Deep Neural Network Training with Distributed K-FAC

J. Gregory Pauloski, Lei Huang, Weijia Xu, Kyle Chard, Ian T. Foster, and Zhao Zhang

Abstract—Scaling deep neural network training to more processors and larger batch sizes is key to reducing end-to-end training time; yet, maintaining comparable convergence and hardware utilization at larger scales is challenging. Increases in training scales have enabled natural gradient optimization methods as a reasonable alternative to stochastic gradient descent and variants thereof. Kronecker-factored Approximate Curvature (K-FAC), a natural gradient method, preconditions gradients with an efficient approximation of the Fisher Information Matrix to improve per-iteration progress when optimizing an objective function. Here we propose a scalable K-FAC algorithm and investigate K-FAC's applicability in large-scale deep neural network training. Specifically, we explore layer-wise distribution strategies, inverse-free second-order gradient evaluation, and dynamic K-FAC update decoupling, with the goal of preserving convergence while minimizing training time. We evaluate the convergence and scaling properties of our K-FAC gradient preconditioner, for image classification, object detection, and language modeling applications. In all applications, our implementation converges to baseline performance targets in 9–25% less time than the standard first-order optimizers on GPU clusters across a variety of scales.

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

1 Introduction

Deep neural networks (DNNs) have revolutionized how tasks in classification, object detection, segmentation, language modeling, and more are solved. As the computational needs for DNN training have grown due to larger models and datasets, there has been a growing interest in exploiting the powerful memory and communication architectures available on high-performance computing (HPC) systems [1–6]. The intersection of deep learning and HPC has specifically enabled new uses of deep learning for scientific applications [7–9].

Supercomputers can yield significant speedups in training time, but it is an open challenge to efficiently utilize available hardware without harming convergence (e.g., loss or validation accuracy) [10]. Many prior approaches are focused on first-order optimization methods such as stochastic gradient descent (SGD) [11]. Significant work has been devoted to understanding and improving the scaling properties of SGD [1, 3–5, 11, 12]. Impressive results have been shown for specific applications such as ResNet-50 [13] and BERT [14]; however, improvements are often made possible via techniques specific to the application or hardware such as distributed batch normalization and communication optimizations, respectively.

SGD alternatives that incorporate second-order information, such as natural gradient methods, have been explored more recently as incorporating second-order information can improve the per-iteration progress made when optimizing an objective function. Kronecker-factored Approximate Curvature (K-FAC), a second-order method, has shown promising results due to K-FAC's efficient approximation of the Fisher Information Matrix (FIM) [6, 15, 16]. K-FAC significantly reduces iterations required for convergence and scales well to $O(1000)$ GPUs [6], but prior implementations have only been evaluated on ResNet-like convolution models and often struggle to either maintain comparable convergence to the acceptable performance baselines [17] or exceed the time-to-convergence of SGD.

In this paper, we investigate methods for designing K-FAC-based optimizers that reduce iterations-to-convergence and exhibit efficient scaling. We investigate the performance, in terms of convergence and time-per-iteration, of an explicit matrix inverse algorithm and implicit eigen decomposition algorithm. We exploit a conventional method in L-BFGS [18] to decouple the approximation of the FIM from gradient preconditioning which allows us to recompute the FIM less frequently which speeds up training time, and we investigate the effect this frequency has on final convergence. We analyze the training time and model convergence of K-FAC specific hyperparameter schedules such as damping and K-FAC update frequency decay.

We implement our preconditioner in the popular PyTorch framework [19] and evaluate with a suite of reference applications including ResNets [13], Mask R-CNN [20], and BERT [14] on two clusters with different hardware. We find that our implementation converges to or exceeds the baseline performance metrics in fewer iterations and 9–25% less time across all applications.

This paper extends our prior work [21] by improving our K-FAC preconditioner implementation and broadening our evaluation to new model architectures. Specifically, we 1) expand our comparisons with other K-FAC work, 2) implement support for more training configurations (communication backends, data-parallel training frameworks, mixed-

- *Z. Zhang, L. Huang, and W. Xu are with the Texas Advanced Computing Center.*
  E-mail: zzhang, huang, xwj@tacc.utexas.edu
- *J. G. Pauloski, K. Chard, and I. T. Foster are with the Department of Computer Science at the University of Chicago.*
  E-mail: jgpauloski, chard, foster@uchicago.edu

precision training, gradient accumulation), 3) to encourage further research, extend our open-source implementation to support prior K-FAC distribution strategies and algorithms [6] in addition to our own novel methods, 4) provide evaluations on a second GPU cluster with a different hardware architecture, 5) extend the convergence and scaling evaluations to include the Mask R-CNN and BERT MLPerf applications, and 6) provide a detailed profiling of the K-FAC algorithm to understand how our distribution strategy improves time-to-converegence compared to prior work.

We focus on the convergence capabilities of K-FAC and refer readers to our related work, KAISA, which investigates the memory and communication tradeoffs in distributed K-FAC strategies [22]. We make the following contributions:

A scalable distributed K-FAC strategy;
A study of K-FAC gradient preconditioning variants;
An analysis of K-FAC update intervals and optimal values for our applications;
A study of the convergence abilities of K-FAC in three application with different distributed frameworks;
An open source implementation of the proposed algorithm using PyTorch [19].

The remainder of the paper is organized as follows. We discuss parallelism in DL training with SGD and K-FAC in §2, review related work in large-scale DL training in §3, introduce our system design choices in §4, provide technical details in §5, present a detailed experimental evaluation in §6, and finally draw conclusions in §7.

2 Background

We review different forms of parallelism in DL training, data parallelism in iterative batch optimization methods, the K-FAC preconditioning method, and frameworks that enable distributed DL training.

2.1 Data Parallelism

Data parallelism, *model parallelism*, and *hybrid parallelism* (i.e., a combination of data and model parallelism), are the typical methods used to distribute DL training across more than one processor. Data parallelism, in which the entire model is replicated on each processor, and in every iteration a unique mini-batch is consumed by each processor, is the common choice for scaling single-processor training to many processors. Model parallelism is beneficial when training cannot be performed on a single processor, such as in the case where the model cannot fit in processor memory; here, a single model instance is split across processors, and this model division can happen layer-wise or even within a layer. Hybrid approaches, combining data and model parallelism in various ways, can also be used. For example, processors can be grouped such that model parallelism is used across processors within a group and data parallelism is used between groups; alternatively, different parallelism methods can be used on a per-layer basis.

Conventionally, model parallelism is used only when data parallelism is not possible, because the use of model parallelism can reduce machine utilization and often requires additional optimizations such as pipeline parallelism [23]. As data parallelism is the dominant form of large scale DL training, numerous frameworks provide native support via NCCL, Gloo [24], or MPI collective operations [25]: for example, Intel MLSL [26], Horovod [27], TensorFlow [28], PyTorch [19], and NVIDIA Apex [29].

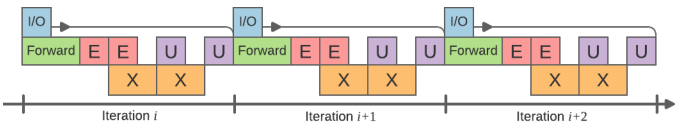


Fig. 1: Synchronous SGD iteration stages. I/O: Batch creation. Forward: Forward pass and loss computation. E: Gradient evaluation (backpropagation). X: Gradient exchange. U: Trainable variable update [21].

