THE UNIVERSITY OF CHICAGO

SCALABLE DEEP NEURAL NETWORK TRAINING WITH DISTRIBUTED K-FAC

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

BY
J. GREGORY PAULOSKI

CHICAGO, ILLINOIS
23 MARCH 2022

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would first like to thank my advisors, Dr. Ian Foster and Dr. Kyle Chard, for their guidance, support, and expertise during the development and writing of this research. I have learned a great deal from the both of you—from navigating the paper review cycle to adjusting to life in Chicago.

I extend my thanks and appreciation to Dr. Zhao Zhang not only for serving on the committee for this thesis but for providing me my first opportunities to engage in HPC research, write and review academic papers, attend conferences, and build relationships with others in the field. You were an invaluable source of mentorship as I navigated the graduate school application process, and your advice continues to be extremely formative to me.

I am grateful to my colleagues, past and present, in Globus Labs for their friendship and support, especially when I joined in an entirely online setting. The breadth of research in the lab continues to inspire me.

Finally, I thank my friends and family for their continued encouragement and support.

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 (SGD) and variants thereof. Kronecker-factored Approximate Curvature (K-FAC), a natural gradient method, has recently been shown to converge with fewer iterations in deep neural network (DNN) training than SGD; however, K-FAC's larger memory footprint and increased communication necessitates careful distribution of work for efficient usage. This thesis investigates scalable K-FAC algorithms to understand K-FAC's applicability in large-scale deep neural network training and presents KAISA, a **K**-FAC-enabled, **A**daptable, **I**mproved, and **S**c**A**lable second-order optimizer framework. Specifically, layer-wise distribution strategies, inverse-free second-order gradient evaluation, dynamic K-FAC update decoupling, and more are explored with the goal of preserving convergence while minimizing training time. KAISA can adapt the memory footprint, communication, and computation given specific models and hardware to improve performance and increase scalability, and this adaptable distribution scheme generalizes existing strategies while providing a framework for scaling second-order methods beyond K-FAC. Compared to the original optimizers, KAISA converges 18.1-–36.3% faster across applications with the same global batch size. Under a fixed memory budget, KAISA converges 32.5% and 41.6% faster in ResNet-50 and BERT-Large, respectively. KAISA can balance memory and communication to achieve scaling efficiency equal to or better than the baseline optimizers.

CHAPTER 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 [2, 11, 46, 53, 70, 72]. The intersection of deep learning and HPC has specifically enabled new uses of deep learning for scientific applications [10, 28, 35].

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) [43]. Significant work has been devoted to understanding and improving the scaling properties of SGD [2, 6, 46, 70, 72, 73]. Impressive results have been shown for specific applications such as ResNet-50 [19] and BERT [13]; however, improvements are often made possible via techniques specific to the application or hardware such as custom fused kernels and topology-aware 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, approximates the Fisher Information Matrix (FIM)—an approximation of the Hessian—and preconditions the gradients with the FIM approximation before parameter updates [41]. K-FAC significantly reduces iterations required for convergence and scales well to $O(1000)$ GPUs [53], 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 [49] or exceed the time-to-convergence of SGD.

Improving on SGD's time-to-convergence with K-FAC is difficult due to K-FAC being 1) compute intensive, due to the required inverse or eigen decomposition calculations with $O(N^3)$ complexity (N is the number of parameters); 2) memory intensive because per-layer activations and gradients must be stored, which may take $O(N^2)$ space compared to $O(N)$ for SGD; and 3) communication intensive for distributed training because the Kronecker factors and accompanying inverses or eigen decompositions must be communicated. To overcome these overheads and achieve faster training times than SGD at scale, K-FAC updates are often decoupled from gradient preconditioning so the expensive computations are performed less frequently.

These various considerations make it difficult to use K-FAC efficiently in practice given the specific requirements of the DNN model or hardware and requires users to choose between optimizing memory or communication. This thesis investigates the design and implementation of a K-FAC-based gradient preconditioner that addresses these concerns by reducing time-to-convergence and improving scaling efficiency. The performance, in terms of convergence and time-per-iteration, of an explicit matrix inverse algorithm and implicit eigen decomposition algorithm is evaluated. A conventional method in L-BFGS [40] is applied to decouple the approximation of the FIM from gradient preconditioning which speeds up training time by reducing the frequency at which the FIM is computed, and the convergence impact of the FIM computation frequency is quantified. The training time and model convergence is analyzed for K-FAC specific hyperparameter schedules such as damping and K-FAC update interval decay.

Key to improving scaling with K-FAC is the introduction of an adaptable distribution strategy for second-order optimization. This strategy generalizes existing strategies that optimize for either reduced communication or reduced memory usage using a single parameter, the *gradient worker fraction*. This adaptation is made possible by grouping the workers, then distributing and communicating data within each group. This framework for adaptable

distribution of second-order optimization is called KAISA, a **K**-FAC-enabled, **A**daptable, **I**mproved, and **S**c**A**lable second-order optimizer framework.

KAISA is implemented as a preconditioner in the popular PyTorch framework [55] and evaluated with a suite of reference applications including ResNets [19], Mask R-CNN [20], U-Net [62], and BERT [13] on two clusters with different hardware. The implementation converges to or exceeds the baseline performance metrics in fewer iterations and 9–42% less time across all applications.

This thesis makes the following contributions:

- An adaptive second-order optimizer framework that trains faster than SGD and its variants, while preserving convergence;
- A study of the two 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 four application with different distributed frameworks;
- A quantitative study of the tradeoff between memory footprint and communication and its impact on training time in K-FAC design;
- An open source implementation of the proposed algorithm using PyTorch [55].

The remainder of this thesis is organized as follows. Chapter 2 outlines methods for parallel training and formalizes the K-FAC optimization process. Chapter 3 reviews related work in large-scale training with SGD and K-FAC. Chapter 4 describes the existing strategy for distributed K-FAC, introduces a new and optimized strategy, and then presents a generalized method for distributed second-order optimization. Chapter 5 details the implementation of KAISA. Chapter 6 evaluates KAISA and the various distributed strategies on

a variety of applications. Finally, Chapter 7 draws conclusions and offers remarks on future avenues for exploration.

CHAPTER 2

BACKGROUND

This chapter discusses forms of parallelism in deep learning training, data-parallelism in iterative batch optimization with SGD, and the K-FAC preconditioning method.

2.1 Distributed Training

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

Conventionally, model-parallelism is used in cases where data-parallelism is not possible because model-parallelism can reduce machine utilization and often requires additional optimizations such as pipeline parallelism [22]. As data-parallelism is the dominant form of large scale DL training, numerous frameworks provide native support via NCCL, Gloo [23], or MPI collective operations [65] such as Intel MLSL [24], Horovod [63], TensorFlow [1], PyTorch [55], and NVIDIA Apex [51].

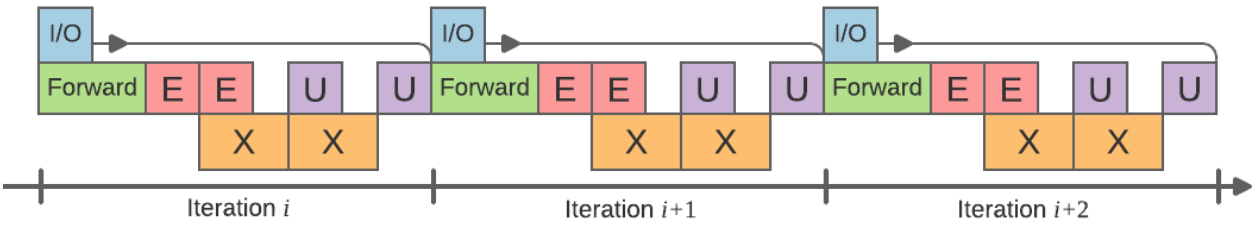


Figure 2.1: Synchronous SGD iteration stages. I/O: Batch creation. Forward: Forward pass and loss computation. E: Gradient evaluation (backpropagation). X: Gradient exchange. U: Trainable variable update.

2.2 SGD

In SGD, a batch optimization method [6], 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 [71] and asynchronous [27, 36, 47, 60, 76], and are distinguished by whether all variables are updated each iteration. This thesis focuses on synchronous SGD methods because asynchronous SGD has a non-linear slowdown compared with the synchronous form [3].

An iteration of synchronous data-parallel SGD has five stages, depicted in Figure 2.1. 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 worker. 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 workers to exchange the gradients. In the final stage, each worker 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 approximation of the Fisher Information Matrix (FIM), which has been shown to be equivalent to the Generalized Gauss-Newton (GGN) matrix in specific cases and can be viewed as an approximation of the Hessian [41].

In the weight update equation for SGD (Equation 2.1), 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 gradient with respect to the loss over each i^{th} example in the mini-batch, denoted as $\nabla L_i(w^{(k)})$.

$$w^{(k+1)} = w^{(k)} - \frac{\alpha^{(k)}}{n} \sum_{i=1}^{n} \nabla L_i(w^{(k)}) \tag{2.1}$$

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

$$w^{(k+1)} = w^{(k)} - \frac{\alpha^{(k)} F^{-1}(w^{(k)})}{n} \sum_{i=1}^{n} \nabla L_i(w^{(k)}) \tag{2.2}$$

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.

It has been shown empirically that training deep neural networks with K-FAC enables convergence with fewer iterations than with SGD alone. Theoretical understandings of the convergence rates of natural gradient methods, such as K-FAC, are an area of active research. Previous work has shown that K-FAC [75] has linear convergence to the global minimum given a sufficiently over-parameterized model. In strongly-convex problems, natural gradient

methods have a quadratic convergence rate compared to the linear convergence of SGD [6]. The strongly-convex case provides some understanding of how K-FAC can improve convergence in non-convex cases. Further, it has been shown that natural gradient methods enable larger learning rates improving the rate of convergence [75]. K-FAC makes greater per-iteration progress in minimizing the objective function at the cost of more computationally expensive iterations.

2.3.1 K-FAC Approximation

K-FAC is based on the Kronecker product, a block matrix factorization that can reduce a large matrix inverse into two smaller matrix inverses. K-FAC exploits the properties of the Kronecker product and the geometry of the FIM for DNNs to greatly reduce the complexity of computing the approximate FIM inverse.

The Kronecker product is written as $A \otimes B$ where A has size $m \times n$ and B has size $p \times q$. The resulting matrix has shape $mp \times nq$.

$$A \otimes B = \begin{bmatrix} a_{11}B & \dots & a_{1n}B \\ \vdots & \ddots & \vdots \\ a_{m1}B & \dots & a_{mn}B \end{bmatrix} \tag{2.3}$$

The Kronecker product has two convenient properties:

$$(A \otimes B)^{-1} = A^{-1} \otimes B^{-1} \tag{2.4}$$

$$(A \otimes B)\vec{c} = B^{\top}\vec{c}A. \tag{2.5}$$

The FIM for a deep neural network is a block matrix where each block maps to layers in

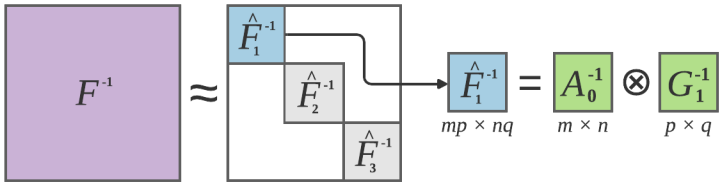


Figure 2.2: The K-FAC approximation of the FIM. $\otimes$ symbolizes the Kronecker product.

the model. The block corresponding to the i and j^{th} layers is approximated as

$$F_{i,j} \approx a_{i-1}a_{j-1}^{\top} \otimes g_i g_j^{\top}. \tag{2.6}$$

Here, a_{i-1} and g_i are the activations of the $i-1^{\text{th}}$ layer and the gradients of the i^{th} layer in the model, respectively [41, 56].[1]

A fundamental assumption of K-FAC is the independence between layers [41, 56].[2] Using this assumption, K-FAC approximates the FIM as a diagonal block matrix $\hat{F}$.

$$\hat{F} = \texttt{diag}(\hat{F}_1, ..., \hat{F}_i, ..., \hat{F}_L) \tag{2.7}$$

As shown in Figure 2.2, the inverse of $\hat{F}$ is a diagonal block matrix composed of the inverses of each diagonal block $\hat{F}_i$:

$$\hat{F}^{-1} = \texttt{diag}(\hat{F}_1^{-1}, ..., \hat{F}_i^{-1}, ..., \hat{F}_L^{-1}) \tag{2.8}$$

where

$$\hat{F}_i = a_{i-1}a_{i-1}^{\top} \otimes g_i g_i^{\top} = A_{i-1} \otimes G_i. \tag{2.9}$$

1. Formally, F represents an expected value, and the expectation of a Kronecker product is not equivalent to the Kronecker product of the expected factors. However, this approximation still reasonably represents the structure of the FIM [41].

