HyLo: A Hybrid Low-Rank Natural Gradient Descent Method

Baorun Mu*
University of Toronto
Toronto, Canada
baorun.mu@mail.utoronto.ca

Saeed Soori*
University of Toronto
Toronto, Canada
sasoori@cs.toronto.edu

Bugra Can
Rutgers University
Piscataway, NJ, US
bugra.can@rutgers.edu

Mert Gürbüzbalaban
Rutgers University
Piscataway, NJ, US
mert.gurbuzbalaban@rutgers.edu

Maryam Mehri Dehnavi
University of Toronto
Toronto, CA
mmehride@cs.toronto.edu

SC22: International Conference for High Performance Computing, Networking, Storage and Analysis | 978-1-6654-5444-5/22/$31.00 ©2022 IEEE | DOI: 10.1109/SC41404.2022.00052

Abstract—This work presents a Hybrid Low-Rank Natural Gradient Descent method, called HyLo, that accelerates the training time of deep neural networks. Natural gradient descent (NGD) requires computing the inverse of the Fisher information matrix (FIM), which is typically expensive at large-scale. Kronecker factorization methods such as KFAC attempt to improve NGD's running time by approximating the FIM with Kronecker factors. However, the size of Kronecker factors increases quadratically as the model size grows. Instead, in HyLo, we use the Sherman-Morrison-Woodbury variant of NGD (SNGD) and propose a reformulation of SNGD to resolve its scalability issues. HyLo uses a computationally-efficient low-rank factorization to achieve superior timing for Fisher inverses. We evaluate HyLo on large models including ResNet-50, U-Net, and ResNet-32 on up to 64 GPUs. HyLo converges 1.4×-2.1× faster than the state-of-the-art distributed implementation of KFAC and reduces the computation and communication time up to 350× and 10.7× on ResNet-50.

Index Terms—Natural Gradient Descent, Deep Neural Networks, Optimization

I. Introduction

Second-order methods specifically natural gradient descent (NGD) [1] have gained traction in recent years as they accelerate the training of deep neural networks (DNN) by capturing the geometry of the optimization landscape with the Fisher Information Matrix (FIM) [2]. These methods demonstrate improved convergence compared to first-order techniques such as stochastic gradient descent (SGD) as they compute the inverse of FIM and use it to precondition the gradients before parameter updates [1]. However, because the Fisher matrix is large and scales with the model size, finding its inverse is a major bottleneck in NGD methods.

Recent work [3] shows that Kronecker-Factored Approximate Curvature (KFAC) can reduce the computation cost of FIM by approximating the FIM for a batch of samples with a block diagonal matrix, where blocks correspond to layers. Other variations of KFAC have been proposed, e.g. EKFAC [4], which improve the accuracy of approximation further by rescaling the Kronecker factors, or KBFGS [5], a Kronecker-based Quasi-Newton method that uses Broyden–Fletcher–Goldfarb–Shanno (BFGS) [6] updates to accelerate KFAC. Amongst these work, to our knowledge, only the original KFAC method [3] has been implemented on large-scale distributed platforms [7]–[11]. Osawa et. al. [12] proposes a memory-optimized implementation of KFAC by distributing the layers' factor computations across workers. Ueno et. al. [7] improves on [12] by using a custom 21-bit floating point format for communication amongst work-ers and to overlap communication with the computations involved in the backward pass. Pauloski et. al. [9] proposes a communication-optimized implementation of KFAC by reducing the frequency of communications among workers and increasing the granularity of KFAC's computations. KAISA [8] improves on [12] and [9] by proposing a hybrid approach to choose between a communication-optimized and a memory-optimized implementation of KFAC. Because KAISA is based on KFAC, its scalability is limited by the computation and communication costs of computing the inverse and eigen decompositions of the Kronecker factors.

Sherman-Morrison-Woodbury-based NGD (SNGD) methods, recently introduced in [13], [14], leverage the structure of FIM for overparametrized models to use a matrix inversion technique called Sherman-Morrison-Woodbury (SMW) identity. This reduces the computation cost of inversion in NGD methods for small batch sizes. As a result, SNGD methods perform better than KFAC approaches when the batch size is relatively smaller than the layer dimension. However, SNGD methods are not suitable for distributed settings because inversion becomes a performance bottleneck as the overall batch size grows linearly with the number of workers. Also, SNGD approaches only support fully-connected layers and have not been extended to Convolutional Neural Networks (CNNs).

This work presents HyLo, a hybrid low-rank NGD method, that approximates the FIM with low-rank matrices using a gradient-based switching heuristic to decide when to switch between a Khatri-Rao-based Interpolative Decomposi-

* Both authors contributed equally to the work.