2.2 Stochastic Gradient Descent

In SGD, a batch optimization method [11], mini-batches of training data are iteratively passed through the network to compute a loss which is used for gradient computation and variable update. Data parallel training with SGD takes two forms, synchronous [30] and asynchronous [31–35], and are distinguished by whether all variables are updated each iteration. We focus on synchronous SGD methods in this work because asynchronous SGD has a non-linear slowdown compared with the synchronous form [36].

An iteration of synchronous data-parallel SGD has five stages, depicted in Figure 1. In the I/O stage, a fixed-size batch of training data is read from storage, pre-processed, and potentially moved to the local memory of the processor. The batch is processed by the model in the forward pass phase and the output is used to compute a loss. The gradient evaluation stage, often referred to as backpropogation, uses the loss to compute the gradient for each trainable variable. Then, there is a collective communication operation between processors to exchange the gradients. In the final stage, each processor updates the variables in their local copy of the model. Key to achieving high hardware utilization in modern frameworks is the exploitation of asynchronous I/O methods for overlapping training data retrieval and gradient exchange with other stages.

The communication required by synchronous SGD involves communicating initial model weights and the gradient exchange in each iteration, and these communications are generally performed with the *broadcast* and *allreduce* collective operations, respectively.

2.3 K-FAC

K-FAC is an efficient natural gradient method that uses the Fisher Information Matrix (FIM) which encodes the curvature of the loss function. Here we briefly summarize the math behind K-FAC, and we adopt the notation from our prior work [21].

In the weight update equation for SGD (Equation 1), the weight at the $k+1$ iteration ($w^{(k+1)}$) is the difference of the current weight ($w^{(k)}$) and the product of the current learning rate ($a^{(k)}$) with gradient of the loss for the mini-batch of size n. The gradient of the loss is computed as the average of the

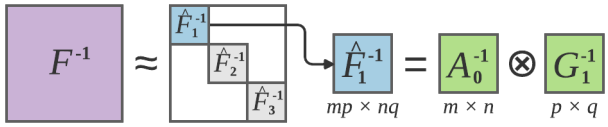


Fig. 2: The K-FAC approximation of the FIM. A Kronecker product is symbolized with [21].

gradient with respect to the loss over each i^{th} example in the mini-batch, denoted as $\nabla L_i(w^{(k)})$.

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

With K-FAC, the gradient is preconditioned by $F^{-1}(w^k)$, the inverse of the FIM for the weights.

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

In this context, precondition means to transform a matrix to accelerate iterative optimization algorithms, and in this case the gradients are transformed by the FIM.

In practice, computing F for an entire model is intractable so K-FAC makes a layer-wise independence assumption of the model to block diagonalize F as $\hat{F}$ where the i^{th} block corresponds to the i^{th} of L layers in the model.

$$\hat{F} = \texttt{diag}(\hat{F}_1, ..., \hat{F}_i, ..., \hat{F}_L) \tag{3}$$

Each $\hat{F}_i$ is then further decomposed as a Kronecker-factorization of two smaller matrices. A Kronecker product, written as $A \otimes B$ where A has size $m \times n$ and B has size $p \times q$ produces an $mp \times nq$ matrix:

$$A \otimes B = \begin{bmatrix} a_{11}B & \dots & a_{1n}B \\ \vdots & \ddots & \vdots \\ a_{m1}B & \dots & a_{mn}B \end{bmatrix}. \tag{4}$$

In K-FAC, the Kronecker factors for layer i, often referred to as covariance matrices, are computed from the activations of the previous layer, a_{i-1}, and the gradients of the output of the current layer, g_i. The resulting equation for $\hat{F}_i$ is

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

The Kronecker product has a convenient property that $(A \otimes B)^{-1} = A^{-1} \otimes B^{-1}$ which allows $\hat{F}_i^{-1}$ to be more efficiently calculated using the inverses of the smaller factors, as shown in Figure 2.

$$\hat{F}_i^{-1} = A_{i-1}^{-1} \otimes G_i^{-1} \tag{6}$$

$\hat{F}_i^{-1}$ is used to precondition the gradient of the parameters, $\nabla L(w_i^{(k)})$, for layer i at iteration k.

$$w_i^{(k+1)} = w_i^{(k)} - \alpha^{(k)} \hat{F}_i^{-1} \nabla L(w_i^{(k)}) \tag{7}$$

However, the space complexity of the Kronecker product means that $\hat{F}_i^{-1}$ is a relatively large matrix, so we apply the relation $(A \otimes B)\vec{c} = B^{\top}\vec{c}A$ to reduce the complexity of preconditioning. The result is a more efficient form of preconditioning the gradient with the two factors.

$$\hat{F}_i^{-1} \nabla L(w_i^{(k)}) = G_i^{-1} \nabla L(w_i^{(k)}) A_{i-1}^{-1}. \tag{8}$$

Tikhonov regularization is applied where γI is added to each factor [6, 37]. The addition of the damping parameter γ compensates for inherent inaccuracies that cause a matrix to be ill-conditioned for inversion. There are many methods for computing γ [6, 15]. $(\hat{F}_i + \gamma I)^{-1}$ is computed as

$$(\hat{F}_i + \gamma I)^{-1} = (A_{i-1} + \gamma I)^{-1} \otimes (G_i + \gamma I)^{-1} \tag{9}$$

such that the final weight update equation at iteration k is

$$w_i^{(k+1)} = w_i^{(k)} - \alpha^{(k)} (G_i + \gamma I)^{-1} \nabla L(w_i^{(k)}) (A_{i-1} + \gamma I)^{-1}. \tag{10}$$

2.4 Frameworks

We develop our K-FAC preconditioner using PyTorch [19] and Horovod [27].

PyTorch is widely adopted in the research community due to its performant C++ runtime, comprehensive support for layer and architectures types, and pythonic code structure that enables easy research and development. Specific to our use case is the PyTorch module hook interface that allows saving the intermediate values necessary for K-FAC computations. PyTorch also provides an interface to collective communication in MPI, NCCL, or Gloo [24] and the *DistributedDataParallel* model wrapper that enables data parallel training.

We also use Horovod [27], a library for data parallel training with support for many major frameworks including PyTorch, TensorFlow, and MXNet [38]. Similar to PyTorch, Horovod provides MPI concepts such as *size*, *rank*, and *local rank* and collective communication operations (*allreduce*, *allgather*, and *broadcast*). Horovod backs these interfaces with MPI, NCCL, or IBM Distributed Deep Learning Library (DDL) primitives. Horovod provides many optimization such as an implementation of *allreduce* via the scatter-reduce algorithm, synchronous and asynchronous collective operations, and user-configurable fusion buffers for batching communication operations.