2. This assumption is sufficient to produce an effective approximation for F and necessary to produce a tractable algorithm.

A_{i-1} and G_i are referred to as the Kronecker factors. In practice, A_{i-1} and G_i are estimated with a running average of the factors computed over training batches [41, 56]. E.g.,

$$A_{i-1}^{(k)} = \xi A_{i-1}^{(k)} + (1-\xi)A_{i-1}^{(k-1)} \tag{2.10}$$

$$G_i^{(k)} = \xi G_i^{(k)} + (1-\xi)G_i^{(k-1)} \tag{2.11}$$

where ξ is the running average hyper-parameter typically in the range $[0.9, 1)$.

Due to the layer-wise independence of K-FAC, the gradient preconditioning and weight update for a single layer i at iteration k can be written as:

$$w_i^{(k+1)} = w_i^{(k)} - \alpha^{(k)} \hat{F}_i^{-1} \nabla L_i(w_i^{(k)}). \tag{2.12}$$

Applying properties 2.4 and 2.5 to the gradient preconditioning, $\hat{F}_i^{-1}\nabla L_i(w_i^{(k)})$, reduces computation to an efficient form where the smaller Kronecker factors, rather than the large FIM, are inverted.

$$\hat{F}_i^{-1}\nabla L_i(w_i^{(k)}) = (A_{i-1} \otimes G_i)^{-1}\nabla L_i(w_i^{(k)}) \tag{2.13}$$

$$= (A_{i-1}^{-1} \otimes G_i^{-1})\nabla L_i(w_i^{(k)}) \tag{2.14}$$

$$= G_i^{-1}\nabla L_i(w_i^{(k)})A_{i-1}^{-1} \tag{2.15}$$

Tikhonov regularization is used to avoid ill-conditioned matrix inverses with K-FAC by adding a *damping parameter* γ to the diagonal of $\hat{F}_i$ [18, 53]. In most implementations, instead of computing ${\hat{F}_i}^{-1}$, $(\hat{F}_i + \gamma I)^{-1}$ is computed as:

$$(\hat{F}_i + \gamma I)^{-1} = (A_{i-1} + \gamma I)^{-1} \otimes (G_i + \gamma I)^{-1}. \tag{2.16}$$

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

$$w_i^{(k+1)} = w_i^{(k)} - \alpha^{(k)}(\hat{F}_i + \gamma I)^{-1}\nabla L_i(w_i^{(k)}) \tag{2.17}$$

$$= w_i^{(k)} - \alpha^{(k)}(G_i + \gamma I)^{-1}\nabla L_i(w_i^{(k)})(A_{i-1} + \gamma I)^{-1}. \tag{2.18}$$

2.3.2 Infrequent K-FAC Updates

A common strategy in second-order optimization methods is to update the second-order information (i.e., factors and their inverses) every few iterations [41, 53, 56]. Intuitively, second-order information does not change as rapidly from one iteration to the next like first-order information. The K-FAC update interval parameter controls the number of iterations between second-order updates, i.e., iterations between recomputing the second-order information. Larger K-FAC update intervals result in more stale information but reduce average iteration time. Infrequent K-FAC updates are key to reducing communication in the distributed strategy introduced in Section 4.2.2, and the tradeoff between stale information and reduced iteration time is evaluated in Section 6.4.

CHAPTER 3

RELATED WORK

While data-parallelism is key to scaling deep learning training, data-parallelism necessitates scaling the batch size by the number of workers. Larger batch sizes have been shown to produce models which do not generalize as well, and this inhibits practical training of deep neural networks on hundreds or thousands of workers [30]. 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 worker count. Finally, the trend towards training larger models necessitates careful usage of limited worker memory and advanced hybrid-parallel schemes that maintain high hardware utilization.

KAISA is an advanced optimization framework that aims to address many of these aforementioned issues. Hence, KAISA builds on many prior efforts to scale out deep learning training and is designed to work in conjunction with a variety of techniques. This chapter summarizes prior work in scalable deep learning and discusses the relationships between KAISA and other existing methods.

3.1 Optimization Frameworks

Distributed optimization frameworks take many forms. In a synchronous optimizer, such as Horovod's `DistributedOptimizer` wrapper [63], all parameters are updated in every iteration. Asynchronous optimizers relax parameter update consistency, for example by passing values via parameter servers [36], to achieve higher performance than synchronous methods [38, 60]. The pipeline parallel paradigm, such as GPipe [22] and PipeDream [50] which hold multiple versions of a model partition in a worker and exploit asynchronous optimizers, has shown comparable training convergence. Generally, asynchronous SGD has a non-linear slowdown compared to synchronous SGD [3], so KAISA implements K-FAC in a

synchronous manner as a preconditioner designed to work with any synchronous optimizer.

BytePS [26] proposes a unified interface for synchronous and asynchronous SGD and leverages heterogeneous computing resources by offloading computations from GPUs on to the CPU. ZeRO, the zero-redundancy optimizer, leverages CPU offloading and improves memory efficiency compared to pipeline parallel methods by partitioning model states across CPUs and GPUs to avoid memory redundancies inherent to data-parallel training [57]. ZeRO-Infinity extends memory offloading to high-speed NVMe storage in order to train models with tens of trillions of parameters [58]. KAISA's configurable distribution strategy for second-order computations enables either improved memory efficiency by reducing memory redundancies or improved communication efficiency by increasing memory redundancies when there is sufficient spare worker memory.

3.2 Memory Shortage and Remedies

Offloading is a common method at the optimizer level for reducing the memory intensity of deep learning training; however, swapping between GPU memory and host memory, such as in SuperNeurons [68] and SwapAdvisor [21], is another viable method for training larger models. SuperNeurons modifies the GPU memory scheduling runtime to make optimal allocation and swap decisions, and SwapAdvisior uses a genetic algorithm to search possible memory allocations and swapping schedules to find a pre-planned schedule that best overlaps computations with communication. Subsequently, SuperNeurons can handle dynamic graph executions while SwapAdvisor requires static graph execution. KAISA uses PyTorch's dynamic execution and is therefore not compatible with SwapAdvisor.

An alternative to swapping out allocated memory is to discard some activation tensors and rematerialize them when needed for back-propagation. Checkmate [25] formulates this tradeoff as an optimization problem and provides an optimal rematerialization schedule. KAISA saves activations during the forward pass to later be used in the second-order com-

putations which negates most of the benefits of rematerialization methods.

Recent support for half-precision computation on GPUs has enabled lower memory usage and faster computation through automatic mixed-precision (AMP) training [44]. AMP performs computations in the faster half-precision FP16 format where possible, but maintains higher-precision for more numerically unstable operations. To avoid numerical instability when computing gradients, AMP scales the loss appropriately to ensure gradient magnitudes are within the supported range of the half-precision format. Many popular frameworks such as NVIDIA Apex [51] and PyTorch [55] offer AMP support, and AMP training is now a de facto standard for high-performance training. KAISA has native support for PyTorch AMP and can store data in lower precisions when necessary to reduce memory consumption.

3.3 Generalization Gap

Researchers have observed that training with large-batches exacerbates the generalization gap—the difference in a model's performance on training data and testing data [30]. 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 [17], and these two techniques enabled ResNet-50 training without harming convergence with batch sizes up to 8192. However, this still limits training to modest numbers of GPUs because ResNet-50 is commonly trained with per-GPU batch sizes of 128 or 256. Later work, LARS [72] and LAMB [73], 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 [72, 73]. 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 [70]. Many of these techniques, such as mixed-precision, interconnect-aware *allreduce*, and distributed batch normalization, are application or hardware specific, so this thesis focuses on a general optimization algorithm that is hardware, network topology, and application agnostic. Consequently, KAISA can be combined with most of these methods. For example, evaluations with BERT-Large in Chapter 6 use KAISA as a preconditioner to LAMB.

3.4 Distributed K-FAC

K-FAC can reduce iterations-to-convergence in image classification tasks with ResNets [4, 53] and natural language tasks with RNNs [42]. The gradient preconditioning in K-FAC, described in Section 2.3.1, results in much slower iterations than SGD, so prior work uses an asynchronous K-FAC distribution scheme with a doubly-factored Kronecker approximation to achieve iteration times on par with SGD for ImageNet-1k training [4]. 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, KAISA uses 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 [53]. They show 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, the work does not show K-FAC to be faster than, nor achieve the same validation accuracy as, SGD.

Later work from the same group addresses these limitations and reaches the MLPerf baseline in 10.5 minutes on 512 GPUs [66] without the use of a stale FIM approximation. A novel 21-bit floating point specification is introduced to reduce communication, and the

baseline SGD code is carefully analyzed and optimized to improve scaling. In contrast, this thesis shows that the MLPerf baseline can be reached with stale FIM approximations and that, in practice, the FP16 and FP32 formats provided by standard machine learning frameworks are sufficient for exceeding the performance of SGD. KAISA is the first K-FAC implementation to reach MLPerf acceptance baselines in less wall-time than SGD, a set of criteria not met by prior works 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 [54]. 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. KAISA addresses 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.

3.5 Alternative K-FAC Methods

Eigenvalue-corrected Kronecker-factorization (EK-FAC) [14], a more accurate approximation of the FIM, can perform cheap, partial updates. Noisy K-FAC [74] and noisy EK-FAC [5] are functionally similar to standard K-FAC but introduce adaptive weight noise. K-BFGS and K-BFGS(L) [16] apply BFGS [7] and L-BFGS [39] in an analogous method to K-FAC (e.g., block-diagonal, Kronecker-factored). These works have shown better-than-K-FAC performance in many small scale (e.g., single GPU) and small dataset/model (e.g., MNIST or CIFAR-10 with VGG16) cases. Kronecker-factor based FIM approximation variants are a growing area of research; however, large-scale studies have largely been limited to standard K-FAC. KAISA introduces a valuable, unified design paradigm that can be applied to these K-FAC variants to efficiently deploy and evaluate their effectiveness on large models at scale.

CHAPTER 4

OPTIMIZING DISTRIBUTED K-FAC DESIGN

This chapter discusses how KAISA addresses limitations in prior works by using an alternative K-FAC approximation and introducing new strategies for distributing second-order computations. These unique KAISA designs are key to achieving performance that exceeds that of SGD.

4.1 Alternative K-FAC Approximation

The largest computational cost in K-FAC is computing the second-order information needed to precondition the gradients. Typically, this second-order information is the inverses of $(A_{i-1} + \gamma I)$ and $(G_i + \gamma I)$ (Equation 2.16), and all existing distributed K-FAC implementations use this method [4, 53, 66]. Grosse and Martens [18] notes that an expansion of the eigen decomposition of $(\hat{F}_i + \gamma I)^{-1}$ can be used to obtain an exact solution compared to a direct inversion of the factors. Given Q_A and Q_G, the eigenvectors of the factors, and υ_A and υ_G, the eigenvalues of the factors, the preconditioned gradient can be computed as follows.

$$V_1 = Q_G^\top \nabla L(w_i^{(k)}) Q_A \tag{4.1}$$

$$V_2 = V_1 / (\upsilon_G \upsilon_A^\top + \gamma) \tag{4.2}$$

$$(\hat{F}_i + \gamma I)^{-1} \nabla L(w_i^{(k)}) = Q_G V_2 Q_A^\top \tag{4.3}$$

The composition of the factors A and G in Equation 2.9 guarantees the factors are symmetric and therefore the factor eigen decompostions have real eigenvalues and orthogonal eigenvectors.

Table 4.1 compares the explicit inverse preconditioning method with the implicit eigen decomposition method. The final validation accuracy for ResNet-32 trained on CIFAR-10

Table 4.1: Validation accuracy with inverse and eigen decomposition K-FAC methods for ResNet-32 trained on CIFAR-10.