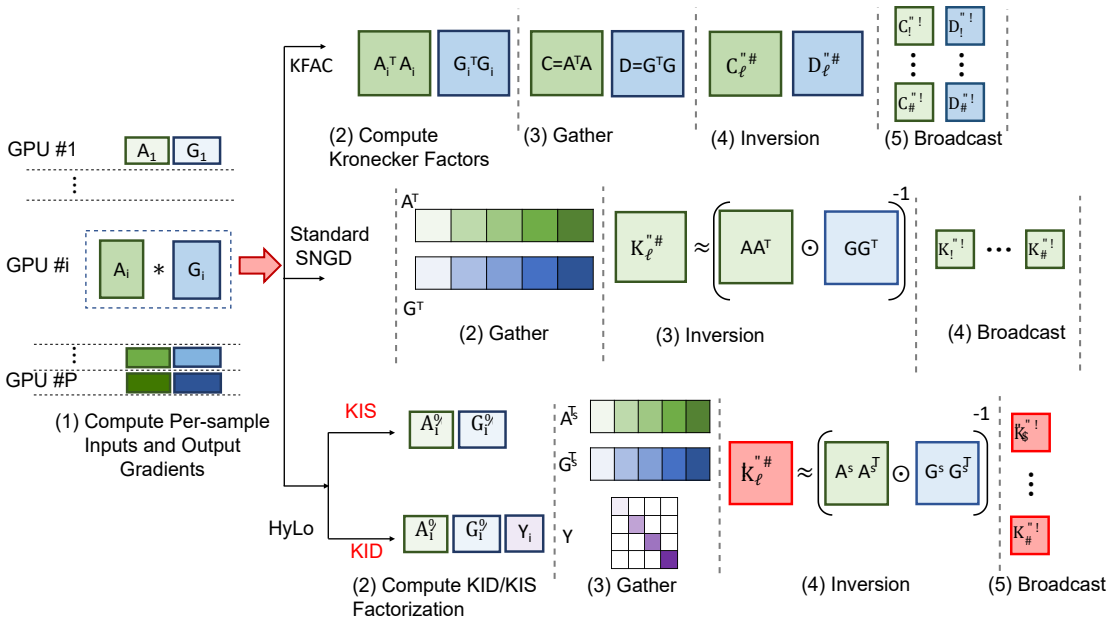


Fig. 1: HyLo vs KFAC and standard SNGD. HyLo reduces the computation and communication time of SNGD methods. It first factorizes the per-sample inputs and output gradients using Khatri-Rao-based interpolative decomposition and importance sampling. Then the reduced-size factors are gathered on workers to efficiently approximate the kernel matrix.

tion (KID) and Khatri-Rao-based Importance Sampling (KIS). HyLo is more scalable than KFAC and SNGD methods and has a lower communication/computation cost because of operating on low-rank matrices that are significantly smaller than Kronecker factors. Our contributions are:

- A novel Khatri-Rao-based interpolative decomposition, as well as an importance sampling approach that leverages the low-rank structure of FIM for fast factorization and inversion.
- A gradient-based switching strategy that determines when to switch between Khatri-Rao-based interpolative decomposition and importance sampling to maintain a good balance between accuracy and running time in NGD methods.
- HyLo uses the Sherman-Morrison-Woodbury formulation, hence we also present the first extension of the Sherman-Morrison-Woodbury NGD method to convolutional neural networks.
- A distributed implementation of HyLo which improves on the state-of-the-art distributed implementation of KFAC, i.e. KAISA, 1.4× on 64 GPUs and is 1.7× faster than SGD on ResNet-50. We reduce the computation and communication cost of NGD 350× and 10.7× compared to KAISA for ImageNet-1k on 64 GPUs.

II. Motivation

In this section, we motivate our approach by analyzing the scalability of second-order optimization methods. We first provide background on deep neural networks (DNNs) as well as first- and second-order optimization methods and then compare the computation and communication complexity of KFAC and SNGD methods at scale. Finally, we show that HyLo outperforms NGD methods in distributed settings for models with large layers and large global batch sizes. Here, global batch size refers to the total number of data points (cumulative over workers) used per iteration and will be discussed further in Section II-B.

A. Background

A deep neural network consists of L layers; each layer has parameters, i.e., weights, that are learned during the training. For simplicity, we consider a fully-connected layer with input and output dimensions d. The optimal parameters of a layer are obtained by minimizing the average loss L over the training dataset:

$$L(w) = \sum_{i=1}^{N} \ell(w, x_i) \tag{1}$$

where $\{x_i\}_{i=1}^{N}$ is a dataset with N data points and w is the parameter to be learned. Typically, a batch of m samples is used to compute the loss.

Stochastic gradient descent (SGD). Stochastic gradient descent is a first-order optimization method that uses the average gradient g_t of the loss function for a batch, to update the estimate of the parameters w_t at time t with the learning rate λ:

$$w_{t+1} = w_t - \lambda g_t \tag{2}$$

(see e.g. [15], [16]).