3 Related Work

While data parallelism is key to scaling DL training, data parallelism necessitates scaling the batch size by the number of processors to maintain high utilization. Larger batch sizes have been shown to produce models which do not generalize as well, and this inhibits practical training of DNNs on hundreds or thousands of processors [39]. Batch sizes are not the only limiting factors at the largest scales; communication costs can also become prohibitive because the data needed to be exchanged is proportional to the processor count. In this section, we summarize prior work that addresses theses limitations in large-scale DL training.

3.1 Scaling Results of SGD

Early work in scaling SGD training focused on adapting the learning rate by instituting a warmup and scaling the learning rate by the batch size [40]. Later work, LARS [1] and LAMB [12], iterated on this idea with an SGD variant that adapts the learning rate per-layer to enable even larger batch sizes while maintaining convergence. This method enabled ResNet-50 training on ImageNet-1k in 20 minutes on 2048 Intel Xeon Platinum 8160 processors and BERT training in 76 minutes on a TPUv3 Pod with batch sizes of 32K for both applications [1, 12]. Many works have leveraged these optimizers with dedicated enhancements for the system architectures (e.g., 2D torus) and hardware improvements such as GPUs and TPUs. These iterative improvements have resulted in researchers training ResNet-50 to 76.3% validation accuracy on ImageNet-1k in 2.2 minutes on 1024 TPUs [4]. Many of these techniques, such as mixed precision, interconnect-aware *allreduce*, and distributed batch normalization, are application or hardware specific, so in our work we focus on a general optimization algorithm that is hardware, network topology, and application agnostic.

3.2 Scaling Results of K-FAC

K-FAC can reduce iterations-to-convergence in image classification tasks with ResNets [6, 41] and natural language tasks with RNNs [42]. The gradient preconditioning in K-FAC, described in §2.3, results in much slower iterations than SGD, so previous work used an asynchronous K-FAC distributions scheme with a doubly-factored Kronecker approximation to achieve iteration times on par with SGD for ImageNet-1k training [41]. The 2 training speedup on eight GPUs is attributed to the faster convergence at the start of training; however, the optimizations necessary to achieve this speedup result in a low final accuracy of 70%. In contrast, we use a synchronous scheme and single-factored Kronecker approximation to maintain convergence.

More recently, researchers have found greater benefits to K-FAC in larger-scale training environments because the factors corresponding to the FIM approximation for each layer can be distributed across GPUs in a model parallel fashion [6]. They showed that 978 iterations are sufficient to reach 74.9% validation accuracy on ImageNet-1k in 10 minutes with K-FAC. While the reduction in iterations is impressive, they did not show K-FAC to be faster than, nor achieve the same validation accuracy as, SGD.

Later work from the same group addressed these limitations and reached the MLPerf baseline in 10.5 minutes on 512 GPUs [43] without the use of a stale FIM approximation. They introduced a novel 21-bit floating point specification to reduce communication, and carefully analyzed and optimized the baseline SGD code. We show in §6.3.1 that the MLPerf baseline can be reached with stale FIM approximations and find in practice that the standard FP16 and FP32 formats provided by PyTorch are sufficient for exceeding SGD performance. We also show that our novel K-FAC design reaches MLPerf acceptance baselines in less wall-time than SGD, a set of criteria not met by prior work but necessary to showcase K-FAC as a viable tool for practitioners.

Scalable and Practical Natural Gradient Descent (SP-NGD) can scale to large batch sizes with minimal overheads [44]. SP-NGD reaches 74.9% ImageNet-1k validation accuracy in only 873 steps, an improvement over prior work, but the method results in sub-MLPerf baseline performance even at small batch sizes. In our work, we address the more general question of can K-FAC easily enable both training time and convergence improvements without the need for careful tuning of all aspects of the baseline code.

KAISA, a K-FAC-enabled, adaptable, improved, and scalable second-order optimizer framework, generalizes the the distributed strategy of this work and prior work [6] and investigates the tradeoffs between memory and communication [22]. Whereas previous K-FAC works have generally focused on comparing K-FAC and SGD with fixed batch sizes, this work concludes that K-FAC can also enable faster-than-SGD convergence with fixed memory budgets.

4 Design

Recent research has shown how the K-FAC overhead in distributed training can be reduced by assigning each layer to a processor so that the K-FAC updates for each layer can be computed in parallel [6]. However, that method has two primary limitations: 1) workers are left idle if the number of layers is less than the number of workers and 2) communication of the preconditioned gradients is required at every iteration regardless of if stale second-order information is used. Our design improves on this method by increasing the granularity of K-FAC computations, reducing the frequency of communication, and maintaining convergence to MLPerf baselines on benchmarks. Our open source K-FAC code is designed as a preconditioner rather than an optimizer so that it can be used in place with any existing optimizer, such as SGD, Adam, or LARS, to gain additional benefits of other optimization techniques.

4.1 Matrix Inversion

The largest computational cost of the K-FAC update step is computing the inverse of $(A_{i\ 1} + \gamma I)$ and $(G_i + \gamma I)$ to precondition the gradients (Equation 9). Existing distributed K-FAC implementations use this method [6, 41, 43], but the gradients can alternatively be preconditioned using an eigen decomposition of the factors [37].

$$V_1 = Q_G^{\top} L(w_i^{(k)}) Q_A \tag{11}$$

$$V_2 = V_1 / (\upsilon_G \upsilon_A^{\top} + \gamma) \tag{12}$$

$$(\hat{F}_i + \gamma I)^{\ 1}\, r\, L(w_i^{(k)}) = Q_G V_2 Q_A^{\top} \tag{13}$$

Here, the eigen decompositions of the factors are $A_{i\ 1} = Q_A \Lambda_A {Q_A}^{1}$ and $G_i = Q_G \Lambda_G {Q_G}^{1}$, and υ_A and υ_G are vectors of the eigenvalues of $A_{i\ 1}$ and G_i, i.e., the diagonals of Λ_A and Λ_G.

We refer to this method as the implicit eigen decomposition method and use it in our work. We compare the explicit inverse and implicit eigen decomposition methods in Table 1. The final validation accuracy for ResNet-32 trained on CIFAR-10 is consistently better as a function of batch size in the eigen decomposition method.

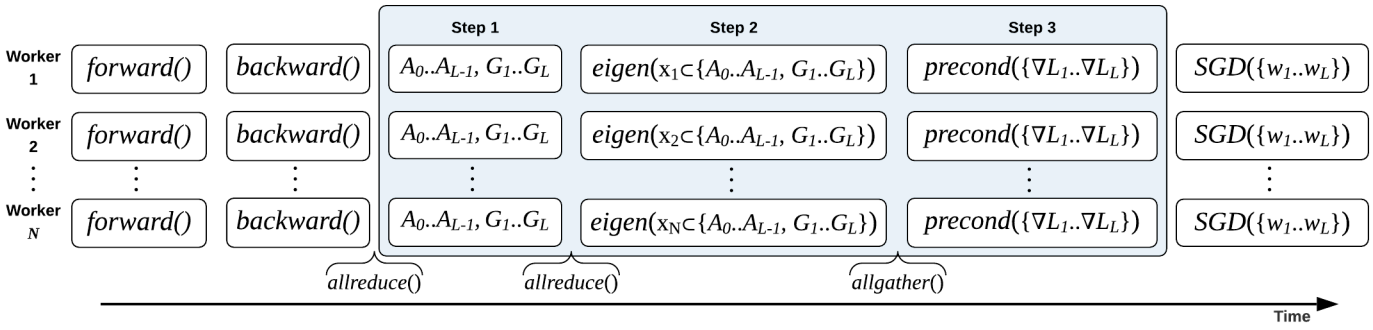


Fig. 3: Overview of a distributed K-FAC update step. The forward pass, backward pass, and gradient allreduce are computed as normal in synchronous SGD. The K-FAC update, shaded in blue, has three steps. In step 1, each worker computes the factors for all layers using the intermediate values cached in the forward and backward passes. The factors are then allreduced across workers prior to step 2 where the eigen decompositions of the factors are computed in a model parallel fashion. That is, each worker eigen decomposes a unique subset of the factors. The resulting eigen decompositions are then communicated to all workers such that in step 3, each worker can precondition the gradients for all layers. After the K-FAC update, the weight update is performed as normal with SGD or a similar optimizer.