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%

is consistently better as a function of batch size in the eigen decomposition method, and the eigen decomposition method performs as well as or better than SGD for all batch sizes. While the implicit decomposition method requires a few more matrix multiplications per layer compared to the explicit inverse method, the empirically more stable solution of the implicit method addresses the generalization gap issues between K-FAC and SGD observed in prior works (Section 3.4).

The remainder of this thesis uses the implicit eigen decomposition method for all experiments; however, in general discussions of distributed second-order methods, *second-order information* will be used to ambiguously refer to the inverses or eigen decompositions of the factors and *second-order computations* refers to the computation of the second-order information using the factors. The choice of preconditioning method does not effect the distribution of second-order computations, and these distribution methods can be extended to many of the alternative preconditioning methods described in Section 3.5.

4.2 Hybrid Parallel K-FAC

Existing distributed K-FAC implementations are hybrid-parallel, with first-order information (e.g., gradients) computed in data-parallel and second-order information (e.g., inverses of the factors) in model-parallel, as described in Figure 4.1. Each iteration starts with standard data-parallel training. The model is replicated across all workers and a random local batch of data is assigned to each worker. The forward pass, loss, and backwards pass are computed in parallel across all workers, and then the gradients computed for each layer in the backwards

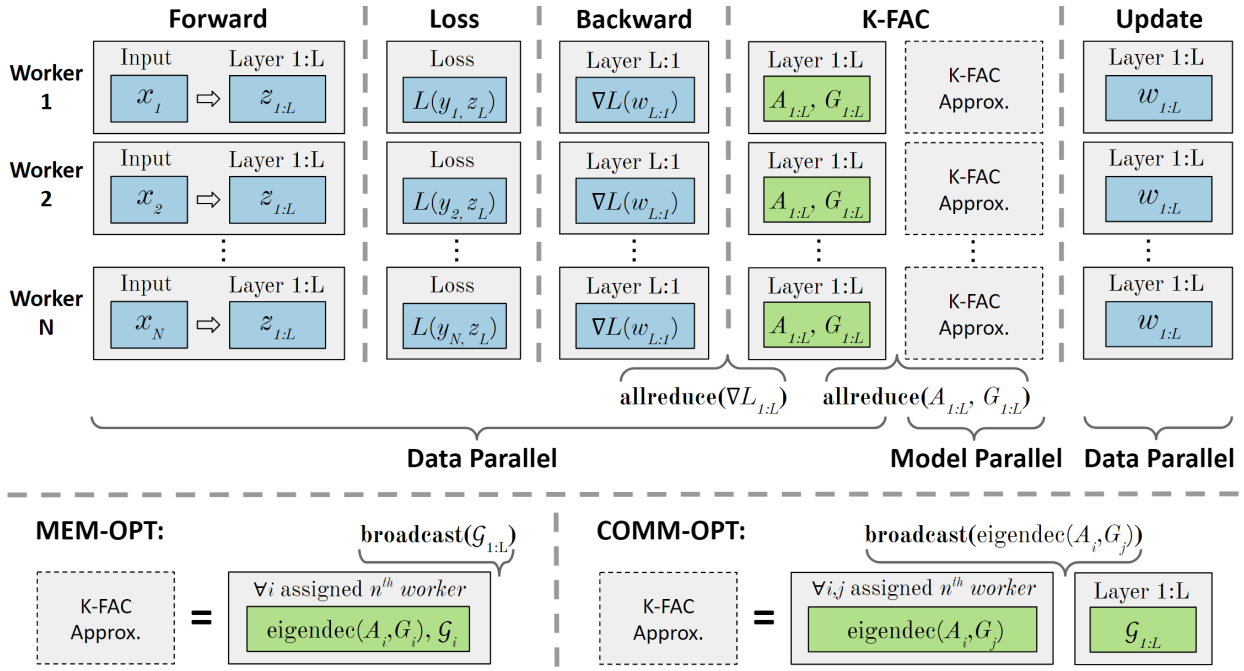


Figure 4.1: Hybrid-parallel distributed K-FAC implementation overview. Blue boxes are standard computations in data-parallel training. Green boxes are computations required by K-FAC. Workers maintain identical copies of the model. The output of each layer z is computed during the forward pass for the local batch x. Then, the loss between the true output y and predicted output z_L is calculated and used to compute gradients in the backward pass. Gradients are averaged across workers with *allreduce*. During the forward/backward pass, K-FAC caches intermediate data for computing factors. In the K-FAC stage, workers compute factors A and G in a data-parallel fashion and *allreduce* the results. The second-order information (such as the eigen decompositions) and preconditioned gradients $\mathcal{G}$ are computed in the K-FAC approximation stage. After the K-FAC stage, all workers have $\mathcal{G}$ and can update weights locally using $\mathcal{G}$ and a standard optimizer (e.g., SGD or Adam).

pass are averaged across all of the workers.

After the gradient communication, the K-FAC preconditioning stage begins. The K-FAC preconditioning stage has three steps for each layer: compute the factors, compute the second-order information using the factors, and precondition the gradients using the second-order information. The factors A and G are computed using intermediate values saved during the forward and backward passes, so after workers compute A and G for each layer locally in a data-parallel fashion, the factors are averaged across workers. Once the factors are computed and communicated, the second-order information can be computed in a model-parallel fashion and the gradients can be preconditioned. Figure 4.1 deliberately leaves this stage ambiguous as the model-parallelism can be implemented in various ways.

At the end of the K-FAC stage, every worker has a local copy of the preconditioned gradients. Then, each worker can update its local copy of the weights using the preconditioned gradients and an optimizer of choice (e.g., SGD).

The remainder of this chapter discusses how prior works implement the model-parallel K-FAC approximation stage, the limitations inherent in these previous designs, and a series of improvements designed to address these prior limitations.

4.2.1 A Memory Efficient Approach

Osawa et al. [53] introduced the first hybrid-parallel distributed K-FAC scheme which is referred to as MEM-OPT (memory optimized). MEM-OPT takes a strong layer-wise approach to the model-parallel K-FAC approximation stage. After the factors A and G have been computed and communicated, each layer in the model is assigned to a different worker. Each worker computes the second-order information for the layers assigned to itself. Then the worker will precondition the gradients using the second-order information for the layers. Once all workers have computed the second-order information and preconditioned the gradients for all of the layers in a model-parallel fashion, the workers share the resulting pre-

conditioned gradients with each other (generally using an *allgather* or a series of *broadcast* operations).

This method maintains a minimal memory footprint because second-order information is not duplicated: each layer's second-order information is stored on only one worker. MEM-OPT has communication in three places: a) gradient *allreduce*, b) factor *allreduce*, and c) preconditioned gradient communication, all shown in Figure 4.1. In non-K-FAC update steps (e.g., steps where the second-order information is not updated), MEM-OPT avoids the factor *allreduce* because the second-order information from a previous step is reused; however, the preconditioned gradient communication is still required because the gradient being preconditioned changes every iteration. This marks a key limitation of this approach: the amount of gradient communication required is invariant to the frequency at which the K-FAC approximations are recomputed. A second limitation of this approach is that workers will be left idle during the K-FAC approximation stage if there are more workers than layers in the model.

4.2.2 Reducing Communication

To remedy the two limitations of MEM-OPT, Pauloski et al. [56] introduces a new method for the model-parallel K-FAC approximation stage, referred to as COMM-OPT (communication optimized). COMM-OPT decouples the second-order information computation from gradient preconditioning. Instead of assigning each layer to a worker, individual second-order computations are assigned to workers to be performed in parallel. The second-order information is broadcast back to all workers such that every worker holds a copy of *all* second-order information for *all* layers. This allows each worker to compute the preconditioned gradients locally.

This method has a larger memory-footprint because every worker caches all second-order information. COMM-OPT has communication in three places: a) gradient *allreduce*, b)

factor *allreduce*, and c) second-order information *broadcast*, also shown in Figure 4.1. Decoupling the second-order calculations from the gradient preconditioning achieves two goals: 1) the second-order calculations for A_{i-1} and G_i can be computed on different workers, doubling the maximum worker utilization over MEM-OPT, and 2) in non-K-FAC update steps, the communication in (b) and (c) can be avoided because every worker can locally precondition gradients with the cached second-order information. Thus, in non-K-FAC update steps, COMM-OPT has no additional communication overhead compared to SGD. In other words, the communication required by K-FAC is proportional to the K-FAC update frequency.

4.3 Generalizing Second-Order Hybrid Parallelism

COMM-OPT and MEM-OPT convey the fundamental tradeoff between caching second-order information locally to avoid communication and communicating the preconditioned gradients every iteration to avoid additional memory overheads. This tradeoff impacts the communication, computation, and memory overhead of K-FAC and motivates a more flexible design for second-order optimization.

KAISA introduces HYBRID-OPT, a configurable distributed K-FAC strategy that can exploit tradeoffs between memory usage and communication overhead.

In this strategy, each layer has a subset of the workers assigned to be gradient preconditioners, referred to as the gradient workers. The *gradient worker fraction*, abbreviated as *grad-worker-frac*, defines the size of this subset, i.e., the gradient worker count is $\texttt{max}(1, \textit{grad-worker-frac} \times \textit{world-size})$. One or two of the gradient workers are also responsible for computing the second-order information for the layer and broadcasting the results to the remaining gradient workers (two if the second-order information for each of the two factors are computed on separate workers). At the end of the second-order computation stage, each gradient worker has a copy of the second-order information and can precondition

Algorithm 1: Optimization with K-FAC pseudocode at iteration k.

```
// Compute Gradients
Compute loss in forward pass with local batch
Compute gradients ∇L_{1:L}(w_{1:L}) in backward pass
allreduce(∇L_{1:L}(w_{1:L}))

// Step 1: Compute Factors
if k (mod factor-update-interval) then
    foreach layer i do
        A_{i-1} = a_{i-1} a_{i-1}^⊤                                   // Equation 2.9
        G_i = g_i g_i^⊤                                               // Equation 2.9
        A_{i-1}^{(k)} = ξ A_{i-1}^{(k)} + (1 − ξ) A_{i-1}^{(k-1)}     // Equation 2.10
        G_i^{(k)} = ξ G_i^{(k)} + (1 − ξ) G_i^{(k-1)}                 // Equation 2.11
        allreduce(A_{i-1}^{(k)}, G_i^{(k)})
    end
end

// Step 2: Compute Eigen Decompositions
if k (mod kfac-update-interval) then
    foreach layer i do
        if second-order worker for A_{i-1} then
            Q_{A_{i-1}}, υ_{A_{i-1}} = eigendecompose(A_{i-1})
            broadcast(Q_{A_{i-1}}, υ_{A_{i-1}})
        else if gradient worker for layer i then
            receive(Q_{A_{i-1}}, υ_{A_{i-1}})
        end
        if second-order worker for G_i then
            Q_{G_i}, υ_{G_i} = eigendecompose(G_i)
            broadcast(Q_{G_i}, υ_{G_i})
        else if gradient worker for layer i then
            receive(Q_{G_i}, υ_{G_i})
        end
    end
end

// Step 3: Precondition Gradients
foreach layer i do
    if gradient worker for layer i then
        G_i = precondition(∇L_i(w_i))                                 // Equation 4.1—4.3
        broadcast(G_i)
    else
        receive(G_i)
    end
end

// Update Weights with SGD
foreach layer i do
    Update weights using preconditioned gradients
end
```

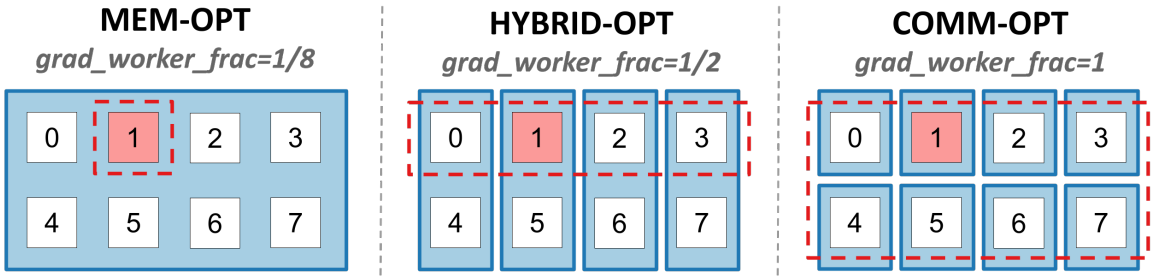


Figure 4.2: A comparison of three *grad-worker-frac* values on eight workers. Process 1, shaded in red, computes the second-order information for this layer and communicates the result to all gradient workers, the workers inside the dashed red box. Gradient workers compute and broadcast the preconditioned gradient to a subset of the remaining workers, denoted by the light blue box. In MEM-OPT, there is one gradient worker that broadcasts the preconditioned gradient to all workers. In HYBRID-OPT, there are four gradient workers that send the preconditioned gradient to one of the four remaining workers (e.g., worker 0 sends the preconditioned gradient to worker 4). In COMM-OPT, the preconditioned gradient need not be communicated because all workers are gradient workers.

the gradient.

