAC into existing training scripts is given in Listing 1. The only necessary changes are to initialize the *KFAC()* preconditioner and add a call to *KFAC.step()* before *optimizer.step()* as seen in lines 5 and 16 respectively. By default when using Horovod's *DistributedOptimizer()*, Horovod waits to call *allreduce()* on the gradients until *optimizer.step()* is called. However, the gradients must be averaged across workers before we can call *KFAC.step()*, so we call *optimizer.synchronize()*, shown on line 15, before performing the K-FAC preconditioning.

C. K-FAC Hyper-Parameters

Our K-FAC preconditioner introduces a number of hyper-parameters for controlling gradient clipping, factor update frequency, and damping.

After preconditioning the gradients, we scale the gradients by a factor ν where

$$\nu = \min\left(1, \sqrt{\frac{\kappa}{\alpha^2 \sum_{i=1}^{n} |\mathcal{G}_i^T \nabla L_i(w_i)|}}\right) \tag{18}$$

where κ is a user-defined constant, typically on the order of 10^{-3}, α is the learning rate, and $\mathcal{G}_i$ is the preconditioned gradient [38, 39]. This gradient scaling is done to prevent the norm of $\mathcal{G}_i$ becoming large compared to w_i [6].

Similar to the work [6], we also use a damping decay scheme whereby we reduce the damping by a fixed scalar quantity at fixed epochs. Starting with a larger damping accounts for rapid changes in the FIM at the start of training.

As training progresses and the FIM becomes more stable, the frequency at which the Kronecker factors and eigen decompositions needs to be updated decreases. We use the hyper-parameter *kfac-update-freq* to control the frequency at which we update the eigen decompositions of A_{i-1} and G_i. We find that the factors A_{i-1} and G_i can be updated and communicated at a frequency of $10\times$ *kfac-update-freq* without loss in performance.

At fixed training epochs, we decrease *kfac-update-freq* by a scalar quantity to reduce the computation and communication required while preserving accuracy. In practice, we found that it was sufficient to maintain a constant *kfac-update-freq* for the entirety of training, however, small performance improvements can be gained by fine-tuning the update frequency schedule.

VI. Experiments

Now we present the empirical results of the scalable K-FAC preconditioner. In this section, we will introduce the hardware and software platforms, applications, and datasets used in our experiments. We adopt the MLPerf [17] acceptance performance as baseline. We study the correctness, performance, and scalability of our K-FAC preconditioner along with comparisons to SGD.

A. Platforms

We use the GPU subsystem of the Frontera supercomputer hosted at Texas Advanced Computing Center (TACC). This machine is powered by IBM Power9 processors and has 112 nodes in total. There are 4 Nvidia V100 GPUs, 256 GB RAM, and 1 TB rotational disk per node. The nodes are connected by an InfiniBand EDR network.

We prototype the K-FAC preconditioner using PyTorch v1.1 and Horovod v0.19.0. These software frameworks rely on CUDA 10.0, CUDNN 7.6.4, and NCCL 2.4.7. We use single-precision floating point numbers (FP32) for training and communication. Our code is open source with the MIT license and is hosted at https://github.com/gpauloski/kfac_pytorch.

B. Datasets and Applications

Throughout the development process, we use ResNet-34 [13] with the CIFAR-10 [41] dataset to test correctness. Then we use the ImageNet-1k dataset [21] with ResNet-50, ResNet-101, and ResNet-152 to empirically evaluate the performance of K-FAC as a preconditioner to SGD.

The CIFAR-10 dataset has 10 categories with a total of 50 000 training images and 10 000 validation images. The ImageNet-1k dataset has 1000 categories with ~1.3 M training images and 50 000 validation images.