TABLE 1: Validation accuracy with inverse and eigen decomposition K-FAC methods for ResNet-32 trained on CIFAR-10 [21].

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

Algorithm 1: K-FAC update pseudocode.

```
   // Compute Gradients
1  foreach worker do
2  |  Compute forward and backward pass
3  end
4  allreduce(∇L_{1:L}(w_{1:L}))
   // Step 1: Compute Factors
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})
   // Step 2: Compute Eigen Decompositions
9  foreach worker w do
10 |  foreach A_i assigned to w do
11 |  |  Q_{A_i}, Λ_{A_i} = eigendecompose(A_i)
12 |  end
13 |  foreach G_j assigned to w do
14 |  |  Q_{G_j}, Λ_{G_j} = eigendecompose(G_j)
15 |  end
16 end
17 allgather(Q_{A_{0:L-1}}, Λ_{A_{0:L-1}}, Q_{G_{1:L}}, Λ_{G_{1:L}})
   // Step 3: Precondition Gradients
18 foreach worker do
19 |  precondition(∇L_{1:L}(w_{1:L}))
20 end
   // Update Weights with SGD
21 foreach worker do
22 |  Update weights using preconditioned gradients
23 end
```

4.2 Parallelism

The K-FAC gradient preconditioning process has three steps, outlined in Figure 3 and Algorithm 1 that occur after the forward and backward pass and before weight update.

In the first step (lines 5–8 in Algorithm 1), the Kronecker factors $A_{i\ 1}$ and G_i are computed as a running average of the individual factors computed from each mini-batch using ξ, a running average hyper-parameter in the range $[0.9, 1)$.

$$A_i^{(k)}{}_1 = \xi A_i^{(k)}{}_1 + (1 \quad \xi) A_i^{(k}{}_1{}^{1)} \tag{14}$$

$$G_i^{(k)} = \xi G_i^{(k)} + (1 \quad \xi) G_i^{(k}{}^{1)} \tag{15}$$

Each worker uses the intermediate values saved during the forward and backward passes to update the running averages of the factors in a data-parallel fashion. At the end of the first step, the factors are *allreduced* across workers.

The second step in the K-FAC update transitions to model-parallelism where individual calculations are assigned to different workers (lines 9–17 in Algorithm 1). Existing implementations [6] assign each worker a layer in the model such that the worker computes $A_{i\ 1}^{\ 1}$, $G_{j\ 1}^{\ 1}$, and the final preconditioned gradient $(\hat{F}_i + \gamma I)^{\ 1}\ r\ L_i(w_i^{(k)})$. Workers then share their preconditioned gradients so workers can update local weights.

We approach the model parallel stage differently and assign each worker a subset of factors to eigen decompose. At the end of the second step, the eigen decompositions are communicated to all workers.

In the third and final step, gradients are preconditioned locally using the eigen decompositions which are now cached locally due to the communication in the prior step (lines 18–20 in Algorithm 1). We detail how decoupling the eigen decomposition and gradient preconditioning steps is key to reducing communication in §4.3.

Once the gradients are preconditioned, any standard optimizer, such as SGD, can be used to update the weights.

4.3 Communication

Training with our K-FAC design requires three types of communication: 1) gradient *allreduce*, 2), Kronecker factor *allreduce*, and 3) eigen decomposition *broadcast* or *allgather*.

Using stale Kronecker factors is commonly used to reduce the average K-FAC iteration time. In this case, the factors are eigen decomposed every n iterations. As a consequence, in iterations where eigen decompositions are not recomputed, the communication in (2) and (3) can be avoided

because all workers cache prior eigen decompositions locally in our design. In Algorithm 1, this is represented as skipping the first and second steps (lines 5–17).

The staleness of the factors increases with n, but the factors stabilize over the course of training because the factors are a running average over all mini-batches. In practice, values of n in the tens or hundreds are acceptable for maintaining convergence. Since K-FAC communications are required less frequently as n increases in this design, minimal extra communication is required over standard data-parallel SGD training in the limit of n.

We contrast this approach with that of prior work where the gradients for a specific layer are preconditioned on a single worker so additional communication is required every iteration regardless of n [6]. Decoupling eigen decomposition and preconditioning also allows double the maximum worker utilization because eigen decompositions for the two factors of a single layer can be performed on different workers.

5 Implementation

In this section, we describe our existing PyTorch implementation [21] and modifications to support a more diverse range of applications. Our K-FAC code is open source with the MIT license and is hosted at https://github.com/gpauloski/kfac_pytorch. Code used for the experiments in §6 is either directly provided or linked to in this repository.

5.1 K-FAC Usage

Core to our goal of enabling easy and fast training with K-FAC is our choice to implement K-FAC as a gradient preconditioner. Practioners can use K-FAC with two lines of code: initializing the preconditioner and calling the *step()* function. In the initialization, hooks are registered to all Conv2D and Linear modules for saving the necessary intermediate data. The K-FAC *step()* function, called prior to the optimizer's *step()* function, performs the K-FAC update to precondition the gradients.

5.2 Improved Application Support

We extend our prior implementation to support a wider variety of applications and use-cases for K-FAC.

5.2.1 Communication Backends

We extend our distributed training support from only Horovod to the distributed data parallel model wrappers provided by PyTorch [19], NVIDIA Apex [29], and DeepSpeed [45]. With Horovod, we use the provided Horovod interfaces to collective communication operations, and with PyTorch, NVIDIA Apex, and DeepSpeed, we use the native PyTorch interfaces to collective communication operations. In both cases, we use asynchronous variants of the operations to overlap communication with computation.

Horovod implements data-parallel training via a custom wrapper for the optimizer, so the gradient communication is performed in the optimization step. Since K-FAC needs to precondition the final averaged gradients prior to the optimization step, the Horovod optimizer's *synchronize()* must be called prior to the K-FAC *step()*.

PyTorch, NVIDIA Apex, and DeepSpeed provide model wrappers rather than optimizer wrappers so gradient communication is performed during the backwards pass. Thus, no additional changes are needed, and K-FAC can be used as described in §5.1.