Gradient receivers are the workers not assigned as gradient workers for the layer. Each gradient worker is responsible for broadcasting the preconditioned gradient to a subset of the gradient receivers. With multiple gradient workers, the broadcast groups are smaller and simultaneous broadcasts are possible, reducing overall communication time. For example, given two gradient workers and two gradient receivers, each gradient worker sends the preconditioned gradients to just one receiver; both gradient workers perform this communication at the same time.

After the gradient broadcast, all workers have a copy of the preconditioned gradient with which local weights can be updated. Pseudocode for this process is provided in Algorithm 1.

Observe that this strategy unifies existing distributed K-FAC strategies because COMM-OPT and MEM-OPT are special cases of HYBRID-OPT. COMM-OPT is the case where *grad-worker-frac* $= 1$ (all workers are gradient workers for every layer), and MEM-OPT is the case where *grad-worker-frac* $= 1/$*world-size* (all workers are gradient receivers except for a single gradient worker which preconditions a layer's gradient and broadcasts the re-

sult). Figure 4.2 compares MEM-OPT, COMM-OPT, and HYBRID-OPT in an eight worker environment.

As the *grad-worker-frac* increases, more workers cache the second-order information and the memory footprint increases. Visually, the number of workers that cache the second-order information is represented by the workers in the dashed red box in Figure 4.2.

Continuing with Figure 4.2, HYBRID-OPT has a lower preconditioned gradient broadcast cost than MEM-OPT because the broadcast groups are smaller and broadcasts are overlapped. HYBRID-OPT requires four separate broadcasts to groups of size two in comparison to MEM-OPT which requires one large broadcast to a group of eight. Since these broadcasts involve non-overlapping workers, all four broadcast calls can be executed simultaneously in the HYBRID-OPT case. Given the complexity of *broadcast* using the minimum spanning tree algorithm is $O(\log p)$, where p is the number of workers in the broadcast group, the complexity is reduced from $O(\log 8)$ in MEM-OPT to $O(\log 2)$ in HYBRID-OPT. Note in steps where the second-order information is updated, HYBRID-OPT incurs an additional second-order information broadcast with $O(\log 4)$ complexity; however, as mentioned in §2.3.2, second-order information is updated infrequently so the average complexity for HYBRID-OPT is still less than that of MEM-OPT.

The problem addressed by KAISA's HYBRID-OPT strategy is akin to that of 2.5D matrix multiplication [64]. Both algorithms can utilize extra worker memory to reduce communication costs by controlling the replication factor of data. In 2.5D matrix multiplication, a parameter c determines the number of data copies, and in KAISA, the *grad-worker-frac* determines the number of workers that store the second-order information.

4.4 Work Assignments

Distributing the second-order computations is key because these computations are typically either matrix inversions or eigen decomposition which are expensive to compute. To most

efficiently use available resources, the second-order computations should be distributed in a manner that minimizes the makespan T, the time it takes for all workers to complete their assigned computations. The longest processing time greedy algorithm produces an assignment with makespan $T \leq \frac{3}{2}T^*$ where T^* is the optimal makespan [32]. This algorithm works by sorting jobs by decreasing length and then iteratively assigning each job to the worker with the lowest current work load.

The processing time for matrix inversion and eigen decomposition can be approximated as $O(N^3)$ where each matrix is $N \times N$ [12]. KAISA approximates the cost of all second-order computations at the start of training and then uses the longest processing time strategy to assign each computation to a worker. Alternatively, memory usage can be optimized for by using $O(N^2)$ as the approximation since each matrix has N^2 elements.

CHAPTER 5

IMPLEMENTATION

5.1 Interface and Usage

KAISA implements K-FAC as a preconditioner to standard optimizers with support for Conv2d and Linear layers. KAISA has an easy-to-use interface and can be incorporated into existing training scripts in two lines: one to initialize and one to call *KFAC.step()* prior to *optimizer.step()* (see Listing 5.1). In the initialization, hooks are registered to all Conv2D and Linear modules for saving the necessary intermediate data, and the second-order computations are distributed across workers using the longest processing time greedy algorithm and the *grad-worker-frac*.

A call to *KFAC.step()* 1) computes the factors using the forward/backward pass data, 2) computes the second-order information in parallel and broadcasts the results, 3) computes the preconditioned gradients and broadcasts the results if necessary, and 4) scales the preconditioned gradients. KAISA defaults to using the implicit eigen decomposition preconditioning method but also provides support for the explicit inverse method.

```
model = DistributedDataParallel(model)
optimizer = optim.SGD(model.parameters(), ...)
preconditioner = KFAC(model, grad_worker_frac=0.5, ...)

for data, target in train_loader:
    optimizer.zero_grad()
    output = model(data)
    loss = criterion(output, target)
    loss.backward()

    preconditioner.step()
    optimizer.step()
```

Listing 5.1: Example K-FAC usage.

5.2 Application Support

Because KAISA is designed to work seamlessly with data-parallel training scripts, KAISA works with the distributed data-parallel model wrappers provided by PyTorch [55], NVIDIA Apex [51], and DeepSpeed [59]. Usage with NVIDIA Apex and DeepSpeed is similar to the example with PyTorch in Listing 5.1. KAISA provides additional features to support a wider-variety of training environments.

5.2.1 Mixed Precision Training

Mixed-precision training is becoming increasingly important for training larger models as hardware support for half-precision operations has improved [45]. KAISA supports storing and communicating the factors and second-order information in single-precision (FP32) or half-precision (FP16) to work with training scripts that use PyTorch's native AMP implementations.

When training with AMP, factors are stored in half-precision, reducing memory costs. K-FAC computations are performed in half-precision where possible. Inverses or eigen decompositions are generally unstable in half-precision so factors are cast to single-precision before decomposition. KAISA can store second order information in half-precision to further reduce memory consumption if needed. With PyTorch AMP, the *GradScaler* object responsible for scaling and unscaling the gradient in the backward pass can be passed to KAISA. KAISA uses the *GradScaler* to correctly unscale the intermediate data used to compute the G factors, since the scale factor can change from iteration to iteration, causing problems when computing the running average of G over the course of training.

Half-precision training has become a staple in achieving state-of-the-art results in deep learning, and KAISA can particularly benefit from half-precision training due to the K-FAC computation and communication overhead.

5.2.2 Gradient Accumulation

Worker memory (e.g., GPU VRAM) limits the maximum per-worker batch size during training. A common strategy to achieve larger effective batch sizes is gradient accumulation, a method where gradient values for multiple forward and backward passes are accumulated between optimization steps. For example, if worker memory limits the batch size to 8, an effective batch size of 32 can be achieved by accumulating gradients over four forward/backward passes prior to the optimization step. This strategy is common in applications such as BERT, where effective batch sizes are often $> 2^{15}$, but modern GPUs are limited to local batch sizes $< 2^{7}$.

The forward/backward pass data needed by KAISA to compute the factors is accumulated over each mini-batch between calls to *KFAC.step()*. As the number of gradient accumulation steps increases, the memory needed to accumulate the forward/backward pass data grows linearly. KAISA can efficiently support gradient accumulation by computing the factors for the current mini-batch during the forward/backward pass instead of accumulating and delaying the computation until the next *KFAC.step()* call. The factor communication is still performed during *KFAC.step()*.

5.3 Hyper-Parameters

KAISA has hyper-parameters for the factor update interval, damping, and gradient scaling.

The *kfac-update-interval* parameter controls the number of iterations between second-order information updates and communication. Pauloski et al. [56] found that the running average of the factors can be updated and communicated at a frequency 10× that of the frequency of the second-order information without affecting the convergence. As mentioned previously, the factors become more stable throughout training so at fixed iterations, the *kfac-update-interval* can be increased by a scalar factor to reduce the computation and communication over the course of training. Small improvements can be gained via advanced

kfac-update-interval schedules, but Pauloski et al. [56] found that a constant value was generally sufficient in terms of performance for the entirety of training.

Similarly, a fixed damping decay schedule is used such that larger damping values early account for rapid changes in the FIM at the start of training [53].

The preconditioned gradient $\mathcal{G}_i$ for a layer is scaled by ν to prevent the norm of $\mathcal{G}_i$ becoming large compared to w_i [53], where ν is computed as

$$\nu = \texttt{min}\left(1, \sqrt{\frac{\kappa}{\alpha^2 \sum_{i=1}^{n} |\mathcal{G}_i^\top \nabla L_i(w_i)|}}\right) \tag{5.1}$$

using a user-defined constant κ [15, 67]. Typically, values for κ on the order of 10^{-3} are sufficient.

5.4 Communication

For KAISA specific communication operations, KAISA uses the PyTorch interfaces to collective communication operations. The asynchronous variants of the operations are preferred to overlap communication with computation.

5.4.1 Triangular Factor Communication

Previous K-FAC work has exploited the symmetric nature of the Kronecker factors to reduce communication volume by sending only the upper triangle for each factor [54, 66]. KAISA supports extracting the upper triangle for the factor *allreduce* and reconstructing the full factor before the second-order computation stage. For the models studied in this thesis, this optimization did not yield performance improvements for two reasons. First, network latency impacted overall communication time more than bandwidth; second, this optimization has an additional overhead for extracting the upper triangle and reconstructing the factor. For models with larger individual layers—and therefore factors—this optimization could yield

greater benefits.

5.4.2 Gradient Preconditioning

The largest overhead of K-FAC in non-K-FAC update steps is computing the preconditioned gradients. This process, described by Equations 4.1–4.3 in the implicit eigen decomposition method, involves a series of matrix additions, divisions, and multiplications. Observe that the gradient $\nabla L_i(w_i^{(k)})$ is the only variable that changes between non-K-FAC update iterations. In particular, the computation involving the outer product of the eigenvalues, $1/(\upsilon_G \upsilon_A^\top + \gamma)$, in Equation 4.2 does not need to be recomputed every iteration—only after the eigen decompositions are updated. Also observe that when $\textit{grad-worker-frac} > 1/\textit{world} - \textit{size}$, multiple workers perform this computation redundantly.

To reduce the total number of operations during the preconditioning stage, the computation of the outer product is moved into the eigen decomposition stage. The worker assigned to eigen decompose G computes $1/(\upsilon_G \upsilon_A^\top + \gamma)$ and broadcasts the result to all gradient workers instead of broadcasting υ_A and υ_G. This ensures that $1/(\upsilon_G \upsilon_A^\top + \gamma)$ is computed once (on a single worker) and then reused many times by other workers. In practice, this reduced the time to precondition the gradients for a single layer by up to 53%. Note that there is no similar optimization when using the explicit inverse preconditioning method; however, as mentioned previously, the evaluations here do not use the explicit method.

5.5 Math Libraries

Matrix eigen decomposition and inversion is performed on the GPU. KAISA supports PyTorch 1.6 and later but the underlying libraries used for eigen decomposition and inversion depend on the PyTorch version. In PyTorch 1.7 and older, *torch.symeig()* is used 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 eigen decomposition algorithms optimized for symmetric matrices can be used. For the analysis in §4.1, CUDA 10.0 and PyTorch 1.6 is used 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.

CHAPTER 6

EVALUATION

6.1 Applications

KAISA is evaluated on the ResNet model family, a U-Net, Mask R-CNN, and BERT to understand the convergence characteristics of KAISA and to compare KAISA's performance with the original optimizers. These applications cover three typical deep neural network usage domains: image classification, object detection, and language modeling.