C. Results

1) Correctness: We first use CIFAR-10 and ResNet-34 to confirm the correctness of our K-FAC implementation. We adopt the baseline for acceptable accuracy to be 92.49% [13]. We train with K-FAC for 100 epochs and SGD for 200 epochs. For both optimization methods, we specify the learning rate as $N \times 0.1$ and the batch size as $N \times 128$ where N is the number of GPUs. The learning rate is decreased by a factor of 10 at epochs 35, 75, 90 for K-FAC and 100, 150 for SGD, and a linear learning rate warmup is used for the first five epochs. We maintain a constant K-FAC update frequency of 10 iterations. Figure 4 shows the validation accuracy for SGD and K-FAC on one and two GPUs. Table II provides the final validation accuracies for SGD and K-FAC on 1, 2, 4, and 8 GPUs. We find that our K-FAC implementation performs as well or better than SGD across a range of batch sizes, while converging in fewer iterations.

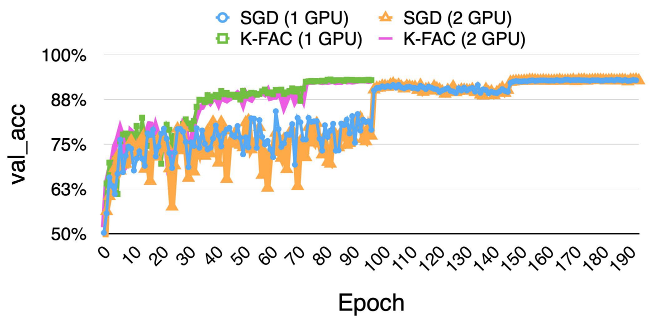

Fig. 4: Validation accuracy comparison of ResNet-32 on CIFAR-10 with KFAC and SGD.

TABLE II: Validation accuracy comparison of ResNet-32 on CIFAR-10 with KFAC and SGD.

GPUs	1	2	4	8
SGD	92.76%	92.77%	92.58%	92.69%
K-FAC	92.93%	92.76%	92.90%	92.92%

We train ResNet-50 on the ImageNet-1k dataset with K-FAC for 55 epochs and SGD for 90 epochs to confirm that K-FAC: (1) converges to the MLPerf 75.9% validation accuracy baseline, (2) converges to an equal or higher validation accuracy than that achieved by SGD, and (3) converges in fewer iterations than SGD. In this experiment, we use the batch size of 32×16=512 and the learning rate of $0.0125\times16 = 0.2$, using a decay schedule at epoch 25, 35, 40, 45, and 50. We specify damping value as 0.001 and evaluate K-FAC approximation every 10 iterations. Labels are smoothed with a factor of 0.1. For SGD, we set the momentum to 0.9. For each test case, we use linear learning rate warmup for the first five epochs.

Figure 5 shows the Top-1 validation accuracy curves of ResNet-50 on the ImageNet-1k dataset on 16 GPUs. K-FAC as a preconditioner for SGD meets all three criteria by converging to 76.4% validation accuracy, outperforming SGD with no preconditioning by 0.2%, and consistently converging to the MLPerf baseline in under 55 epochs compared to the standard 90 epochs required by SGD. In this case, K-FAC reaches 75.9% in the 43rd epoch, compared to 76th epoch of SGD.

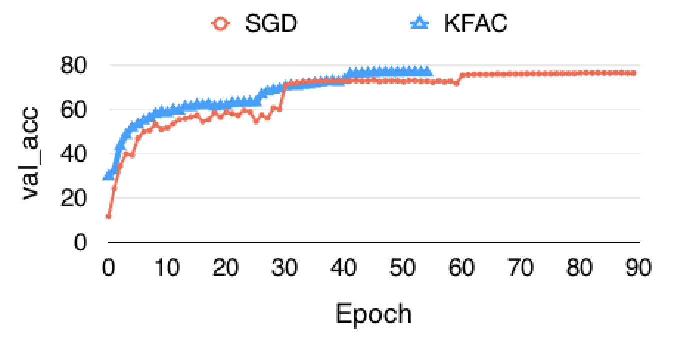

Fig. 5: Validation accuracy comparison of ResNet-50 on ImageNet-1k with KFAC and SGD on 16 GPUs. K-FAC converges to 76.4% while SGD converges to 76.2%.