5.2.2 Mixed Precision Training

Mixed precision training is becoming increasingly important for training larger models as hardware support for FP16 operations has improved [46]. We provide support for storing and communicating the factors and eigen decompositions in FP32 or FP16. CUDA does not allow eigen decomposition with FP16 matrices so we cast to FP32 and back if the factors are stored in FP16.

5.2.3 Gradient Accumulation

Gradient accumulation is a process where multiple forward and backward passes are performed between optimization steps. The result is that the effective batch size is scaled by the number of passes per optimization step. Previously, our implementation collated each mini-batch between K-FAC *step()* calls into a larger batch and computed the factors from the larger batch in the K-FAC *step()*. However in applications that require many gradient accumulation iterations, the K-FAC memory usage grows linearly so we adapt our code to provide support for computing factors for individual mini-batches during the forward and backward passes.

5.3 K-FAC Hyper-Parameters

Our K-FAC preconditioner introduces hyper-parameters for factor update interval, damping, and gradient scaling.

The *kfac-update-freq* parameter controls the number of iterations between eigen decomposition updates and communication. We find the running average of the factors can be updated and communicated at a frequency 10 that of the *kfac-update-freq* without affecting the convergence. As mentioned previously, the factors become more stable throughout training so at fixed iterations, we decrease the *kfac-update-freq* by a scalar factor to reduce the computation and communication. Small improvements can be gained via advanced *kfac-update-freq* schedules, but we found that a constant value was generally sufficient in terms of performance for the entirety of training.

Similarly, we use a fixed damping decay schedule such that larger damping values early account for rapid changes in the FIM at the start of training [6].

We scale preconditioned gradients G_i by ν to prevent the norm of G_i becoming large compared to w_i [6], where ν is computed as

$$\nu = \min\left(1, \sqrt{\frac{\kappa}{\alpha^2 \sum_{i=1}^{n} |\mathsf{G}_i^{\top} \nabla L_i(w_i)|}}\right) \tag{16}$$

using a user-defined constant κ [47,48]. We found values for κ on the order of 10^{-3} were sufficient for all of our applications.

5.4 Math Libraries

We perform matrix eigen decomposition and inversion on the GPU. Our code supports PyTorch 1.6 and later, but we

note the underlying libraries used for eigen decomposition and inversion depend on the PyTorch version. In PyTorch 1.7 and older, we use *torch.symeig()* for eigen decomposition and *torch.inverse()* for matrix inversion. The functions are deprecated in favor of *torch.linalg.eigh()* and *torch.linalg.inv()* in PyTorch 1.8. The construction of the factors ensures the factors are real and symmetric so we can use eigen decomposition algorithms optimized for symmetric matrices. For the analysis in §4.1, we use CUDA 10.0 and PyTorch 1.6 so the matrix inverses are computed using MAGMA's *getrf* and *getri* routines. For symmetric eigen decomposition, PyTorch 1.8 and older uses MAGMA's *syevd* and *heevd* routines while PyTorch 1.9 and newer use the cuSolver equivalents.

6 Experiments

We select the ResNet model family, Mask R-CNN, and BERT to examine convergence and to compare with the original optimizers. These applications cover three typical DNN usage domains: image classification, object detection, and language modeling. For ResNet-50 and Mask R-CNN, we adopt the MLPerf [17] acceptance performance as baseline.

6.1 Platform and Software Stacks

We use two clusters in our experiments. The first is Longhorn, the GPU subsystem of the Frontera supercomputer hosted at the Texas Advanced Computing Center (TACC). Longhorn has 112 nodes each with two IBM Power9 processors, four NVIDIA V100 GPUs, and 256 GB of RAM. Nodes are connected by an InfiniBand EDR network. We run ResNet and Mask R-CNN training on Longhorn using PyTorch 1.6, CUDA 10.0, CUDNN 7.6.4, and NCCL 2.4.7. We use single-precision floating point numbers (FP32) for training and communication.

The second system is ThetaGPU, the GPU subsystem of the Theta supercomputer at Argonne National Laboratory. ThetaGPU has 24 NVIDIA DGXA100 nodes each with eight 40GB A100 GPUs (192 A100 GPUs in total). We perform BERT training on ThetaGPU with PyTorch 1.9, CUDA 11.3, CUDNN 8.2.0, and NCCL 2.9.9.

We enable distributed synchronous data-parallel training with Horovod for ResNet examples, NVIDIA Apex's *DistributedDataParallel* for Mask R-CNN, and PyTorch's native *DistributedDataParallel* for BERT.

6.2 Datasets and Applications

We use ResNet-34 [13] with the CIFAR-10 [49] dataset to test correctness. We then use the ImageNet-1k dataset [50] with the ResNet model family and the COCO 2014 [51] dataset with Mask R-CNN to evaluate the performance of K-FAC as a preconditioner to the original optimizers.

A common practice to evaluate BERT convergence is to fine tune the pretrained model for downstream tasks, such as SQuAD v1.1, a question and answer benchmark, and use the validation results of the downstream tasks as the convergence metric. Researchers have reported F1 scores of 91.08% using the NVIDIA implementation [52] for SQuAD v1.1. However, this case trains with the English Wikipedia [53] and the Toronto BookCorpus [54] datasets, the latter of which is no longer available online as a holistic package. So for BERT experiments, we train a BERT-Large uncased model with only the WikiText dataset and report the results with Fused-LAMB optimizer as the baseline.

Using these applications, we study the correctness, performance, and scalability of our K-FAC preconditioner and provide comparisons to the original optimizers.

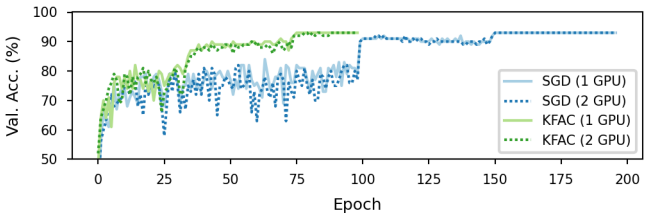


Fig. 4: Validation accuracy comparison of ResNet-32 on CIFAR-10 with KFAC and SGD [21].

TABLE 2: Validation accuracy comparison of ResNet-32 on CIFAR-10 with KFAC and SGD [21].

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

6.3 Convergence

We now present training results with K-FAC and compare the validation performance with the original optimizers in ResNet model family, Mask R-CNN, and BERT.

6.3.1 ResNet Convergence

We use the small ResNet-34 model trained on CIFAR-10 to confirm the correctness of our K-FAC preconditioner and adopt a 92.49% target for acceptable validation accuracy [13]. We use the same batch sizes and learning rates for both optimization methods. Specifically, the learning rate is N 0.1 and the batch size as N 128 where N is the number of GPUs. A linear learning rate warmup is used for the first five epochs and then decreased by a factor of 10 at epochs 35, 75, 90 for K-FAC and 100, 150 for SGD. We train for 100 epochs with K-FAC and twice as long with SGD. K-FAC eigen decompositions are recomputed every 10 iterations.

We repeat the experiment on 1, 2, 4, and 8 V100 GPUs and present the final validation accuracies in Table 2. Figure 4 depicts the validation accuracy throughout training in the 1 and 2 GPU runs. Our K-FAC implementation meets or exceeds the performance of SGD across a range of batch sizes while converging in fewer iterations.