6.1.1 Image Classification

Correctness, i.e., that KAISA enables convergence to validation metrics on-par with the default first-order optimizers for the application, is first tested with ResNet-34 [19] and the CIFAR-10 [33] dataset. Then, ResNet-50 [19], trained on the ImageNet-1k dataset [34], is used to evaluate KAISA's convergence and performance at larger scales. The ResNet training script is based on the Horovod reference ResNet training script [63] and modified to use PyTorch's native `DistributedDataParallel` model wrapper. Unless otherwise specified, training and communication is done in single-precision. By default, the script uses SGD with momentum (abbreviated to SGD for the remainder of this section) for optimization, and when training with KAISA, K-FAC is used is to precondition all convolutional and linear layers prior to using SGD for the weight updates. For ResNet-34, the acceptable performance target is 92.49% validation accuracy [19], and for ResNet-50, the MLPerf [49] acceptance performance of 75.9% validation accuracy is used as the baseline.

6.1.2 Object Detection

Two object detection tasks are explored. The first application is the NVIDIA reference PyTorch implementation of Mask R-CNN [20, 52] with the Common Objects in Context

(COCO) 2014 dataset [37]. NVIDIA Apex's `DistributedDataParallel` model wrapper is used to enable data-parallel training. SGD is the default optimizer. KAISA is used to precondition the convolutional and linear layers in the region of interest (ROI) heads of Mask R-CNN. The MLPerf benchmark target results [49] of 0.377 bounding-box mean average precision (bbox mAP) and 0.342 segmentation mean average precision (segm mAP) are adopted as the baseline.

The second application is a U-Net [62] architecture for segmenting brain tumor sub-regions. A Kaggle competition implementation [8] is extended to enable multi-GPU training with PyTorch's native `DistributedDataParallel` model wrapper. The model is trained on the LGG Segmentation dataset [9], which contains Magnetic Resonance (MR) images of the brain from 110 patients across five hospitals. Images from a random subset of 100 patients are used as the training dataset and the remaining 10 patients are used for validation. Adam [31] is the default optimizer, and KAISA is applied to all convolutional layers in the model. The baseline performance target is a 91.0% validation Dice similarity coefficient (DSC) [8]. Training for both object detection applications is done in single-precision.

6.1.3 Language Modeling

To evaluate KAISA on language modeling tasks, BERT-Large uncased is trained using a modified version of the NVIDIA reference PyTorch implementation for BERT [13, 52]. The model is trained on the English Wikipedia [69] and Toronto BookCorpus datasets [77]. Fused-LAMB is used as the optimizer [73]. Each transformer in BERT is implemented using a series of linear layers, so KAISA is applied to all linear layers. KAISA is not used to precondition the embedding layer and prediction head because both of these layers have a Kronecker factor with shape *vocab-size* $\times$ *vocab-size*, and since the vocab size for BERT-Large is 30K, these factors cannot be efficiently inverted or eigen decomposed. The strategy outlined in §5.2.2 is used to reduce memory consumption since gradient accumulation is used for training.

PyTorch's `DistributedDataParallel` is used for data-parallel training, and PyTorch AMP is used to enable half-precision training.

A common practice to evaluate BERT convergence is to fine-tune the pre-trained 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% [52], 91.0% [61], and 90.4% [73] for fine-tuning on the SQuAD v1.1 dataset. Due to the partial unavailability of the Toronto BookCorpus training dataset, the baseline implementation trained with Fused-LAMB only converges to 90.8%.

6.2 Platforms

Two GPU clusters are used for evaluation. 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. For training on Longhorn, PyTorch 1.6, CUDA 10.0, CUDNN 7.6.4, and NCCL 2.4.7 is used.

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). Training on ThetaGPU is performed with PyTorch 1.9, CUDA 11.3, CUDNN 8.2.0, and NCCL 2.9.9.

These two systems are distinguished between in the following text by specifying either V100 or A100 GPUs.

6.3 Convergence

The performance of KAISA is evaluated against the default optimizers in two scenarios: when both cases use the same batch-size and when the maximum possible batch size al-

Table 6.1: Baseline validation set performance and hardware summary for ResNet-34, ResNet-50, Mask R-CNN, U-Net, and BERT-Large. "acc." is accuracy, "mAP" is mean average precision, and "DSC" is Dice similarity coefficient.

App	Ref	Baseline	GPU	# GPUs
ResNet-34	[19]	92.49% top-1 acc.	V100	1
ResNet-50	[49]	75.9% top-1 acc.	V100	64
			A100	8
Mask R-CNN	[49]	0.377 bbox mAP, 0.342 segm mAP	V100	32
				64
U-Net	[8]	91.0% DSC	A100	4
BERT-Large	[52]	90.8% SQuAD v1.1 F1 score	A100	8

lowable by the GPU memory is used. "Optimizing with KAISA" is used as shorthand for preconditioning the gradients with KAISA prior to using the default optimizers for variable updates.

6.3.1 Fixed Batch Size

KAISA's performance (converged accuracy, epochs to convergence, and time to convergence) is examined with ResNet-34, ResNet-50, Mask R-CNN, U-Net, and BERT-Large. The applications, their baseline targets, and training environments (i.e., type and count of GPUs) are summarized in Table 6.1.

In this section, the same global batch size is used with and without KAISA to isolate the improvement from second-order information. Table 6.2 summarizes the hyperparameters for each application.

For ResNet-34, the learning rate is $N \times 0.1$ and the batch size is $N \times 128$ where N is the number of GPUs. The 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. ResNet-34 is trained for 100 epochs with K-FAC and twice as long with SGD. K-FAC eigen decompositions are recomputed every 10 iterations. For ResNet-50, the learning rate is

Table 6.2: Summary of hyperparameters used for each application. BS = global batch size, LR = learning rate, WU = warm up iterations, K-int = number of iterations between eigen decomposition re-computations, F-int = iterations between factor updates. *grad-worker-frac* = 1 and $\gamma = 0.003$ for all cases.

App	BS	LR	WU	K-int	F-int
ResNet-34	256	0.2	250	10	1
ResNet-50	2,048	0.8	3,130	500	50
Mask R-CNN	64	8e-2	800	500	50
U-Net	64	4e-4	500	200	20
BERT-Large	65,536	5e-5	103	100	10

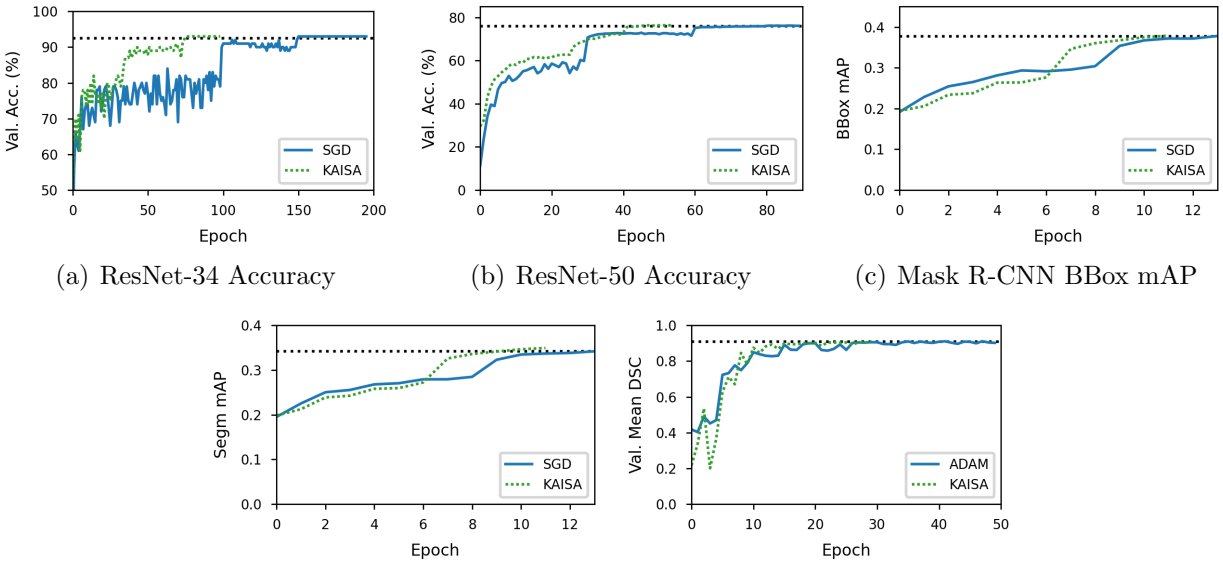
$N \times 0.0125$ and the batch size is $N \times 32$. The learning rate is linearly warmed up for the first five epochs then decayed at epochs 25, 35, 40, 45, and 50. Labels are smoothed with a factor of 0.1. The SGD momentum is 0.9 for ResNet-34 and ResNet-50.

With Mask R-CNN and BERT-Large, the NVIDIA reference hyperparameters [52] are used. Further performance improvements could be gained through more extensive hyperparameter tuning; however, this is not necessary to achieve results better than the original optimizers with KAISA.

ResNet-34: Training ResNet-34 on the Cifar-10 dataset with KAISA reduces the epochs to convergence in half, from 150 to 75 epochs, as shown in Figure 6.1(a). However, since training is performed on a single GPU, the time-to-convergence is worse with KAISA as no data- or model-parallelism in the K-FAC stage can be taken advantage of.

ResNet-50: The second experiment compares K-FAC and SGD training with ResNet-50 and the ImageNet-1k dataset. The goal is to 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.

ResNet-50 is trained for 55 and 90 epochs for KAISA and SGD, respectively, using half-precision on eight NVIDIA A100 GPUs. Figure 6.1(b) shows the validation accuracy curve using the two optimization methods. KAISA converges to the baseline validation accuracy at epoch 46 and SGD at epoch 65. The time-to-convergence is 268.1 minutes for KAISA:



(a) ResNet-34 Accuracy (b) ResNet-50 Accuracy (c) Mask R-CNN BBox mAP

(d) Mask R-CNN Segm mAP (e) U-Net Dice Score

Figure 6.1: Validation Metric Curve Comparison between KAISA and SGD/Adam for ResNet-34, ResNet-50, Mask R-CNN, and U-Net. The dotted lines represent the target metric.

24.3% less than the 354.0 minutes for SGD.

Mask R-CNN: The baseline measurement of Mask R-CNN on 32 NVIDIA V100 GPUs takes 25,640 iterations to converge to 0.377 bbox mAP and 0.342 segm mAP in single-precision. With K-FAC enabled for the ROI heads, training converges to 0.379 bbox mAP and 0.350 segm mAP in 21,000 iterations. The bbox mAP and segm mAP comparisions for SGD and KAISA are shown in Figures 6.1(c) and 6.1(d). KAISA reduces the training time from 136.1 minutes to 115.8 minutes, a 14.9% improvement. With a batch size of 128 on 64 V100 GPUs, KAISA converges in 12,000 iterations compared to 15,000 with SGD and reduces training time by 18.1%.

U-Net: The reference U-Net implementation [8] with Adam converges to 91.0% validation DSC within 50 epochs with four NVIDIA A100 GPUs using single-precision training. Using KAISA, the training converges above 91.0% in 30 epochs as seen in Figure 6.1(e). KAISA reduces the training time by 25.4% (10.9 minutes vs. 14.6 minutes).

BERT: BERT pre-training has two phases. The first trains with maximum sequence

Table 6.3: BERT performance comparison: KAISA vs. LAMB

		KAISA, with iterations:		
Metric	LAMB	1200	1000	800
SQuAD F1	90.8	91.0	91.0	90.8
SQuAD F1 Stdev.	0.13	0.15	0.17	0.24
Iterations	1,536	1,200	1,000	800
Time (hours)	47.7	41.5	34.4	30.4

length of 128 for 7038 iterations and the second trains with maximum sequence length of 512 for 1,563 iterations. The pre-trained BERT model is fine-tuned for three epochs using the SQuAD dataset. All phases and fine-tuning are done in half-precision. Tuning the hyperparameters of BERT is expensive as each run can take over 130 hours with 8 A100 GPUs, so the effectiveness of KAISA is shown with the second phase of BERT pre-training. Phase two training begins with the same model pre-trained with LAMB during phase one. Training with KAISA is performed for {800, 1,000, 1,200} iterations then fine-tuned on SQuAD. Table 6.3 summarizes the validation SQuAD F1 scores after fine-tuning and the pre-training performance improvements.