2) Performance: Reducing the frequency of computing the factors A_{i-1} and G_i and their eigen decompositions is key to achieving high-performance when training with K-FAC. By reducing the K-FAC update frequency, we can avoid significant computation and communication at the cost of introducing stale information. Understanding the tradeoff between staleness of information and improved time-to-solution is key when tuning the update frequency.

Table III shows the Top-1 validation accuracy of ResNet-50, ResNet-101, and ResNet-152 trained with K-FAC for 55 epochs with a varying update frequency of {100, 500, 1000} iterations using 64 V100 GPUs. Figure 6 shows the Top-1 validation accuracy in the last 10 epochs of each test case. We find that ResNet-50 with all update frequencies except 1000 converge above the 75.9% baseline. This consistent performance is key to achieving fast and scalable performance when training with K-FAC. Observing that a larger interval than 500 iterations achieves marginal performance improvement, we choose the K-FAC update interval as 500 iterations with 64 GPUs. For the scalability experiments in §VI-C3, we scale the interval according to the total batch size, so that the K-FAC update frequency is constant per epoch. Specifically, we use 2000, 1000, 500, 250, 125-iteration K-FAC update intervals for all ResNet scaling experiments on 16, 32, 64, 128, 256-GPUs. For ResNet-101 and ResNet-152, we observe a 0.2% validation accuracy drop with K-FAC compared to SGD. Although there is no well-established baseline for ResNet-101 and ResNet-152, 76.4% and 76.6% are documented in Keras Applications [42], respectively. Both our SGD and K-FAC results are significantly higher than these numbers.

3) Scalability: We test the scalability of our distributed K-FAC algorithm by measuring the time-to-solution on {16, 32, 64, 128, 256} GPUs. For each experiment, we measure the average time per epoch over 10 epochs and project the completed time span to 55 epoch for K-FAC and 90 epochs for SGD. Assuming we have N GPUs, we specify the learning

TABLE III: ResNet-50, ResNet-101, and ResNet-152 validation accuracy vs. K-FAC update frequency with 64 GPUs.

			K-FAC Update Freq.		
Model		SGD	100	500	1000
ResNet-50	Val Accu	76.2%	76.2%	76.1%	75.5%
	Train. Time (min)	178	152	128	124
ResNet-101	Val Accu	78.0%	77.7%	77.7%	77.3%
	Train. Time (min)	244	227	197	195
ResNet-152	Val Accu	78.2%	78.0%	78.0%	77.8%
	Train. Time (min)	345	369	310	300

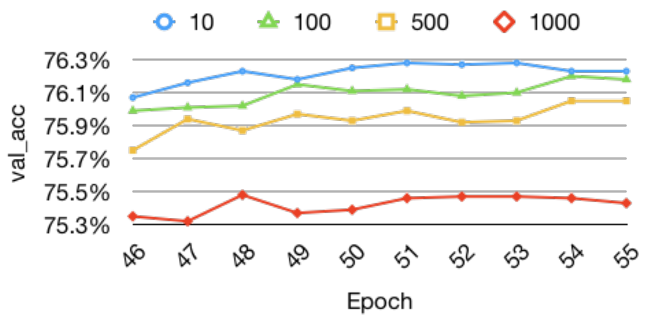

Fig. 6: ResNet-50 validation accuracy of the last 10 epochs with K-FAC update frequency of {10, 100, 500, 1000} iterations. The MLPerf baseline is 75.9%.

rate as $N\times 0.0125$ the batch size of $N\times 32$. We set the damping value to 0.001 and the same learning rate decay schedule of 25, 35, 40, 45, 50 for K-FAC and 30, 40, 80 for SGD. For both K-FAC and SGD, we use the linear learning rate warmup for the first five epochs. We set the momentum of SGD to 0.9. We maintain the same hyper-parameters across all runs including the SGD hyper-parameters when SGD is used on its own and with K-FAC as a preconditioner.