Natural gradient descent (NGD). Natural gradient descent belongs to the class of second-order methods. It uses the Fisher

Information Matrix (FIM) as a preconditioner for the gradient

$$w_{t+1} = w_t - \lambda(F_t + \alpha I)^{-1} g_t \quad (3)$$

where F_t is the Fisher Information Matrix (FIM) at the iterate w_t and α is the damping factor which is used to stabilize the training [1]. The FIM is an approximation of the Hessian and contains information about the curvature. Here, FIM is defined as [12]:

$$F(\cdot) = \frac{1}{m}\sum_{i}^{m} \nabla L(\cdot, x_i) \nabla L(\cdot, x_i)^T = \frac{1}{m} U^T U \quad (4)$$

where $U = [\nabla L(w, x_1), \ldots, \nabla L(w, x_m)]^T$ and $F_t = F(w_t)$. We refer to the matrix $U \in R^{m \times d}$ [2] as the Jacobian matrix. It contains the gradients for each sample in the batch, i.e., $U = [U_1, U_2, ..., U_m]^T$ where U_i is the gradient of the sample i. The Jacobian matrix has a row-wise Khatri-Rao structure, i.e.

$$U = A \bullet G \quad (5)$$

where $A \in R^{m \times d}$ and $G \in R^{m \times d}$ are the per-sample layer's inputs and output gradients and $\bullet$ is the row-wise Khatri-Rao product.

Kronecker Factorization Methods (KFAC). KFAC approximates the FIM with a block diagonal matrix, each block corresponds to a layer. Then the inverse of a block is approximated using the Kronecker product of two matrices $C_1, C_2 \in R^{d \times d}$ called the Kronecker factors, γ is the factor damping parameter

$$(F + \alpha I)^{-1} \approx \underbrace{(A^T A + \gamma I)^{-1}}_{C_1} \otimes \underbrace{(G^T G + \gamma I)^{-1}}_{C_2} \quad (6)$$

SMW-based Natural Gradient Descent Methods (SNGD). SMW-based NGD methods, which we call SNGD, leverage the structure of FIM in Equation 4 for overparametrized models as well as the Khatri-Rao structure of Jacobian in Equation 5 while using the SMW identity to obtain the inverse update:

$$(F + \alpha I)^{-1} = \frac{1}{\alpha}\left(I - U^T \underbrace{(AA^T \odot GG^T + \alpha I)^{-1}}_{K} U\right) \quad (7)$$

where $\odot$ is the element-wise matrix product. We refer to $K = AA^T \odot GG^T + \alpha I$ as the kernel matrix throughout the paper. The kernel matrix is symmetric positive semi-definite and has a dimension of the global batch size.

B. Complexity Analysis of Distributed NGD Methods

In this section, we discuss the efficient distributed implementations of KFAC (adopted from KAISA [8]) and standard SNGD methods and provide an analysis of the communication and computation costs when executed on a multi-GPU cluster; we refer to each GPU as a worker from here on. Figure 1 illustrates the distributed implementation of KFAC [8] and standard SNGD methods.

Distributed KFAC. Figure 1 shows the distributed implementation of KFAC on P workers, which involves five stages. On each worker (1) per-sample inputs A_i and output gradients G_i are computed using forward and backward passes on the network; (2) the Kronecker factors are computed according to Equation 6; (3) the Kronecker factors are communicated to other workers and their average is computed; (4) the Kronecker factors are inverted for worker's assigned layers; (5) the inverted factors are broadcast to other workers.

Distributed SNGD. To our knowledge SNGD methods have not been implemented on distributed platforms, so we show a communication-optimized implementation for SNGD based on the strategies proposed by [9] for second-order methods in Figure 1, and list the steps involved in the following. On each worker i, (1) the per-sample inputs A_i and output gradients G_i are computed during forward/backward passes on the network for a batch size of m; we refer to m as the local batch size, these matrices have a size of $m \times d$; (2) the matrices A_i and G_i are then gathered on the worker to create matrices A and G which have size $Pm \times d$. We refer to Pm as the global batch size since it shows the total number of samples used in each iteration of the training; (3) the kernel matrix of layer ℓ, K_ℓ is computed and inverted using SNGD inversion in Equation 7. (4) The inverted Kernel matrices K_ℓ^{-1} are broadcast to other workers.

The computation cost in distributed KFAC includes computing and inverting the Kronecker factors in steps 2 and 4 with a total cost of $O(d^3 + md^2)$. KFAC communicates the Kronecker factors and their inverses in steps 3 and 5 leading to an overall communication cost of $O(d^2)$. SNGD computes the kernel matrix inversion in step 3 which has a computation cost of $O(P^3m^3 + P^2m^2d)$ and communicates the per-sample inputs/output gradients and inverses in steps 2 and 4 with a total cost of $O(P^2m^2)$. KFAC's computation and communication cost is proportional to the layer dimension and has a cubic time complexity while SNGD's cost grows with a cubic rate with respect to the global batch size.

C. Distributed HyLo vs. KFAC and SNGD

From the complexity analysis in Section III.B, we observe that KFAC becomes inefficient for large layers and SNGD fails to scale for large global batch sizes. In many deep learning models, the layer dimension is large and hence KFAC becomes inefficient [18]. Figure 2 shows the distribution of layer dimension across the most popular deep learning models which shows the layer dimension is large for many layers in a model.