KAISA converges to the 90.8 F1 baseline in 800 iterations, 47.9% less iterations than required for LAMB, and KAISA takes 36.3% less time than LAMB to converge.

6.3.2 Fixed Memory Budget

To understand how the gradient worker fraction can enable second-order optimization in memory-constrained environments, KAISA's performance is compared against baselines with fixed memory budgets. In particular, ResNet-50 is trained with 64 V100 GPUs and BERT-Large phase two is trained with eight A100 GPUs. For each experiment, the maximum possible local batch size is used and the epochs to convergence and time to convergence are measured. Table 6.4 summarizes the hyperparameters and time to convergence.

ResNet-50: With SGD, the maximum local batch size is 128 per GPU and the global

Table 6.4: Time to convergence and hyperparameters for each optimizer. BS in the global batch size, LR in the learning rate, g-frac in the gradient worker fraction, K-f is the iterations between eigen decomposition re-computations, F-f is the iterations between factor updates, and T-conv is time to convergence in minutes.

App	Optimizer	BS	LR	g-frac	K-f	F-f	T-conv
ResNet-50	SGD	8K	3.2	—	—	—	N/A
	KAISA	5K	2.0	1/64	200	20	96
	KAISA	5K	2.0	1/2	200	20	83
BERT-Large	LAMB	24K	3e-3	—	—	—	2,917.6
	KAISA	32K	4e-3	1/2	100	10	1,702.5
	KAISA	32K	4e-3	1	100	10	1,703.5

batch size is 8,192. ResNet-50 when trained for 90 epochs using SGD achieves 75.0% validation accuracy—0.9% lower than the MLPerf baseline. KAISA with *grad-worker-frac* = 1 and a local batch size of 80 exceeds the local memory of the GPU. Lowering *grad-worker-frac* to 1/2, training takes 83 minutes to converge to 75.9% in the 48th epoch. The complete 55 epoch training takes 95 minutes and the validation accuracy reaches 76.0%. So, even if SGD converges to 75.9% by the 90th epoch, KAISA still reduces the time to convergence by 32.5%. Finally, the same training process is repeated with MEM-OPT by setting the *grad-worker-frac* to 1/64. This configuration converges at the 47th epoch and the time to convergence is 96 minutes. The complete 55 epoch training takes 111 minutes. This experiment highlights the benefit of KAISA in cases where compute resource can not be efficiently utilized with the original optimizers. With a *grad-worker-frac* value of 1/2, KAISA offers benefits over COMM-OPT and MEM-OPT by enabling second-order optimization under a tight memory budget that is 13.9% faster than COMM-OPT and not feasible with MEM-OPT.

BERT: With LAMB, the maximum possible local batch size per GPU is 12 for the second phase of this BERT implementation [52], and the global batch size is 24,576. For KAISA, a local batch size of 8 and global batch size of 32,768 is achievable. Note that the accumulation steps parameter is kept constant. This experiment uses the same hyperparameters as in §6.3.1, so all cases with LAMB and BERT should converge to the baseline, thus the training

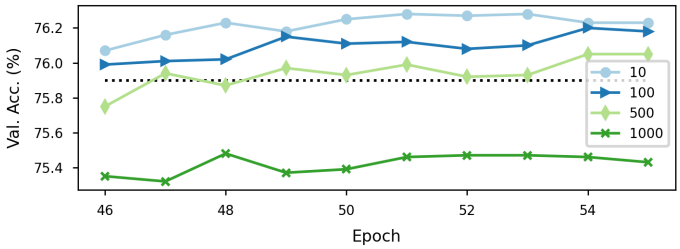


Figure 6.2: ResNet-50 validation accuracy of the last 10 epochs with K-FAC update intervals of {10, 100, 500, 1000} iterations. The MLPerf baseline of 75.9% is represented with the dotted black line.

Table 6.5: ResNet-50, ResNet-101, and ResNet-152 validation accuracy vs. K-FAC update interval with 64 GPUs.

Model		SGD	K-FAC Update Interval 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

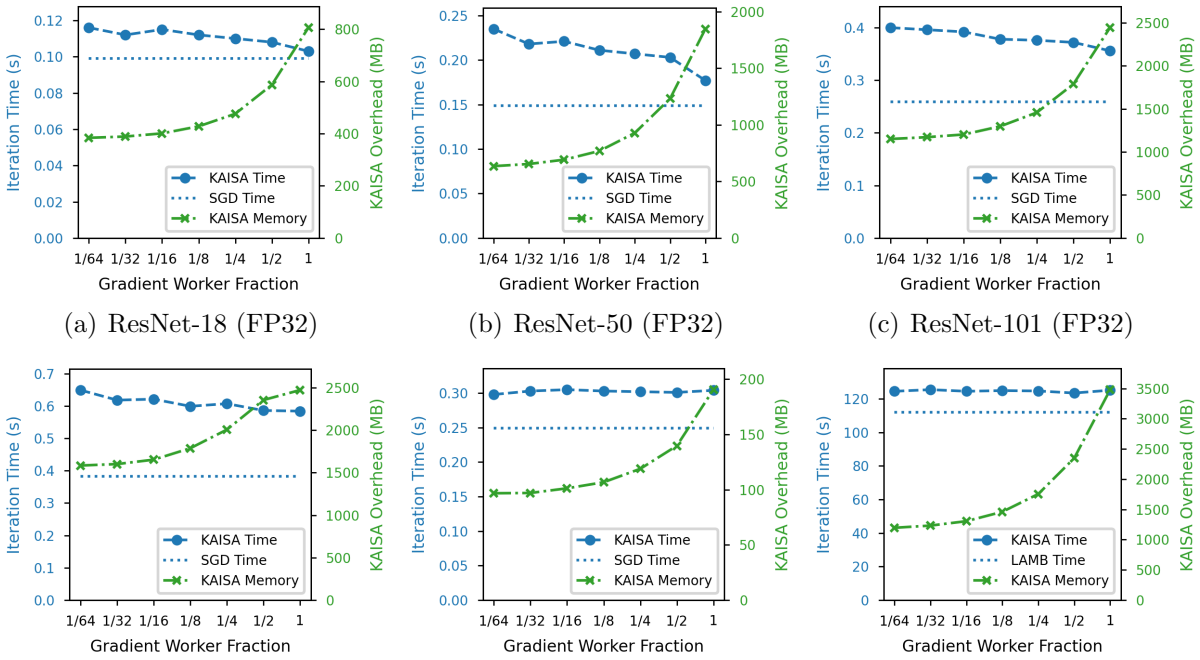
time is projected using the time measured for the first 100 steps. As the global batch size with LAMB is 24,576, 2,084 training steps are required to finish three epochs, and the training time is 2,917.6 minutes. KAISA, with *grad-worker-frac* = 1/2, takes 3,268.8 minutes to finish the three epochs. However, KAISA converges to baseline after 800 steps, as shown in Table 6.3, so the time to converge is 1,702.5 minutes—41.6% faster than LAMB. Setting *grad-worker-frac* = 1 takes 1,703.5 minutes to converge for KAISA. The performance is comparable to the case with *grad-worker-frac* = 1/2. A detailed study on this convergence phenomena is conducted for different gradient worker fraction values in §6.5.

6.4 K-FAC Update Frequency

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.

ResNet-50, ResNet-101, and ResNet-152 are trained 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 6.5 and Figure 6.2. All update intervals except 1000 converge to the MLPerf baseline with ResNet-50, so 500 is chosen to be the optimal K-FAC update intervals with 64 V100 GPUs for the scaling experiments in §6.6.1. There are no recognized baselines for ResNet-101 nor ResNet-152 so the 76.4% and 76.6% baselines, respectively, reported by Keras Application [29] are used. While a 0.2% validation accuracy decrease with KAISA is observed compared to SGD with ResNet-101 and ResNet-152, the KAISA results are better than the general baselines.

A similar study is repeated 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 reduces 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 50 steps is chosen to be an appropriate middle ground for the K-FAC update interval in phase 1 and phase 2.



(a) ResNet-18 (FP32) (b) ResNet-50 (FP32) (c) ResNet-101 (FP32)

(d) ResNet-152 (FP32) (e) Mask R-CNN (FP32) (f) BERT-Large Phase 2 (FP16)

Figure 6.3: Average iteration time and K-FAC memory overhead across *grad-worker-frac* values on 64 V100 GPUs. Dotted lines are the baseline iteration times without K-FAC.

6.5 Memory vs. Communication

To understand the impact of *grad-worker-frac* on training times, ResNet-{18, 50, 101, 152}, Mask R-CNN, and BERT-Large are trained with 64 V100 GPUs for *grad-worker-frac* $\in$ {1/64, 1/32, 1/16, 1/8, 1/4, 1/2, 1}. For each experiment, the average iteration time, i.e., time between weight updates, and the GPU memory usage, is recorded. The *K-FAC overhead* is the memory required to store the factors and eigen decompositions, and the K-FAC overhead is computed as the difference between the K-FAC and SGD memory usage. For all ResNet models, the same hyperparameters are used for ResNet-50 (§6.3.1) except for ResNet-152 where the local batch size was lowered to 24. The results are presented in Figure 6.3, and a summary of the memory usage is provided in Table 6.6.

The K-FAC memory overhead increases linearly as a function of *grad-worker-frac* for all models. KAISA requires 1.5–45.8% more memory than SGD depending on the application

Table 6.6: Summary of per-GPU memory usage for training, in MB. Abs. is the absolute memory required for training. Δ is the %-increase in memory required over SGD. The K-FAC overhead is the K-FAC abs. memory minus the SGD abs. memory.

		SGD	K-FAC Min		K-FAC Max	
Model	Precision	Abs.	Abs.	Δ	Abs.	Δ
ResNet-18	FP32	2454	2838	16.7%	3260	32.8%
ResNet-50	FP32	4762	5396	13.3%	6608	38.8%
ResNet-101	FP32	6313	7463	18.2%	8755	38.7%
ResNet-152	FP32	6620	8204	23.9%	9092	37.3%
Mask R-CNN	FP32	6553	6650	1.5%	6743	2.9%
BERT-Large	FP16	8254	9555	15.8%	12038	45.8%

and value of *grad-worker-frac* (Table 6.6). The maximum K-FAC overhead (i.e., when *grad-worker-frac* = 1) is 1.5–2.9× that of the minimum K-FAC overhead (i.e., when *grad-worker-frac* = 1/64).

With respect to iteration times, the ResNet models scale well with the number of gradient workers with ResNet-50 scaling the best. For ResNet-50, the speedup from a gradient worker count of 1 to 64 is 24.4% with a 22% increase in total memory usage. The average iteration times for Mask R-CNN and BERT-Large remain constant as the number of gradient workers is increased. These promising results for ResNet-50, a de facto standard benchmark for deep learning systems, are important as the performance characteristics of ResNet-50 represent a large set of commonly used models (e.g. VGG16, U-Nets, etc.).

To understand why ResNet model performance varies across *grad-worker-frac* values while the Mask R-CNN and BERT-Large performance remains constant, consider the bandwidth requirements of these applications. The bandwidth required by KAISA is a function of the size of the factors, eigen decompositions, and frequency of K-FAC updates.

With 64 V100s, ResNet-50 calls *KFAC.step()* frequently (4–6 calls/second) and incurs a K-FAC memory overhead between 634 MB and 1.8 GB. In comparison, Mask R-CNN calls *KFAC.step()* with a lower frequency (3 calls/second) and has a much smaller K-FAC overhead (100–200 MB). Thus, the changes in how KAISA communicates data with respect to

grad-worker-frac are less apparent in Mask R-CNN. BERT-Large has the lowest bandwidth requirements of all applications even though it has the largest K-FAC overhead. BERT-Large uses gradient accumulation to achieve very large batch sizes (32K for phase 2) and as a result only calls *KFAC.step()* every ∼120 seconds. These K-FAC overheads are summarized in Table 6.6.

While the iteration times for low-communication models such as BERT and Mask R-CNN are invariant to the *grad-worker-frac* value in KAISA, KAISA still produces faster-than-SGD training times with small increases in memory-overhead. This is due to KAISA's unique features outlined in §4 and §5. Further, practitioners training these models at larger scales, e.g., 100s or 1000s of GPUs, where communication becomes a greater bottleneck will benefit more from the flexibility KAISA provides to adapt training to environments with increasing communication costs.