To understand the performance benefits of our distribution and communication scheme for K-FAC, we compare two variants of our K-FAC preconditioner, *K-FAC-lw* and *K-FAC-opt*. *K-FAC-lw* refers to the K-FAC optimizer with just layer-wise distribution strategy and *K-FAC-opt* represents the optimized distribution strategy. Both variants use the K-FAC update procedure outlined in Equations (13)–(15); the only differences between the two are how work is distributed among workers and where communication occurs. *K-FAC-lw* uses the layer-wise distribution scheme of [6] where each worker computes the entire K-FAC update, i.e. the eigen decompositions of the factors and the preconditioned gradient, for a single layer and communicates the final preconditioned gradient for that layer to all other workers. *K-FAC-opt* utilizes all of the optimizations introduced in this paper to reduce the frequency of communication by decoupling the eigen decomposition calculation from the preconditioning of the gradients. We verify that all *K-FAC-lw* and *K-FAC-opt* experiments converge to validation accuracy of 76.2%, 77.7%, and 78.0% for ResNet-50, ResNet-101, and ResNet-152, respectively.

Figure 7 shows the time-to-solution comparison between K-FAC and SGD across scales. At all scales, K-FAC as a preconditioner to SGD converges to the MLPerf baseline in

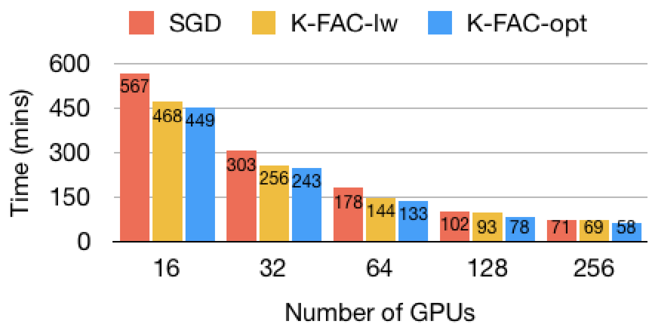

Fig. 7: Time-to-solution comparison of ResNet-50 using K-FAC with SGD.

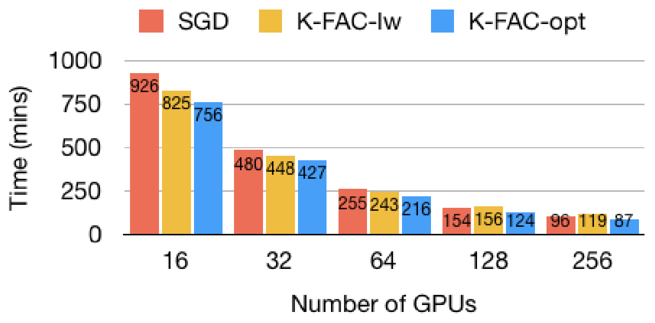

Fig. 8: Time-to-solution comparison of ResNet-101 using K-FAC and SGD.

55 epochs. *K-FAC-lw* outperforms SGD by 2.8-19.1%, and *K-FAC-opt* outperforms SGD by 17.7-25.2%. On 128 GPUs, the sustained scaling efficiency of *K-FAC-opt* is 71.8% which is a 9.4% improvement over the 62.4% efficiency of *K-FAC-lw*. It is also higher than the 68.6% scaling efficiency of SGD. The reduced communication frequency of *K-FAC-opt* results in better scaling. On 256 GPUs, the scaling efficiency of all three cases drop below 50%. However, *K-FAC-lw* achieves 2.8% improved performance than SGD whereas *K-FAC-opt* yields an 18.3% improvement at 256 GPUs.

4) Limitations: We compare the scaling of SGD and K-FAC across model sizes by training ResNet-50, ResNet-101, and ResNet-152 on the ImageNet-1k dataset. Figure 8 and Figure 9 show the time-to-solution comparison between K-FAC and SGD across scales. *K-FAC-opt* outperforms SGD by 9.7-19.5% on ResNet-101 at all scales and by 4.9-8.2% on ResNet-152 up to 128 GPUs. At 256 GPUs on ResNet-152, we find that *K-FAC-opt* is 11.1% slower than SGD.