Figure 3 compares HyLo to standard SNGD and KFAC empirically for the ResNet-50 model on 8 to 64 GPUs. While SNGD's running time is significantly less than KFAC on 8 and 16 GPUs, it fails to scale on 64 GPUs since the global batch size increases and hence does its overall cost. However, HyLo outperforms both KFAC and SNGD on all settings because its

[1] Note that FIM is sometimes defined differently [17], such assumptions do not affect our findings.

[2] For simplifying the notations, we drop the index t.

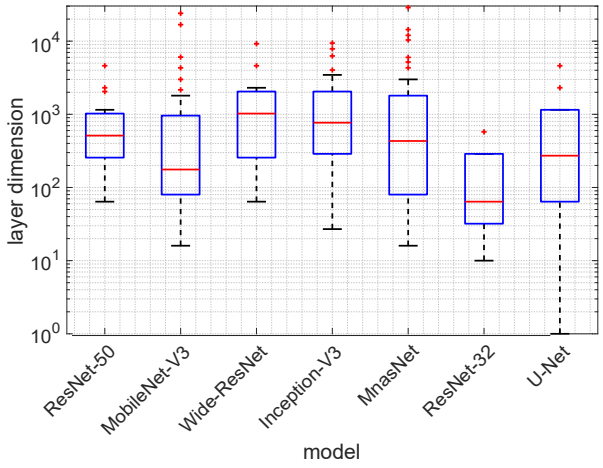


Fig. 2: Distribution of layer dimensions for different DNN models. The layer dimension is large across all models.

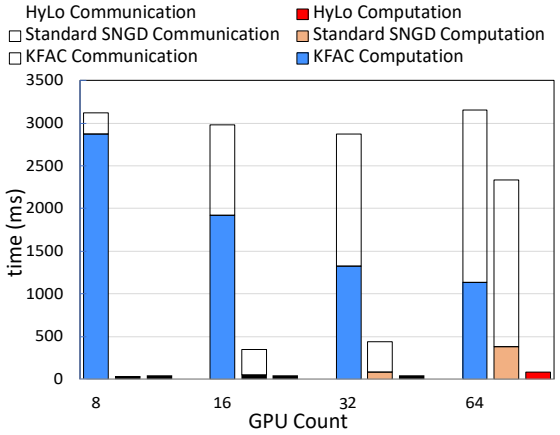


Fig. 3: Computation and communication time of KFAC, HyLo and Standard SNGD on ResNet-50 for the iterations that the second-order information is updated. Running time of KFAC and SNGD grows at scale. HyLo reduces the overall time by 28× and 20× compared to KFAC and SNGD.

overall cost is not adversely affected by the global batch size or the layer dimension.

III. HyLo: An Efficient Implementation of the Hybrid Low-Rank Second-Order Method

We propose a hybrid low-rank second-order Method, called HyLo, that uses a Khatri-Rao-based interpolative decomposition and an importance sampling method to improve the performance of NGD methods on distributed platforms. HyLo belongs to the class of SNGD methods, however, it scales efficiently for large global batch sizes. Algorithm 1 shows the steps in HyLo. It first uses a Gradient-based Switching Heuristic (lines 2-3) that decides when to switch between Khatri-Rao-based Interpolative Decomposition (KID) (lines 4 to 13) and Importance Sampling (KIS) (lines 16 to 24) to reduce the size of per-sample inputs A_i and output gradients G_i and hence the computation and communication overheads associated with them. In the following, we will first explain KID and analyze its computation/communication complexity and then discuss the importance sampling step. The gradient-based switching heuristic is explained at the end.

Algorithm 1: HyLo: A Hybrid Low-rank Natural Gradient Method

```
   /* E is the number of epochs */
1  for e : 1, ..., E do
2      R = |‖A_{-1}‖ − ‖A_{-2}‖| / ‖A_{-2}‖
         /* Gradient-based heuristic */
3      if R ≥ η or learning rate decays then
           /* T is the #iterations/epoch */
4          for t : 1, ..., T do
5              Compute per-sample inputs A_i and output
                gradients G_i
               /* Compute KID-factors with
                  algorithm 2 */
6              A_i^s, G_i^s, Y_i = KID(A_i, G_i, r)
               /* Gather KID-factors */
7              A^s = [A_1^s, ..., A_P^s], G^s = [G_1^s, ...G_P^s], Y =
                diag(Y_1, ..., Y_P)
8              for ℓ : 1, ..., L do
9                  if layer is assigned to worker then
                       /* Inversion */
10                     K̂_ℓ^{-1} = (A^s A^{s⊺} ⊙ G^s G^{s⊺})^{-1}
                       /* Broadcast */
11                     Broadcast(K̂_ℓ^{-1})
12             update w using Equation 3 and Equation 8
13             Δ_e = Δ_e + g_t
14     else
15         for t : 1, ..., T do
16             Compute per-sample inputs A_i and output
                gradients G_i
               /* Compute KIS-factors with
                  algorithm 3 */
17             [A_i^s, G_i^s] = KIS(A_i, G_i, r)
               /* Gather KIS-factors */
18             A^s = [A_1^s, ..., A_P^s], G^s = [G_1^s, ...G_P^s]
19             for ℓ : 1, ..., L do
20                 if layer is assigned to worker then
                       /* Inversion */
21                     K̂_ℓ^{-1} = (A^s A^{s⊺} ⊙ G^s G^{s⊺} + αI)^{-1}
                       /* Broadcast */
22                     Broadcast(K̂_ℓ^{-1})
23             update w using Equation 3 and Equation 9
24             Δ_e = Δ_e + g_t
```