Tuning the *grad-worker-frac* hyperparameter, to determine an optimal balance between iteration time and memory usage, is simple as it only requires profiling the average iteration time for each *grad-worker-frac* value over a few iterations.

With respect to training times, *grad-worker-frac* has the most impact in applications that spend a larger proportion of time doing communication. To further understand how the *grad-worker-frac* impacts training times, the execution time for each section within *KFAC.step()* with ResNet-50 on 64 V100s is analyzed. Figure 6.4 provides the time spent in each section for all layers in the model during a call to *KFAC.step()*. Times are averaged over 10,000 iterations and across all workers. Eigen decompositions are updated every 500 iterations. As shown in Figure 6.4, factor computation and communication, eigen decomposition, and scaling and updating the gradients, are invariant to the *grad-worker-frac*.

The time required to broadcast the eigen decompositions increases substantially as the number of gradient workers is increased. Referring back to Figure 4.2, the more gradient workers there are, the more workers that need to receive the eigen decompositions. However,

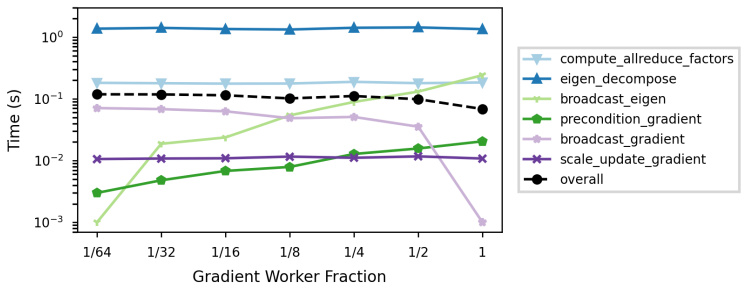


Figure 6.4: Average function execution time during calls to *KFAC.step()* for ResNet-50 on 64 GPUs.

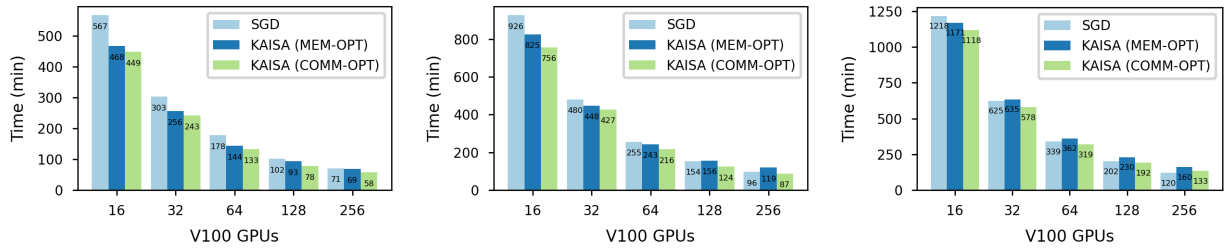
the eigen decompositions, in this case, are only recomputed every 500 iterations so the eigen decomposition broadcast has a negligible effect on the average iteration time.

On the other hand, the gradient preconditioning and broadcast occur every iteration regardless of if the factors or eigen decompositons are updated and have the greatest influence on average iteration time. The time to precondition the gradients increases with the gradient worker count because each worker is assigned as a gradient worker for more layers. This discovery highlights the importance of the gradient preconditioning optimizations made in §5.2.2. The time to broadcast the preconditioned gradients decreases to 0 as the gradient worker count approaches *world-size*, and notably, the time decreases at a *faster rate* than the increase in time required in the preconditioning stage. This trend is a result of each gradient worker needing to send the results to fewer other workers as the gradient worker count increases.

6.6 Scaling

6.6.1 ResNet Scalability

The time to solution (time to reach the MLPerf baseline validation accuracy) is measured on {16, 32, 64, 128, 256} 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



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

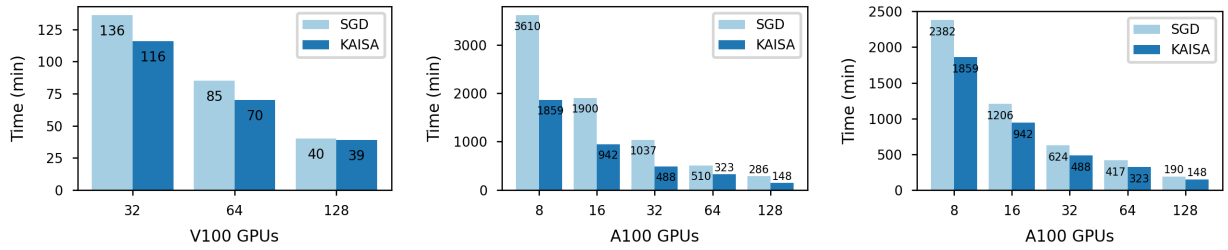
Figure 6.5: Time-to-solution comparison of ResNet models.

interval is scaled inversely to the global batch size to maintain a constant number of K-FAC updates per epoch. In §6.4, the ideal interval was determined to be 500 for 64 V100 GPUs, so intervals of {2000, 1000, 500, 250, 125} for {16, 32, 64, 128, 256} V100 GPUs are used, respectively. All other hyper-parameters are the same as described in §6.3.1.

The ResNet training implementation is a simple data-parallel implementation which contains no additional optimizations for improved training or scaling efficiency. While this means that the end-to-end training time for the SGD baseline is not state-of-the-art for this hardware configuration, using a simple implementation showcases that K-FAC can improve training time and scaling without needing additional complex optimizations.

Here, the COMM-OPT and MEM-OPT strategies of KAISA are compared to training with SGD. Both KAISA configurations 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, MEM-OPT outperforms SGD by 2.8-19.1%, and COMM-OPT outperforms SGD by 17.7-25.2%, as seen in Figure 6.5. The scaling efficiency of COMM-OPT is 71.8%, a 9.4% improvement over the 62.4% efficiency of MEM-OPT, and also higher than SGD's 68.6% scaling efficiency. The better scaling of COMM-OPT is due to its reduced communication frequency compared to MEM-OPT. While the scaling efficiency at 256 GPUs is below 50% for all three cases, MEM-OPT achieves 2.8% improved performance over SGD whereas COMM-OPT yields an 18.3% improvement.



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

Figure 6.6: Time-to-solution comparison for Mask R-CNN and BERT-Large across scales.

Table 6.7: Iterations required to converge for KAISA and SGD at different scales for Mask R-CNN.

Optimizer	32 GPUs	64 GPUs	128 GPUs
KAISA	21 000	12 000	6800
SGD	25 640	15 000	7320

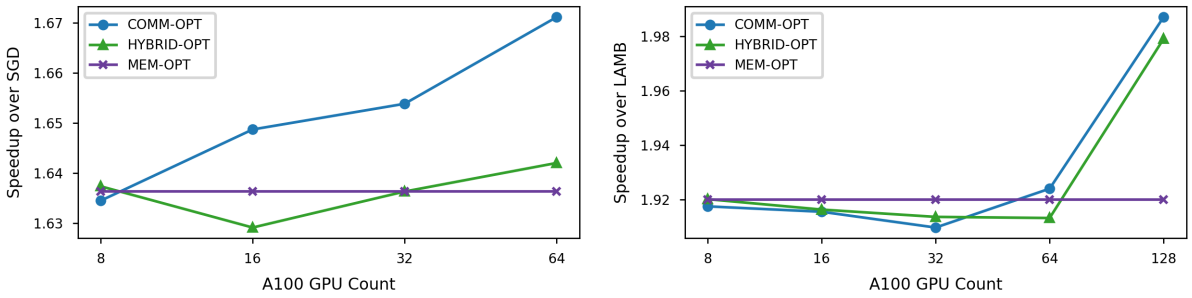
6.6.2 Mask R-CNN Scalability

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 [48]. Table 6.7 summarizes the number of steps to converge for KAISA and SGD at the scale of 32, 64, and 128 V100 GPUs.

Figure 6.6(a) illustrates the training time at each scale for KAISA and SGD. All three KAISA cases converge to above the baseline metrics. With 32 and 64 GPUs, KAISA 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%.

6.6.3 KAISA Speedups over the Baseline

The scaling characteristics of KAISA are examined in more detail with ResNet-50 and the second phase of BERT-Large training. ResNet-50 and BERT-Large are chosen because



(a) ResNet-50 (b) BERT-Large Phase 2

Figure 6.7: Speedup for the ResNet-50 and BERT-Large applications on A100 nodes.

they represent a high-communication and low-communication model, respectively (see §6.5). Three KAISA variants are used: COMM-OPT, MEM-OPT, and HYBRID-OPT with a *grad-worker-frac* of 1/2, and the projected end-to-end training time speedups for the KAISA variants over the base optimizers (SGD and LAMB) are reported in Figure 6.7. For ResNet-50, the training time is projected to 90 epochs for SGD and 55 for KAISA. For BERT-Large, the phase 2 training time is projected to 1,563 steps for LAMB and 800 for KAISA based on the study in §6.3.1. The hyperparameters in Table 6.2 are used, and for ResNet-50, the K-FAC update interval is scaled inversely with the global batch size to keep the number of K-FAC updates per training samples constant.

In Figure 6.7(a) and 6.7(b), MEM-OPT maintains a constant speedup over SGD and LAMB, respectively, across all scales. In contrast, the speedup for COMM-OPT improves in both cases as the scale increases indicating the tradeoff of more memory for reduced communication does have scaling benefits. HYBRID-OPT sees performance improvements on par with COMM-OPT with BERT-Large while using less GPU memory. Overall, HYBRID-OPT has the best balance of scaling and memory usage for large-scale BERT pre-training.

As a whole, KAISA achieves better speedups with BERT-Large which is likely due to ResNet-50 being more communication bound than BERT-Large as noted in §6.5. Further scaling experiments on a machine with more GPUs is needed to understand the characteristics and limits of KAISA's speedup over SGD.

CHAPTER 7

SUMMARY AND FUTURE WORK

This thesis presents KAISA, a **K**-FAC-enabled, **A**daptable, **I**mproved, and **S**c**A**lable second-order optimizer framework. KAISA incorporates an adaptable layer-wise distribution scheme to perform K-FAC computations efficiently at scale. This adaptability enables appropriate distribution of the complex K-FAC computations to best suit the model and hardware characteristics. 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 are evaluated. KAISA is designed to be easily incorporated into existing training scripts and is implemented in the widely adopted PyTorch framework. The code is open source and available under the permissive MIT license at `https://github.com/gpauloski/kfac_pytorch` .

A few directions for future work are apparent. First, extending KAISA to enable distributed second-order optimization with K-FAC variants such as those described in §3.5 could yield better convergence. To reduce the communication inherent to K-FAC, tensor bucketing methods for communication operations or lossless and lossy tensor compression techniques can be incorporated into KAISA.

The fundamental tradeoff between data access on local and remote memory is explored, and KAISA's correctness and impact on training time is evaluated with a suite of real-world applications. With the same global batch size, KAISA provides an 18.1–36.3% training time reduction for ResNet-50, Mask R-CNN, U-Net, and BERT-Large, while preserving convergence to the baseline. Under the same memory budget, ResNet-50 and BERT phase 2 converge to the baseline in 32.5% and 41.6% less time compared to momentum SGD and Fused-LAMB. In high-communication applications, such as ResNet models, extra memory can be used to improve iteration times by reducing communication. In low-communication applications, such as Mask R-CNN and BERT-Large, the optimal KAISA usage is shown

and used to achieve efficient scaling on par with SGD up to 128 GPUs.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation*, 2016.

[2] Takuya Akiba, Shuji Suzuki, and Keisuke Fukuda. Extremely large minibatch SGD: Training ResNet-50 on ImageNet in 15 minutes. *arXiv preprint arXiv:1711.04325*, 2017.

[3] Dan Alistarh, Christopher De Sa, and Nikola Konstantinov. The convergence of stochastic gradient descent in asynchronous shared memory. In *ACM Symposium on Principles of Distributed Computing*, pages 169–178. ACM, 2018.

[4] Jimmy Ba, Roger B. Grosse, and James Martens. Distributed second-order optimization using Kronecker-factored approximations. In *ICLR*, 2017.