Table IV shows the improvement of *K-FAC-opt* over SGD across the three models and scales. We observe that *K-FAC-opt*'s relative performance to SGD deteriorates with model complexity and scale. To explain the observed scaling trend, we use the model in Figure 1 in §II-B. The terms $T_{i/o}$, T_f, T_e, T_x, and T_u refer to the time cost of each of the five steps performed in each iteration: 1) I/O, 2) forward compute, 3) gradient evaluation, 4) gradient exchange, and 5) variable update. To ease the discussion, we assume that these steps are

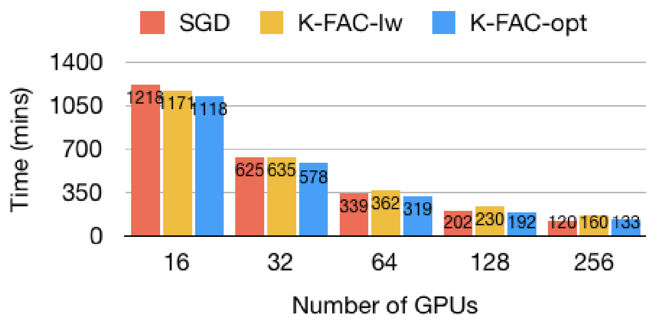

Fig. 9: Time-to-solution comparison of ResNet-152 using K-FAC and SGD.

TABLE IV: Summary of *K-FAC-opt* improvement over SGD

Scale	16	32	64	128	256
ResNet-50	20.9%	19.7%	25.2%	23.5%	17.7%
ResNet-101	18.4%	11.1%	15.1%	19.5%	9.7%
ResNet-152	8.2%	7.6%	6.0%	4.9%	-11.1%

performed in sequential order without any pipeline parallelism.

The performance deterioration with increasing model complexity is attributed to the super-linear increase in T_e: as the model is more complex, the training time with SGD increases proportionally to the parameter count. With K-FAC, $T_{i/o}$, T_f, T_x, and T_u consume identical time as SGD. On the other hand, T_e in K-FAC includes two additional stages compared to SGD: factor computation and eigen decomposition. We profile the factor computation cost per step on 16 V100 GPUs and observe a super-linear increase in time spent computing the factors A_{i-1} and G_i as model complexity increases, as shown in Figure 10. Unlike the eigen decomposition stage which can take advantage of the layer-wise distribution scheme, the factor computation time is constant as the GPU count is increased, as show in Table V, which leads to the super-linear increase in T_e and in turn to the relative performance deterioration of K-FAC relative to SGD.

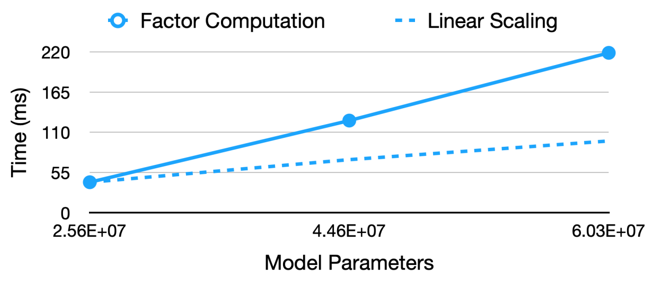

Fig. 10: Factor computation time as model complexity increases.

To understand the performance deterioration at increasing scale, we examine the time spent computing the eigen decompositions. The time required to complete the eigen decomposition stage is bounded by the slowest worker. Ideally,

TABLE V: Time (ms) profile for the factor computation and eigen decomposition of a K-FAC update step across various model sizes and scales. T_{comm} is the communication time, and T_{comp} is the computation time.