Our second experiment compares K-FAC and SGD training with ResNet-50 and the ImageNet-1k dataset. We verify that K-FAC 1) converges to 75.9% validation accuracy, the MLPerf baseline; 2) converges to a validation accuracy that is at least equal to that of SGD; and 3) requires fewer iterations. We train on 16 V100 GPUs with a batch size of 32 16=512 and learning rate of 0.0125 16 = 0.2 The learning rate is linearly warmed up for the first five epochs then decayed at epochs 25, 35, 40, 45, and 50. Labels are smoothed with a factor of 0.1. The SGD momentum is 0.9, the K-FAC

damping value is 0.001, and the K-FAC approximation is updated every 10 iterations.

The Top-1 validation accuracy is presented in Figure 5a. K-FAC converges to 76.4% validation accuracy, outperforms SGD by 0.2%, and converges to the MLPerf baseline in 43 epochs compared to the 76 epochs required by SGD.

6.3.2 Mask R-CNN Convergence

Mask R-CNN has four components: the feature pyramid networks (FPN) that encode the image, the region proposal network (RPN) that generates bounding box proposals, the box head that detects bounding boxes and classifies objects, and the mask head that generates a pixel-wise mask for the detected object. The original implementation uses SGD as the optimizer. The validation baseline is 0.377 for bbox mAP (mean average precision) and 0.342 for segmentation mAP. We evaluate various ways to integrate K-FAC with SGD and find that training converges by enabling K-FAC for only the box head and mask head.

We empirically examine Mask R-CNN convergence with 32 V100 GPUs. We set the K-FAC update interval to 500 steps and the factorization interval to 50 steps. Figures 5b and 5c show the validation curves of the bbox and segm mAP. With SGD, these converge to the baseline at the end of 14th epoch with values of 0.378 and 0.342, respectively. In contrast, K-FAC converges to the baseline by the end of 12th epoch, with bbox mAP of 0.379 and segm mAP of 0.350. K-FAC converges to the baseline in 18.1% fewer steps than SGD on 32 V100 GPUs. The training time with K-FAC is 116 minutes—14.9% less than SGD.

6.3.3 BERT Convergence

In this experiment, we first measure the BERT-Large baseline with Fused-LAMB using the hyper-parameter values specified in the NVIDIA documentation [52]. Phase 1 trains with a maximum sequence length of 128 tokens for 7038 steps then Phase 2 trains with the maximum sequence length of 512 tokens for 1563 steps. We run the experiment on 16 A100 GPUs and fine tune using the SQuAD v1.1 training set, obtaining a validation F1 score of 90.1%, a 0.98% loss compared to the case trained with Wikitext and Toronto BookCorpus. We attribute the reduced F1 score to the unavailability of the Toronto BookCorpus dataset. The global batch size is 64K and 32K for Phase 1 and Phase 2, respectively. We use a local batch size of 96 for Fused-LAMB and 90 for K-FAC in Phase 1; in Phase 2, these values are 16 and 12, respectively.

Next, we determine the earliest step that SQuAD v1.1 converges to this baseline by reducing the training steps in Phase 2 with Fused-LAMB. The results in Table 3 show that, with Fused-LAMB, training requires the full 1563 steps in Phase 2 to converge to the 90.1% baseline.

We next examine convergence behavior when using K-FAC. We first apply K-FAC for Phase 2 only. We evaluate the K-FAC approximation every 50 steps, update A and G every five steps, and fine tune SQuAD v1.1 at 1000, 1200, 1400, and 1563 steps. We find (Table 4a) that with these settings, SQuAD v1.1 converges above the 90.1% baseline with as few as 1000 steps, a 36.0% reduction in iterations and 16.4% reduction in training time. The overall training time, combining Phase 1 and Phase 2, is reduced by 6.2%.

TABLE 3: SQuAD v1.1 F1 score with varying numbers of Phase 2 steps. LAMB is short for Fused-LAMB. highlights the diverged cases.

Phase 1		Phase 2		F1
Optimizer	Steps	Optimizer	Steps	
LAMB	7038	LAMB	1563	90.1%
LAMB	7038	LAMB	1400	89:94%
LAMB	7038	LAMB	1200	89:87%

In a second experiment, we train Phase 1 with K-FAC and Phase 2 with Fused-LAMB. Table 4b reports the converged cases with the fewest steps in each phase. In the best case, the number of steps is reduced by 14.7% and 23.2% for Phases 1 and 2, respectively, and overall training time is reduced by 9.0%.

Finally, we train both phases with K-FAC. Table 4c reports the results. K-FAC efficiently reduces the training steps to 5000 for Phase 1, but takes 1400 steps to converge in Phase 2. The overall training time is reduced by 4.7%.

6.3.4 K-FAC Configuration

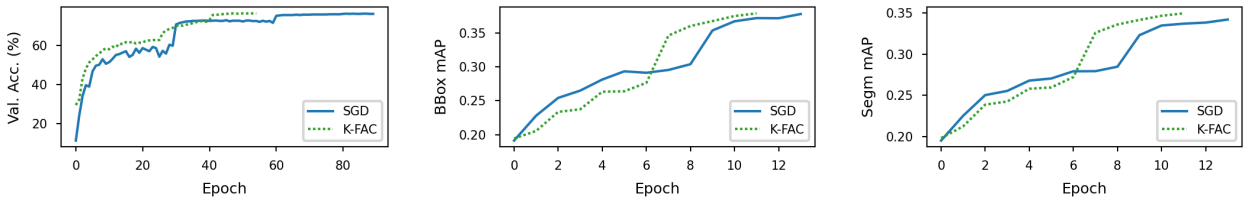
Key to efficient training with second-order methods is reducing the frequency at which second-order information is computed. In K-FAC, less frequent K-FAC updates reduce computation and communication but increase the staleness of the second-order information, so understanding this tradeoff is necessary to optimally apply K-FAC.

We train ResNet-50, ResNet-101, and ResNet-152 with K-FAC for 55 epochs and K-FAC update intervals of $\{100, 500, 1000\}$. The Top-1 validation accuracy from training on 64 V100 GPUs is presented in Table 5 and Figure 6. All update frequencies except 1000 converge to the MLPerf baseline with ResNet-50, so we choose 500 to be the optimal K-FAC update intervals with 64 V100 GPUs for the scaling experiments in §6.4. There are no recognized baselines for ResNet-101 nor ResNet-152 so we use the 76.4% and 76.6% baselines of our prior work for ResNet-101 and ResNet-152, respectively [21]. While we observe a 0.2% validation accuracy drop with K-FAC compared to SGD with ResNet-101 and ResNet-152, our K-FAC results are better than the general baselines.

We repeat a similar study to find a good update interval for BERT. On average, each BERT K-FAC step takes 25 seconds longer than a Fused-LAMB step with 16 A100 GPUs. With 7038 steps in Phase 1, a K-FAC update interval of 100 steps reduces the training time by 1750 seconds compared to an interval of 50 steps. This only reduced training time for Phase 1 by 1.58% (total training time is 111,059 seconds with 16 A100 GPUs) which is not significant. On the other hand, decreasing the K-FAC update interval to 25 steps will introduce an overhead of 3500 seconds, thus we find 50 steps to be an appropriate middle ground for the K-FAC update interval in Phase 1 and Phase 2.