A. Khatri-Rao-based Interpolative Decomposition

HyLo per worker uses a Khatri-Rao-based interpolative decomposition to reduce the size of per-sample inputs/output gradients and hence the computation cost of the kernel matrix

inversion and the communication cost of the gather and broadcast steps (steps 3 and 5 in Figure 1). First, the matrices A_i and G_i (line 5) are computed. In line 6, the per-sample inputs and output gradients matrices are approximated with smaller matrices, which we call KID-factors, using interpolative decomposition. Each worker then communicates the KID-factors to other workers. We propose a Khatri-Rao-based ID, shown in Algorithm 2, that leverages the structure of Jacobian in Equation 5 to create the KID-factors. KID applies the factorization to the Gram matrix $A_i A_i^T \odot G_i G_i^T$, shown in line 1, which has a smaller size compared to individual matrices A_i and G_i, and uses its decomposition to create the KID-factors A_i^s, G_i^s, and Y_i shown in line 4. In the second step, shown in line 7 of Algorithm 1, A_i^s, G_i^s, and Y_i are gathered on the worker to create matrices $A^s = [A_1^s, \ldots, A_P^s]$, $G^s = [G_1^s, \ldots, G_P^s]$ and $Y = \text{diag}[Y_1, \ldots, Y_P]$, where diag(.) creates a block diagonal matrix. Each worker computes and inverts the approximated kernel matrix for its assigned layers, shown in lines 9-10. We use A, G and Y to efficiently compute the kernel matrix inversion. To reduce the inversion cost, we apply the SMW formula [19] to the approximated kernel and obtain:

$$(F + \alpha I)^{-1} \approx \frac{1}{\alpha} I - U^s \left[Y - Y(\hat{K}^{-1} + Y)^{-1} Y \right] U^{sT} \quad (8)$$

where $\hat{K} = A^s A^{sT} \odot G^s G^{sT}$ and $U^s = A^s \odot G^s$ are the reduced size kernel and Jacobian. Once $\hat{K}$ is inverted, it is broadcast to all workers, as shown in line 11.

Algorithm 2: Khatri-Rao-based Interpolative Decomposition

Input : Per-sample inputs and output gradients matrices $A_i \in R^{m \times d}$, $G_i \in R^{m \times d}$, rank r.
Output: $A_i^s \in R^{r \times d}$, $G_i^s \in R^{r \times d}$ and the projected approximation error matrix $Y_i \in R^{r \times r}$.

/* Form Gram matrix */
1 $Q_i = A_i A_i^T \odot G_i G_i^T$
/* Compute row indices S and projection P: $Q_i \approx P Q_{i(S:)}$ */
2 $[P, S] = \text{ID}(Q_i, r)$
/* Compute the residue */
3 $R = Q_i - P Q_{i(S:)}$
/* Compute the KID-factors */
4 $A_i^s = A_{i(S:)}, G_i^s = G_{i(S:)}, Y_i = P^T (R + \alpha I)^{-1} P$
5 return A_i^s, G_i^s, Y_i

B. Khatri-Rao-based Importance Sampling

An alternative to reduce the size of per-sample inputs A_i and output gradients G_i is to use randomized sampling. HyLo uses Khatri-Rao-based importance sampling [21], shown in line 17 in Algorithm 1, to create the reduced-sized per-sample inputs and gradients, called KIS-factors. We apply norm-based sampling to reduce the size of A_i and G_i by choosing the samples that contribute the most to approximating the kernel matrix. The steps for importance sampling are shown in Algorithm 3. It assigns scores using Euclidean norm to each sample of a batch and selects from those accordingly. The Khatri-Rao structure of Jacobian, (Equation 5), allows for an efficient evaluation of scores. In particular, the score for sample j is obtained as the product of its input and gradient norm, i.e., $\|A_{i(j:)}\| \|G_{i(j:)}\|$ where $A_{i(j:)}$ is the j-th row of A_i and similarly for G_i. Once the KIS-factors A_i^s and G_i^s are created, they are gathered on the workers to form $A^s = [A_1^s, \ldots A_P^s]$ and $G^s = [G_1^s, \ldots, G_P^s]$.