		Factor		Eigen Decomposition	
Model	GPUs	T_{comp}	T_{comm}	T_{comp}	T_{comm}
ResNet-50	16	36.83	155.79	2256.64	117.28
	32	43.30	171.57	1668.19	149.60
	64	44.90	154.63	1497.96	142.93
ResNet-101	16	125.23	224.15	3271.72	199.69
	32	126.14	267.08	2280.38	265.57
	64	126.95	239.33	2410.24	253.23
ResNet-152	16	218.36	276.83	4067.69	279.08
	32	219.00	313.17	2758.42	329.05
	64	219.12	312.52	2212.24	347.99

TABLE VI: Minimum and maximum eigen decomposition worker speedup.

	ResNet-50		ResNet-101		ResNet-152	
GPUs	min	max	min	max	min	max
16	1.00	1.00	1.00	1.00	1.00	1.00
32	1.34	2.88	1.41	3.33	1.51	2.03
64	1.55	6.61	1.26	6.18	1.85	8.27

as GPU count is doubled, the eigen decomposition time is halved because each GPU has approximately half the number of factors to decompose. However, in ResNet models, the individual factors can vary widely in size, and thus while each worker may have roughly the same number of factors to decompose, there can be an imbalance in aggregate sizes.

We can understand the factor size imbalance across workers by computing the total number of parameters assigned to each worker in ResNet-50. On 16 GPUs, the minimum number of parameters assigned to a worker is 1.46×10^6 and the maximum is 2.83×10^7. In comparison, on 64 GPUs, the minimum parameter count is 1.64×10^4 and the maximum is 2.26×10^7. It is clear that while some workers see a dramatic decrease in the number of parameters assigned, and thus a decrease in time spent decomposing the factors, some workers still have a similar amount of work, even though the number of workers was quadrupled.

To further quantify this trend, we measure the time for each worker to eigen decompose its assigned factors for ResNet-50, 101, and 152 on 16, 32, and 64 GPUs. We record the times of the fastest and slowest worker in each iteration such that we can observe the relative decrease in eigen decomposition time for the fastest and slowest workers as the worker count is increased. The minimum and maximum worker speedups are reported in Table VI. Across all three models, the fastest workers saw a 6.18-8.27x speedup in time when moving from 16 to 64 GPUs; however, the slowest workers saw minimal speedups of 1.26-1.85x. The imbalance in the work assigned to each GPU is exacerbated at scale as workers are left idle as they wait for the slower workers to finish the computation.

Factors are distributed in a greedy, round-robin fashion which contributes to the imbalance in work assigned. To resolve this scaling bottleneck, one direction is to implement a placement policy that uses factor size as a heuristic for the eigen decomposition time such that factors can be assigned to workers in a way that balances the time spent in this stage across workers.

VII. Conclusion and Future Work

We have presented a distributed K-FAC optimizer that is correct, efficient, and scalable. The optimizer integrates techniques of layer-wise distribution, inverse-free second-order gradient evaluation, K-FAC approximation decoupling, and dynamic K-FAC update frequency. We prototype the design using the widely adopted PyTorch and Horovod framework and release the source code under the permissive MIT license. We empirically evaluate the correctness and scalability of the proposed optimizer using the classic ResNet model family with the CIFAR-10 and ImageNet-1k datasets. Our results show that our optimizer converges to the 75.9% MLPerf ResNet-50 baseline with 18–25% less time than SGD.

In the future, we will explore alternative K-FAC approximation strategy that is more scalable than the current design. We will also design and evaluate solutions to avoid communications and reduce communication quantity to further enhance the scalability of K-FAC.

Acknowledgments

This work was supported in part by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, Contract DE-AC02-06CH11357, the Exascale Computing Project, Project Number 17-SC-20-SC, and by NSF OAC-1931354 and OAC-1818253.

References

[1] Y. You, Z. Zhang, C.-J. Hsieh, J. Demmel, and K. Keutzer, "ImageNet training in minutes," in *47th International Conference on Parallel Processing*. ACM, 2018, p. 1.

[2] V. Codreanu, D. Podareanu, and V. Saletore, "Scale out for large minibatch SGD: Residual network training on ImageNet-1K with improved accuracy and reduced time to train," *arXiv preprint arXiv:1711.04291*, 2017.