6.4 Scalability

We evaluate the scalability of our distributed K-FAC algorithm on ResNet, Mask R-CNN, and BERT.



(a) ResNet-50 Validation Accuracy (b) Mask R-CNN BBox mAP (c) Mask R-CNN Segm mAP

Fig. 5: ResNet-50 validation accuracy and Mask R-CNN validation bounding box and segmentation mean average precision (mAP) comparison between K-FAC and SGD. ResNet-50 is run on 64 V100 GPUs and Mask R-CNN is run on 32 V100 GPUs.

Phase 1		Phase 2		F1
Opt.	Steps	Opt.	Steps	
LAMB	7038	K-FAC	1563	90.54%
LAMB	7038	K-FAC	1400	90.35%
LAMB	7038	K-FAC	1200	90.18%
LAMB	7038	K-FAC	1000	90.12%

(a) Phase 1 Fused-LAMB Phase 2 K-FAC

Phase 1		Phase 2		F1
Opt.	Steps	Opt.	Steps	
K-FAC	6000	LAMB	1563	90.17%
K-FAC	6000	LAMB	1400	90.26%
K-FAC	6000	LAMB	1200	90.52%
K-FAC	6000	LAMB	1000	89.95%

(b) Phase 1 K-FAC Phase 2 Fused-LAMB

Phase 1		Phase 2		F1
Opt.	Steps	Opt.	Steps	
K-FAC	5000	K-FAC	1563	90.15%
K-FAC	5000	K-FAC	1400	90.25%
K-FAC	5000	K-FAC	1200	89.94%
K-FAC	5000	K-FAC	1000	89.81%

(c) Phase 1 K-FAC Phase 2 K-FAC

TABLE 4: SQuAD v1.1 F1 score with varying numbers of Phase 1 and Phase 2 steps. LAMB is short for Fused-LAMB. indicates the diverged cases.

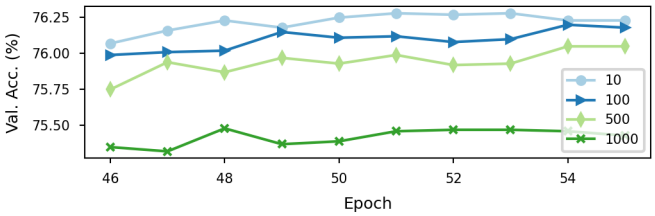


Fig. 6: ResNet-50 validation accuracy of the last 10 epochs with K-FAC update intervals of f10, 100, 500, 1000g iterations. The MLPerf baseline is 75.9%.

TABLE 5: ResNet-50, ResNet-101, and ResNet-152 validation accuracy vs. K-FAC update interval with 64 GPUs [21]

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

6.4.1 ResNet Scalability

We measure the time-to-solution (time to reach the MLPerf baseline validation accuracy) on f16, 32, 64, 128, 256g V100 GPUs. The average time per epoch is measured over 10 epochs and then projected to 55 epochs for K-FAC and 90 epochs for SGD. The K-FAC update interval is scaled inversely to the global batch size to maintain a constant number of K-FAC updates per epoch. In §6.3.4, the ideal interval was determined to be 500 for 64 V100 GPUs, so we use intervals of f2000, 1000, 500, 250, 125g for f16, 32, 64, 128, 256g V100 GPUs, respectively. All other hyperparameters are the same as described in §6.3.1.

We use the basic ResNet training implementation provided by Horovod [27] which contains no additional optimizations for improved training or scaling efficiency. While this means that the end-to-end training time for our SGD baseline is not close to state-of-the-art results for this hardware configuration, we choose a simple implementation to showcase that K-FAC can improve training time and scaling without needing additional complex optimizations. In contrast, Mask R-CNN and BERT training is performed using the optimized NVIDIA reference implementations [52].

Here we compare our distribution strategy, referred to as *K-FAC-opt*, to that used in prior work [6], referred to as *K-FAC-lw*. We use the eigen decomposition K-FAC update procedure (Equations (11)–(13)) for both variants such that this experiment specifically compares how work is distributed among workers and where communication occurs. The *lw* in *K-FAC-lw* stands for the layer-wise distribution scheme [6]. Each worker computes the entire K-FAC update, that is, 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* is our optimized strategy described in §4. The optimized strategy reduces the frequency of communication by decoupling eigen decompositions from gradient preconditioning. All *K-FAC-lw* and *K-FAC-opt* experiments converge to the same 76.2%, 77.7%, and 78.0% validation accuracies for ResNet-50, ResNet-101, and ResNet-152, respectively, within 55 epochs.

Across GPU and model scales, *K-FAC-lw* outperforms SGD by 2.8-19.1%, and *K-FAC-opt* outperforms SGD by 17.7-25.2%, as seen in Figure 8. The scaling efficiency of *K-FAC-opt* is 71.8%, a 9.4% improvement over the 62.4% efficiency of *K-FAC-lw*, and also higher than SGD's 68.6% scaling efficiency. We attribute the better scaling of *K-FAC-opt* to its reduced communication frequency compared to *K-FAC-lw*. While the scaling efficiency at 256 GPUs is below 50% for all three cases, *K-FAC-lw* achieves 2.8% improved performance over SGD whereas *K-FAC-opt* yields an 18.3% improvement.

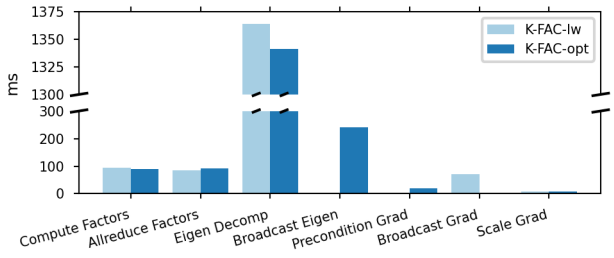


Fig. 7: Average time (ms) for each stage within a call to *KFAC.step()* measured using ResNet-50 on 64 V100 GPUs.

TABLE 6: Iterations required to converge for K-FAC and SGD at different scales for Mask R-CNN.

Optimizer	32 GPUs	64 GPUs	128 GPUs
K-FAC	21 000	12 000	6800
SGD	25 640	15 000	7320

To further understand why *K-FAC-opt* achieves better scaling efficiency than *K-FAC-lw*, we measure the average time spent in each stage of a call to *K-FAC.step()* for ResNet-50 on 64 V100 GPUs. As seen in Figure 7, the factor computation and *allreduce*, eigen decomposition, and preconditioned gradient scaling stages take similar time between the two methods (within 2%). The biggest differences are in the eigen decomposition broadcast and gradient broadcast stages because *K-FAC-lw* does not need to broadcast eigen decompositions and *K-FAC-opt* does not need to broadcast gradients. The eigen decomposition broadcast in *K-FAC-opt* takes 3.4 longer than the gradient broadcast in *K-FAC-lw*; however, eigen decomposition broadcast is only performed every 500 iterations while the gradient broadcast is required every iteration. The effect is that the additional cost to communicate and cache all of the eigen decompositions locally in *K-FAC-lw* yields a valuable reduction in average iteration time for iterations where eigen decompositions are not updated. *K-FAC-opt* also has a longer gradient preconditioning stage because every worker preconditions all gradients locally, but this cost is much lower than the alternative in *K-FAC-lw* where gradient broadcast is required.