[5] Juhan Bae, Guodong Zhang, and Roger B. Grosse. Eigenvalue corrected noisy natural gradient. *CoRR*, abs/1811.12565, 2018. URL `http://arxiv.org/abs/1811.12565`.

[6] Léon Bottou, Frank E Curtis, and Jorge Nocedal. Optimization methods for large-scale machine learning. *SIAM Review*, 60(2):223–311, 2018.

[7] C. G. BROYDEN. The Convergence of a Class of Double-rank Minimization Algorithms 1. General Considerations. *IMA Journal of Applied Mathematics*, 6(1):76–90, 03 1970. ISSN 0272-4960. doi: 10.1093/imamat/6.1.76. URL `https://doi.org/10.1093/imamat/6.1.76`.

[8] Mateusz Buda and Soumith Chintala. Brain-Segmentation-PyTorch, 2019. `https://github.com/mateuszbuda/brain-segmentation-pytorch`.

[9] Mateusz Buda and Soumith Chintala. LGG Segmentation Dataset, 2019. `https://www.kaggle.com/mateuszbuda/lgg-mri-segmentation`.

[10] Juan Carrasquilla and Roger G Melko. Machine learning phases of matter. *Nature Physics*, 13:431–434, 2017.

[11] Valeriu Codreanu, Damian Podareanu, and Vikram 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.

[12] James Demmel, Ioana Dumitriu, and Olga Holtz. Fast linear algebra is stable. *Numerische Mathematik*, 108(1):59–91, Oct 2007. ISSN 0945-3245. doi: 10.1007/s00211-007-0114-x. URL `http://dx.doi.org/10.1007/s00211-007-0114-x`.

[13] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[14] Thomas George, César Laurent, Xavier Bouthillier, Nicolas Ballas, and Pascal Vincent. Fast approximate natural gradient descent in a kronecker factored eigenbasis. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018. URL `https://proceedings.neurips.cc/paper/2018/file/48000647b315f6f00f913caa757a70b3-Paper.pdf`.

[15] Thomas George, César Laurent, Xavier Bouthillier, Nicolas Ballas, and Pascal Vincent. Fast approximate natural gradient descent in a Kronecker-factored eigenbasis. In *32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 9573–9583, Red Hook, NY, USA, 2018. Curran Associates Inc.

[16] Donald Goldfarb, Yi Ren, and Achraf Bahamou. Practical quasi-newton methods for training deep neural networks. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 2386–2396. Curran Associates, Inc., 2020. URL `https://proceedings.neurips.cc/paper/2020/file/192fc044e74dffea144f9ac5dc9f3395-Paper.pdf`.

[17] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch SGD: Training ImageNet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

[18] Roger Grosse and James Martens. A Kronecker-factored approximate Fisher matrix for convolution layers. *arXiv e-prints*, art. arXiv:1602.01407, February 2016.

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

[20] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask r-cnn. In *IEEE International Conference on Computer Vision*, pages 2961–2969, 2017.

[21] Chien-Chin Huang, Gu Jin, and Jinyang Li. SwapAdvisor: Pushing deep learning beyond the GPU memory limit via smart swapping. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1341–1355, 2020.

[22] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism, 2019.

[23] Facebook Incubator. Gloo: Collective communications library with various primitives for multi-machine training, 2017. `https://github.com/facebookincubator/gloo`.

[24] Intel. Intel Machine Learning Scaling Library, 2019. https://github.com/intel/MLSL.

[25] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Joseph Gonzalez, Kurt Keutzer, and Ion Stoica. Checkmate: Breaking the memory wall with optimal tensor rematerialization. *Proceedings of Machine Learning and Systems*, 2: 497–511, 2020.

[26] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A unified architecture for accelerating distributed DNN training in heterogeneous GPU/CPU clusters. In *14th USENIX Symposium on Operating Systems Design and Implementation*, pages 463–479, 2020.

[27] Peter H Jin, Qiaochu Yuan, Forrest Iandola, and Kurt Keutzer. How to scale distributed deep learning? *arXiv preprint arXiv:1611.04581*, 2016.

[28] Julian Kates-Harbeck, Alexey Svyatkovskiy, and William Tang. Predicting disruptive instabilities in controlled fusion plasmas through deep learning. *Nature*, 568(7753): 526–531, 2019.

[29] Keras. Keras Applications, 2021. https://keras.io/api/applications/.

[30] Nitish Shirish Keskar, Jorge Nocedal, Ping Tak Peter Tang, Dheevatsa Mudigere, and Mikhail Smelyanskiy. On large-batch training for deep learning: Generalization gap and sharp minima. In *Proceedings of the 5th International Conference on Learning Representations*, 2017.

[31] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. URL http://arxiv.org/abs/1412.6980.

[32] Jon Kleinberg and Eva Tardos. *Algorithm Design*. Addison-Wesley Longman Publishing Co., Inc., USA, 2005. ISBN 0321295358.

[33] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. Technical Report TR-2009, University of Toronto, 2009.

[34] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 1097–1105, 2012.

[35] Hyungro Lee, Matteo Turilli, Shantenu Jha, Debsindhu Bhowmik, Heng Ma, and Arvind Ramanathan. DeepDriveMD: Deep-learning driven adaptive molecular simulations for protein folding. In *IEEE/ACM 3rd Workshop on Deep Learning on Supercomputers*, pages 12–19. IEEE, 2019.

[36] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on Operating Systems Design and Implementation*, pages 583–598, 2014.

[37] Tsung-Yi Lin, Michael Maire, Serge Belongie, Lubomir Bourdev, Ross Girshick, James Hays, Pietro Perona, Deva Ramanan, C. Lawrence Zitnick, and Piotr Dollár. Microsoft COCO: Common objects in context, 2015.

[38] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J Dally. Deep gradient compression: Reducing the communication bandwidth for distributed training. *arXiv preprint arXiv:1712.01887*, 2017.

[39] Dong C. Liu and Jorge Nocedal. On the limited memory bfgs method for large scale optimization. *MATHEMATICAL PROGRAMMING*, 45:503–528, 1989.

[40] Dong C Liu and Jorge Nocedal. On the limited memory BFGS method for large scale optimization. *Mathematical Programming*, 45(1-3):503–528, 1989.

[41] James Martens and Roger Grosse. Optimizing neural networks with Kronecker-factored approximate curvature. In *International Conference on Machine Learning*, pages 2408–2417, 2015.

[42] James Martens, Jimmy Ba, and Matt Johnson. Kronecker-factored curvature approximations for recurrent neural networks. In *International Conference on Learning Representations*, 2018.

[43] Sam McCandlish, Jared Kaplan, Dario Amodei, and the OpenAI Data Team. An empirical model of large-batch training. *arXiv preprint arXiv:1812.06162*, 2018.

[44] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory F. Diamos, Erich Elsen, David García, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed precision training. *CoRR*, abs/1710.03740, 2017. URL `http://arxiv.org/abs/1710.03740`.

[45] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed precision training, 2018.

[46] Hiroaki Mikami, Hisahiro Suganuma, Yoshiki Tanaka, Yuichi Kageyama, et al. Massively distributed SGD: ImageNet/ResNet-50 training in a flash. *arXiv preprint arXiv:1811.05233*, 2018.

[47] Ioannis Mitliagkas, Ce Zhang, Stefan Hadjis, and Christopher Ré. Asynchrony begets momentum, with an application to deep learning. In *54th Annual Allerton Conference on Communication, Control, and Computing*, pages 997–1004. IEEE, 2016.

[48] MLPerf. MLPerf Results v0.6, 2019. `https://mlperf.org/training-results-0-6`.

[49] MLPerf. MLPerf, 2021. https://www.mlperf.org/.

[50] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. PipeDream: generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 1–15, 2019.

[51] NVIDIA. NVIDIA Apex (a PyTorch extension), 2017. https://github.com/NVIDIA/apex.

[52] NVIDIA. NVIDIA Deep Learning Examples, 2019. https://github.com/NVIDIA/DeepLearningExamples.

[53] Kazuki Osawa, Yohei Tsuji, Yuichiro Ueno, Akira Naruse, Rio Yokota, and Satoshi 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.

[54] Kazuki Osawa, Yohei Tsuji, Yuichiro Ueno, Akira Naruse, Chuan-Sheng Foo, and Rio Yokota. Scalable and practical natural gradient for large-scale deep learning. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44:404–415, 2020.

[55] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[56] J. Gregory Pauloski, Zhao Zhang, Lei Huang, Weijia Xu, and Ian T. Foster. Convolutional Neural Network Training with Distributed K-FAC. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20. IEEE Press, 2020. ISBN 9781728199986. doi: 10.5555/3433701.3433826.

[57] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20. IEEE Press, 2020. ISBN 9781728199986.

[58] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450384421. doi: 10.1145/3458817.3476205. URL https://doi.org/10.1145/3458817.3476205.

[59] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. DeepSpeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery Data Mining*, KDD '20, page 3505–3506, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450379984. doi: 10.1145/3394486.3406703. URL https://doi.org/10.1145/3394486.3406703.

[60] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pages 693–701, 2011.

[61] Google Research. BERT, 2018. https://github.com/google-research/bert.

[62] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer, 2015.

[63] Alexander Sergeev and Mike Del Balso. Horovod: Fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799*, 2018.

[64] Edgar Solomonik and James Demmel. Communication-optimal parallel 2.5d matrix multiplication and lu factorization algorithms. In Emmanuel Jeannot, Raymond Namyst, and Jean Roman, editors, *Euro-Par 2011 Parallel Processing*, pages 90–109, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[65] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of collective communication operations in MPICH. *International Journal of High Performance Computing Applications*, 19(1):49–66, 2005.

[66] Yuichiro Ueno, Kazuki Osawa, Yohei Tsuji, Akira Naruse, and Rio Yokota. Rich information is affordable: A systematic performance analysis of second-order optimization using K-FAC. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '20, page 2145–2153, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450379984. doi: 10.1145/3394486.3403265. URL https://doi.org/10.1145/3394486.3403265.

[67] Chaoqi Wang, Roger Grosse, Sanja Fidler, and Guodong Zhang. EigenDamage: Structured pruning in the Kronecker-factored eigenbasis. In *36th International Conference on Machine Learning*, volume 97, pages 6566–6575. PMLR, 2019. URL http://proceedings.mlr.press/v97/wang19g.html.

[68] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. Superneurons: Dynamic GPU memory management for training deep neural networks. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 41–53, 2018.

[69] Wikipedia. Wikipedia Corpus, 2020. https://www.english-corpora.org/wiki/.

[70] Chris Ying, Sameer Kumar, Dehao Chen, Tao Wang, and Youlong Cheng. Image classification at supercomputer scale. *arXiv preprint arXiv:1811.06992*, 2018.

[71] Yang You, Igor Gitman, and Boris Ginsburg. Large batch training of convolutional networks. *arXiv preprint arXiv:1708.03888*, 2017.

[72] Yang You, Zhao Zhang, Cho-Jui Hsieh, James Demmel, and Kurt Keutzer. ImageNet training in minutes. In *47th International Conference on Parallel Processing*, page 1. ACM, 2018.

[73] Yang You, Jonathan Hseu, Chris Ying, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. Large-batch training for LSTM and beyond. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362290. doi: 10.1145/3295500.3356137. URL `https://doi.org/10.1145/3295500.3356137`.

[74] Guodong Zhang, Shengyang Sun, David Duvenaud, and Roger Grosse. Noisy natural gradient as variational inference. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 5852–5861. PMLR, 10–15 Jul 2018. URL `http://proceedings.mlr.press/v80/zhang18l.html`.

[75] Guodong Zhang, James Martens, and Roger B Grosse. Fast convergence of natural gradient descent for over-parameterized neural networks. In *Advances in Neural Information Processing Systems*, pages 8082–8093, 2019.

[76] Sixin Zhang, Anna E Choromanska, and Yann LeCun. Deep learning with elastic averaging SGD. In *Advances in Neural Information Processing Systems*, pages 685–693, 2015.

[77] Yukun Zhu, Ryan Kiros, Rich Zemel, Ruslan Salakhutdinov, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. Aligning books and movies: Towards story-like visual explanations by watching movies and reading books. In *IEEE International Conference on Computer Vision*, pages 19–27, 2015.