[3] "Extremely large minibatch SGD: Training ResNet-50 on ImageNet in 15 minutes."

[4] C. Ying, S. Kumar, D. Chen, T. Wang, and Y. Cheng, "Image classification at supercomputer scale," *arXiv preprint arXiv:1811.06992*, 2018.

[5] H. Mikami, H. Suganuma, Y. Tanaka, Y. Kageyama *et al.*, "Massively distributed SGD: ImageNet/ResNet-50 training in a flash," *arXiv preprint arXiv:1811.05233*, 2018.

[6] 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 *IEEE Conference on Computer Vision and Pattern Recognition*, June 2019.

[7] H. Lee, M. Turilli, S. Jha, D. Bhowmik, H. Ma, and A. Ramanathan, "DeepDriveMD: Deep-learning driven adaptive molecular simulations for protein folding," in *IEEE/ACM 3rd Workshop on Deep Learning on Supercomputers*. IEEE, 2019, pp. 12–19.

[8] J. Carrasquilla and R. G. Melko, "Machine learning phases of matter," *Nature Physics*, 2017.

[9] J. Kates-Harbeck, A. Svyatkovskiy, and W. Tang, "Predicting disruptive instabilities in controlled fusion plasmas through deep learning," *Nature*, vol. 568, no. 7753, pp. 526–531, 2019.

[10] S. McCandlish, J. Kaplan, D. Amodei, and O. D. Team, "An empirical model of large-batch training," *arXiv preprint arXiv:1812.06162*, 2018.

[11] 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.

[12] Y. You, J. Hseu, C. Ying, J. Demmel, K. Keutzer, and C.-J. Hsieh, “Large-batch training for LSTM and beyond,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC 19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3295500.3356137

[13] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 770–778.

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

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

[16] L. Ma, G. Montague, J. Ye, Z. Yao, A. Gholami, K. Keutzer, and M. W. Mahoney, “Inefficiency of k-fac for large batch size training,” 2019.

[17] “MLPerf,” https://www.mlperf.org/.

[18] D. C. Liu and J. Nocedal, “On the limited memory bfgs method for large scale optimization,” *Mathematical programming*, vol. 45, no. 1-3, pp. 503–528, 1989.

[19] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, “Pytorch: An imperative style, high-performance deep learning library,” in *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[20] A. Sergeev and M. D. Balso, “Horovod: Fast and easy distributed deep learning in TensorFlow,” *arXiv preprint arXiv:1802.05799*, 2018.

[21] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, “ImageNet: A large-scale hierarchical image database,” in *IEEE Conference on Computer Vision and Pattern Recognition*, 2009, pp. 248–255.

[22] Intel, “Intel(r) machine learning scaling library,” 2019, https://github.com/intel/MLSL.

[23] 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*, 2016.

[24] “Gloo: Collective communications library with various primitives for multi-machine training,” https://github.com/facebookincubator/gloo.

[25] R. Thakur, R. Rabenseifner, and W. Gropp, “Optimization of collective communication operations in MPICH,” *International Journal of High Performance Computing Applications*, vol. 19, no. 1, pp. 49–66, 2005.

[26] B. Ginsburg, I. Gitman, and Y. You, “Large batch training of convolutional networks with layer-wise adaptive rate scaling,” 2018.

[27] B. Recht, C. Re, S. Wright, and F. Niu, “Hogwild: A lock-free approach to parallelizing stochastic gradient descent,” in *Advances in Neural Information Processing Systems*, 2011, pp. 693–701.

[28] S. Zhang, A. E. Choromanska, and Y. LeCun, “Deep learning with elastic averaging SGD,” in *Advances in Neural Information Processing Systems*, 2015, pp. 685–693.

[29] P. H. Jin, Q. Yuan, F. Iandola, and K. Keutzer, “How to scale distributed deep learning?” *arXiv preprint arXiv:1611.04581*, 2016.

[30] I. Mitliagkas, C. Zhang, S. Hadjis, and C. Ré, “Asynchrony begets momentum, with an application to deep learning,” in *54th Annual Allerton Conference on Communication, Control, and Computing*. IEEE, 2016, pp. 997–1004.