Each worker inverts the kernel matrix K for its assigned layers, shown in lines 20-21. K is approximated with a smaller matrix using A^s and G^s. Hence the FIM is inverted by:

$$(F + \alpha I)^{-1} \approx \frac{1}{\alpha} I - U^s \hat{K}^{-1} U^{sT} \quad (9)$$

where $\hat{K} = A^s A^{sT} \odot G^s G^{sT} + \alpha I$ and $U^s = A^s \odot G^s$ are the reduced size kernel and Jacobian. Once the kernel matrix for a layer is inverted, it is broadcast to all workers, as shown in line 22.

Algorithm 3: Khatri-Rao-based Importance Sampling

Input : Per-sample inputs and output gradients matrices $A_i \in R^{m \times d}$, $G_i \in R^{m \times d}$, number of samples r.
Output: $A_i^s \in R^{r \times d}$, $G_i^s \in R^{r \times d}$

/* Compute norm of inputs and gradients */
1 for j : 1, ..., m do
2 $\quad P_j = \|A_{i(j:)}\|, Q_j = \|G_{i(j:)}\|$
/* Compute sample scores */
3 $\Omega = P \odot Q$
/* Choose r samples at random based on scores Ω */
4 $A_i^s, G_i^s = \text{sample}(A_i, G_i, \Omega, r)$

Table I summarizes the complexity analysis of HyLo and KFAC [8] implemented on a distributed platform with P workers for each step of the methods[3]. The computation cost of HyLo involves the cost of factorization and inversion step and is $O\left(r^3 + m^3 + (m^2 + r^2)d\right)$. HyLo reduced the computation cost compared to KFAC and SNGD from $O(d^3)$ and $O(P^3 m^3)$ to $O(r^3 + m^3)$. HyLo also reduced the communication cost compared to KFAC and SNGD from $O(d^2)$ and $O(P^2 m^2)$ to $O(r^2)$.

C. Gradient-based Switching

HyLo chooses between Khatri-Rao-based interpolative decomposition and importance sampling at the beginning of an epoch. The accumulated gradient of the objective function w.r.t the parameters is used as a metric to determine which epochs benefit the most from KID (the method that has a lower approximation error) or KIS (the method with the lower

[3] KIS has a computation complexity of $O(m^2 + md)$ which is asymptotically smaller than KID and hence omitted.

TABLE I: Comparison between the computation and communication complexities of distributed KFAC, standard SNGD, and HyLo on a system with P workers for a fully-connected layer with input/output dimension = d. m is the batch size per worker for KAISA and SNGD. r is the rank of the kernel matrix and $\rho = \frac{r}{P}$ is the number of samples per worker for HyLo. The storage complexities for second-order methods along with ADAM and SGD is also provided.

	Computation		Communication		Storage
	Factorization	Inversion	Gather	Broadcast	
HyLo	$O(m^2d + m^3)$	$O(r^3 + r^2d)$	$O(\rho d)$	$O(r^2)$	$O(rd + r^2 + d^2)$
KFAC [8]	$O(md^2)$	$O(d^3)$	$O(d^2)$	$O(d^2)$	$O(d^2)$
SNGD	-	$O(P^3m^3 + P^2m^2d)$	$O(md)$	$O(P^2m^2)$	$O(Pmd + P^2m^2 + d^2)$
ADAM [20]	-	-	-	-	$O(d^2)$
SGD	-	-	-	-	$O(d^2)$

computation complexity). From [22], epochs with a larger change in their accumulated gradient, i.e., critical epochs, contribute more to the training. Small gradient errors in critical epochs can impair the training progress. Hence, our switching method uses KID for critical epochs and importance sampling for others. As shown in Algorithm 1 lines 2-3, an epoch is critical when the learning rate decays or when the accumulated gradient changes pass a threshold η:

$$\frac{\|\Delta_e\| - \|\Delta_{e-1}\|}{\|\Delta_{e-1}\|} \geq \eta \tag{10}$$

where $\|\Delta_e\|$ is the l2-norm of accumulated gradients for the e-th epoch.

IV. HyLo for Convolutional Neural Networks

HyLo is an SNGD-based approach. Since, to our knowledge, SNGD approaches are not formulated to support convolutional neural networks (CNNs), in this section, we present the first extension of SNGD methods to CNNs. First, we briefly review convolutional layers and then propose an efficient formulation of SNGD for convolutional layers.

A convolutional layer operates on large dimensional tensors. In particular, the output tensor Y of a convolutional layer is obtained by applying convolution to its input X and weights. To simplify the notations, we assume $X, Y \in R^{d\times s\times s}$, i.e., they contain d images of size $s \times s$. Similarly, we denote the corresponding output gradient with G.