6.4.2 Mask R-CNN Scalability

The Mask R-CNN training exploits early stopping, that is, the training automatically stops if the validation metrics reach the baseline target. With a different number of V100 GPUs, the global batch size changes proportionally, and the required steps to converge also varies. Such results can be observed from MLPerf v0.6 [55]. Table 6 summarizes the number of steps to converge for K-FAC and SGD at the scale of 32, 64, and 128 V100 GPUs.

Figure 9a illustrates the training time at each scale for K-FAC and SGD. All three K-FAC cases converge above baseline. With 32 and 64 GPUs, K-FAC takes 14.9% and 18.1% less time than SGD to converge, respectively. That reduction drops to 3.0% with 128 GPUs. Even though the number of steps does not decrease with more GPUs, the scaling efficiency is 74.4%.

6.4.3 BERT Scalability

In the experiments reported in §6.3.3, we use different local batch sizes for BERT to minimize the training time for K-FAC and Fused-LAMB cases. Here, we instead fix the local batch size in order to measure K-FAC's improvement on training time assuming no limitation on memory size. Figure 9 shows the projected training time of both BERT phases across f8, 16, 32, 64, 128g A100 GPUs with a fixed local batch of 60 per GPU for Phase 1 and 12 for Phase 2. With 128 A100 GPUs, the scaling efficiencies are 78.9% for Phase 1 and 78.3% for Phase 2. We do not complete full end-to-end training at scales larger than 16 GPUs due to limited node hours on ThetaGPU. However, as all experiments are trained with a global batch of 64K and 32K for Phase 1 and Phase 2, respectively, we expect similar convergence to 16 GPUs for larger scales. The projected overall training time with K-FAC is 16.1%–16.9% less than with Fused-LAMB.

7 Conclusion

We have presented a distributed K-FAC preconditioner that incorporates a layer-wise distribution scheme to perform K-FAC computations efficiently at scale. We evaluate techniques such as inverse-free second-order gradient preconditioning to maintain convergence across batch sizes and K-FAC approximation update decoupling for reduced time per iteration. We design the preconditioner to be incorporated easily into existing training scripts and implement it in the widely adopted PyTorch framework. Our code is open source and available under the permissive MIT license. We evaluate convergence and scalability empirically with standard DL benchmarks representing a diverse set of model architectures. Our preconditioner enables 18–25% faster convergence to 75.9% MLPerf ResNet-50 baseline validation accuracy on up to 256 NVIDIA V100 GPUs. K-FAC reduces time-to-convergence with Mask R-CNN and BERT by 3.0%–14.9% and 9.0%–16.9%, respectively, while maintaining good scaling on up to 128 NVIDIA A100 GPUs.

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 17-SC-20-SC; and NSF OAC-1931354, OAC-1818253, OAC-2106661, and OAC-2107511.

References

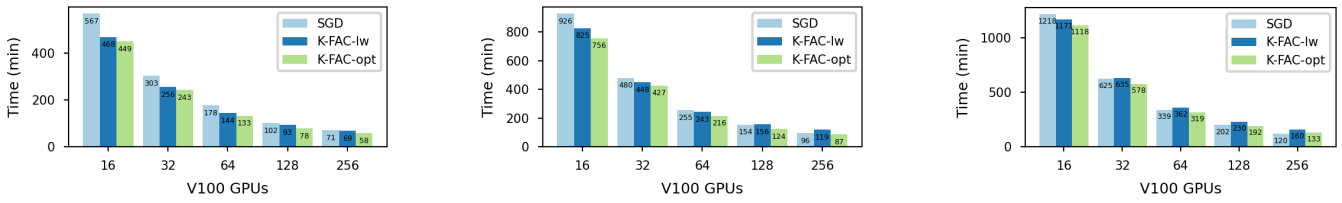
[1] Y. You, Z. Zhang, C.-J. Hsieh, J. Demmel, and K. Keutzer, "ImageNet training in minutes," in *47th International Conference on Parallel Processing*. ACM, 2018, p. 1.

[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] T. Akiba, S. Suzuki, and K. Fukuda, "Extremely large minibatch SGD: Training ResNet-50 on ImageNet in 15 minutes," *arXiv preprint arXiv:1711.04325*, 2017.

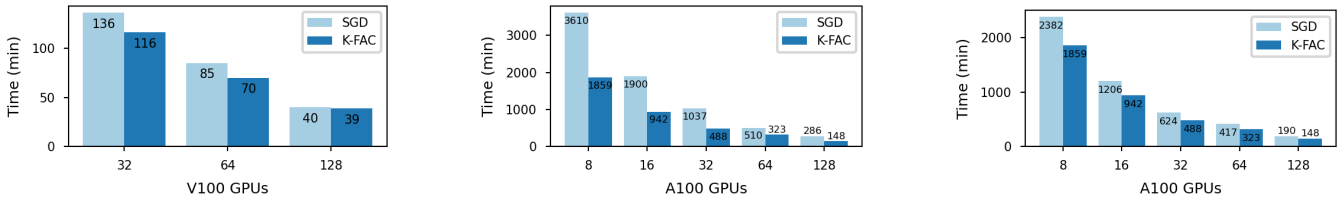
[4] C. Ying, S. Kumar, D. Chen, T. Wang, and Y. Cheng, "Image classification at supercomputer scale," *arXiv preprint arXiv:1811.06992*, 2018.

[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.



(a) ResNet-50 (b) ResNet-101 (c) ResNet-152

Fig. 8: Time-to-solution comparison of ResNet models [21].



(a) Mask R-CNN (b) BERT-Large Phase 1 (c) BERT-Large Phase 2

Fig. 9: Time-to-solution comparison for Mask R-CNN and BERT-Large across scales.

[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*, vol. 13, p. 431–434, 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 the OpenAI Data 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 *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*, 2019, pp. 8024–8035.

[20] K. He, G. Gkioxari, P. Dollár, and R. Girshick, "Mask r-cnn," in *IEEE International Conference on Computer Vision*, 2017, pp. 2961–2969.

[21] J. G. Pauloski, Z. Zhang, L. Huang, W. Xu, and I. T. Foster, "Convolutional neural network training with distributed K-FAC," in *International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2020.

[22] 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 *International Conference for High Performance Computing, Networking, Storage and Analysis*. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3458817.3476152

[23] Y. Huang, Y. Cheng, A. Bapna, O. Firat, M. X. Chen, D. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, and Z. Chen, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," 2019.

[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] Intel, "Intel machine learning scaling library," 2019, https://github.com/intel/MLSL.

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

[28] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation*, 2016.

[29] "NVIDIA Apex (a PyTorch extension)," https://github.com/NVIDIA/apex.

[30] Y. You, I. Gitman, and B. Ginsburg, "Large batch training of convolutional networks," *arXiv preprint arXiv:1708.03888*, 2017.

[31] 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.