[31] 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 Symposium on Operating Systems Design and Implementation*, 2014, pp. 583–598.

[32] D. Alistarh, C. De Sa, and N. Konstantinov, “The convergence of stochastic gradient descent in asynchronous shared memory,” in *ACM Symposium on Principles of Distributed Computing*. ACM, 2018, pp. 169–178.

[33] R. Grosse and J. Martens, “A Kronecker-factored approximate Fisher matrix for convolution layers,” *arXiv e-prints*, p. arXiv:1602.01407, Feb. 2016.

[34] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, “MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems,” *arXiv preprint arXiv:1512.01274*, 2015.

[35] P. Patarasuk and X. Yuan, “Bandwidth optimal all-reduce algorithms for clusters of workstations,” *Journal of Parallel and Distributed Computing*, vol. 69, no. 2, pp. 117–124, 2009.

[36] J. Ba, R. B. Grosse, and J. Martens, “Distributed second-order optimization using kronecker-factored approximations,” in *ICLR*, 2017.

[37] J. Martens, J. Ba, and M. Johnson, “Kronecker-factored curvature approximations for recurrent neural networks,” 2018.

[38] C. Wang, R. Grosse, S. Fidler, and G. Zhang, “EigenDamage: Structured pruning in the Kronecker-factored eigenbasis,” in *Proceedings of the 36th International Conference on Machine Learning*, vol. 97. PMLR, 2019, pp. 6566–6575. [Online]. Available: http://proceedings.mlr.press/v97/wang19g.html

[39] T. George, C. Laurent, X. Bouthillier, N. Ballas, and P. Vincent, “Fast approximate natural gradient descent in a kronecker-factored eigenbasis,” in *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, ser. NIPS18. Red Hook, NY, USA: Curran Associates Inc., 2018, p. 95739583.

[40] J. Martens, “Kfac-tensorflow,” 2019. [Online]. Available: https://github.com/tensorflow/kfac

[41] A. Krizhevsky, G. Hinton *et al.*, “Learning multiple layers of features from tiny images,” University of Toronto, Technical Report TR-2009, 2009.

[42] “Keras applications,” https://keras.io/api/applications/.

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

We ran the ResNet model family with Cifar10 and ImageNet-1k dataset on the double-precision GPU subsystem of TACC's Frontera Supercomputer. This machine is powered by IBM Power9 processors and has 112 nodes in total. There are 4 Nvidia V100 GPUs, 256 GB RAM, and 1 TB rotational disk per node. The nodes are connected by an InfiniBand EDR network. We use PyTorch v1.1 and Horovod v0.19.0. These software frameworks rely on CUDA 10.0, CUDNN 7.6.4, and NCCL 2.4.7.

ARTIFACT AVAILABILITY

Software Artifact Availability: All author-created software artifacts are maintained in a public repository under an OSI-approved license.

Hardware Artifact Availability: There are no author-created hardware artifacts.

Data Artifact Availability: There are no author-created data artifacts.

Proprietary Artifacts: None of the associated artifacts, author-created or otherwise, are proprietary.

Author-Created or Modified Artifacts:

```
Persistent ID:
↪  https://github.com/gpauloski/kfac\_pytorch
Artifact name: PyTorch Distributed KFAC Optimizer
Citation of artifact: 10.5281/zenodo.3871837
```

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

Relevant hardware details: Frontera, Longhorn, IBM Power9 system, Power9 CPU, Nvidia V100 GPUs, Lustre and

Operating systems and versions: Red Hat Enterprise Linux Server 7.6 running Linux Kernel 4.14.0-115.10.1.el7a.ppc64le

Compilers and versions: GCC 4.8.5

Applications and versions: ResNet v1.5

Libraries and versions: PyTorch v1.1, Horovod v0.19.0

Key algorithms: K-FAC approximation

Input datasets and versions: Imagenet-1k, Cifar10