The SNGD method in Equation 7 does not apply to CNNs because convolutional layers typically operate on 4D tensors while the SMW identity is only for matrices (2D tensors). We extend the SNGD formulation to convolutional layers by first expressing the convolution operation as a matrix multiplication between reshaped input and output tensors. Formally, the gradient for one sample is computed by $X^T \cdot G$ where $X = im2col(X)$ and $G = vec(G)$ in which $im2col$ unfolds a three dimensional tensor to a matrix by reshaping its blocks into 1-D vectors and vec reshapes the tensor G by flattening its spatial dimension. Then the reshaped tensors are approximated over their spatial dimensions. In particular, $\hat{x} = \sum_i^S X_{(i:)}$ and $\hat{g} = \sum_i^S G_{(i:)}$ where $X_{(i:)}$ and $G_{(i:)}$ are the i-th row of X and G respectively. By concatenating the vectors $\hat{x}$ and $\hat{g}$ for a batch of samples, the approximated per-sample inputs and gradients matrices are obtained, i.e. $\hat{X} = [\hat{x}_1, ..., \hat{x}_m]^T$ and $\hat{G} = [\hat{g}_1, ..., \hat{g}_m]^T$. The approximated Jacobian $\hat{U}$ is computed with their row-wise Khatri-Rao product, similar to Equation 5. Finally, we leverage this structure and reformulate SNGD:

$$(F + \alpha I)^{-1} \approx \frac{1}{\alpha}(I - \hat{U}^T C_1 \odot C_2 + \alpha I^{-1}\hat{U}) \tag{11}$$

where $C_1 = \hat{X}\hat{X}^T$ and $C_2 = GG^T$.

TABLE II: Models and datasets used for experimental results.

Model	Dataset	Target	GPU (#, type)	
ResNet-50	ImageNet-1k	75.9%	64	V100
U-Net	LGG Segmentation	91%	4	V100
ResNet-32	CIFAR-10	92.5%	32	K80
DenseNet	CIFAR-100	75%	1	V100
3C1F	Fashion-MNIST	93%	1	V100

V. Results

We compare the performance of HyLo on a cluster of 64 GPUs to KAISA [8], the state-of-the-art distributed implementation of KFAC, and the efficient distributed implementation of stochastic gradient descent and ADAM [20], used as a first-order optimization method.[4] To show that HyLo also improves the performance for small batch sizes, we compare it on a single-GPU to KFAC and to other two more-recent methods EKFAC [4] and KBFGS-L [23]. The methods EKFAC and KBFGS-L do not have distributed implementations.

A. Methodology

The datasets and deep learning models listed in Table II are used for the experimental results.

Datasets. We use datasets from image classification and segmentation applications. ImageNet-1k [24] has 1000 categories with approximately 1.3M training images and 50K validation images. CIFAR-10 and CIFAR-100 [25] consist of 50K training images and 10K validation images in 10 classes. The LGG Segmentation dataset [26] contains Magnetic Resonance (MR) images of the brain. We use 3336 images for training and 332 images for validation. Fashion-MNIST [27] consists of a training set of 60K images and a test set of 10K examples that belong to 10 classes.

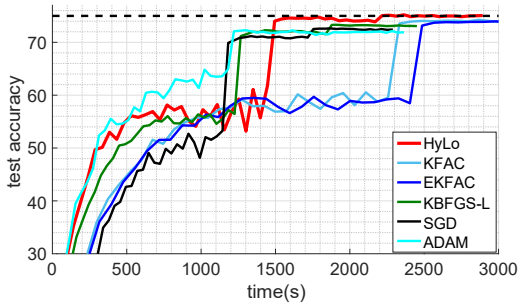
[4]The only work that attempts to implement SNGD is SENG [14], however, their approach at a large scale does not communicate second-order information and hence is not a (standard) NGD method.

Models. We use state-of-the-art deep learning models listed in Table II for the experimental results. For Fashion-MNIST a network with three convolutional layers and one fully-connected layer is used, hence called 3C1F. For other benchmarks, the following DNN models are used: ResNet-50, U-Net, ResNet-32 and DenseNet.

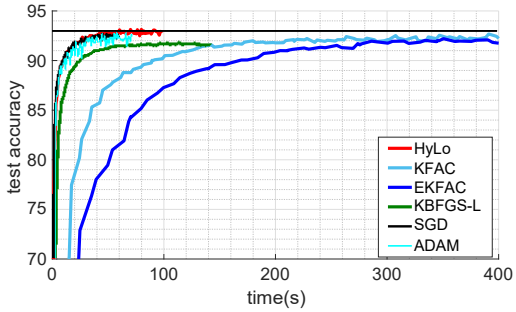
Target accuracy. Following lists our baselines for target accuracy for different experiments: (1) For ResNet-50, we use the MLPerf benchmark target results [28]; (2) For U-Net, we use the target validation Dice similarity coefficient (DSC) [26]. (3) For ResNet-32, we use the target test accuracy reported in [29]. (4) For DenseNet we use the test accuracy in [30]. (5) For 3C1F we use the test accuracy of tuned SGD.

GPU Clusters. The results for ResNet-50 and U-Net are obtained on the Mist cluster [31]. Mist has 54 nodes, each with 32 IBM Power9 cores with 256GB RAM and 4 NVIDIA V100 GPUs with 32GB memory and NVLink in between. Nodes are connected via InfiniBand EDR. The results for ResNet-32 is obtained on Amazon Web Services (AWS) P2.8xlarge system which has 8 K80 GPUs with 12 GB memory. DenseNet and 3C1F are ran on a single V100 GPU.

Software. We use PyTorch 1.7.1, CUDA 10.2, CUDNN 7.6.5, and NCCL 2.7.8.



(a) DenseNet



(b) 3C1F

Fig. 4: Test accuracy comparison between HyLo and KFAC, EKFAC, KBFGS-L, ADAM and SGD for a) DenseNet b) 3C1F on single-GPU. The dotted line shows the target metric.

Training parameters. We train ResNet-50 for 50 epochs for HyLo and KAISA, 90 epochs for SGD and 65 epochs for ADAM with a batch size of 80 per GPU, similar to [8]. We use a similar approach to [8] and train U-Net for 30 epochs for HyLo and KAISA and 50 epochs for ADAM and SGD with a batch size of 16 per GPU. ResNet-32 is trained for 100 epochs for HyLo and KAISA, 200 for SGD and 150 epochs for ADAM with a batch size of 128 per GPU, similar to [8]. DenseNet and 3C1F models are trained for 60 epochs with a batch size of 128. We use momentum, and tune the learning rate and weight decay for all methods. For HyLo and KAISA damping is tuned. All the methods except SGD and ADAM update the second-order information every few iterations. The frequency of the update for HyLo is similar to that of KAISA and is chosen according to the authors' default parameters for KFAC, EKFAC, and KBFGS-L. We choose the parameter r in KID and KIS as 10% of the global batch size in all experiments unless otherwise stated.

B. Single-GPU Setting

This section shows the performance of HyLo on a single-GPU for smaller models DenseNet and 3C1F. We compare HyLo to KFAC, EKFAC, KBFGS-L, SGD, and ADAM. In order to demonstrate the performance of HyLo, we compare the single-GPU implementation of these methods for small batch sizes.

Figure 4a shows the test accuracy vs. time on DenseNet model. HyLo converges to the target test accuracy of 75% in 36.9 minutes while KFAC and EKFAC achieve an accuracy of 74.2% in 50 minutes. KBFGS-L ,SGD, and ADAM, respectively, achieve a lower accuracy of 73.2%, 73%, and 72.3% with a time-to-convergence of 40.8, 38, and 38 minutes. HyLo outperforms all methods in accuracy and is 1.4× faster than KFAC.

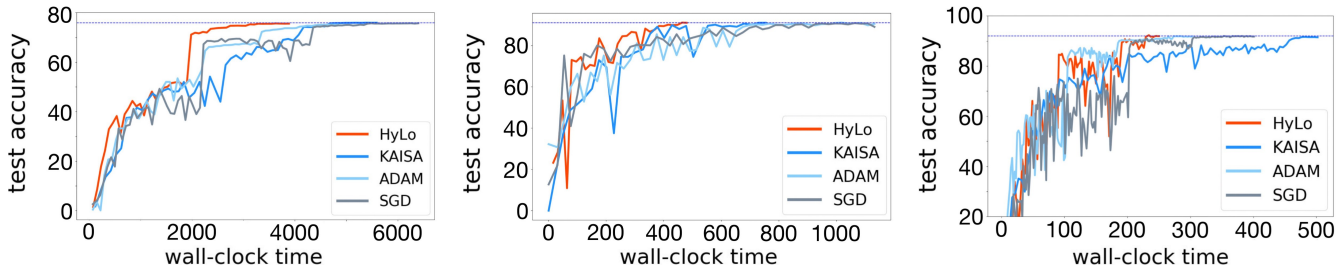
To further demonstrate the generalization performance of HyLo, we compare its test accuracy to KFAC, EKFAC, KBFGS-L, SGD, and ADAM for the 3C1F model tested on Fashion-MNIST in Figure 4b. HyLo achieves the target accuracy and outperforms KBFGS-L with a margin of 1.5% and KFAC and EKFAC with 0.95%. HyLo also accelerates the end-to-end training time up to 3× compared to KFAC and EKFAC.

C. Multi-GPU Setting

To demonstrate the performance of HyLo at scale, we conduct experiments on ResNet-50, U-Net, and ResNet-32 on up to 64 GPUs and compare HyLo with KAISA, SGD and ADAM. First, we show the end-to-end accuracy and time-to-convergence and then analyze HyLo's performance with communication and computation cost breakdown.

Accuracy and time-to-convergence. Figure 5a compares the test accuracy of HyLo to KAISA, SGD and ADAM for ResNet-50 model on 64 GPUs. HyLo converges to the target accuracy in 64.8 minutes which is 1.4× faster than KAISA with 93.2 minutes , 1.7× faster than SGD with 106.7 minutes and 1.3× faster than ADAM with 83.1 minutes. We also show the test accuracy curve w.r.t epochs in Figure 6a. HyLo improved the convergence and outperforms KAISA, ADAM and SGD in per-epoch accuracy.

HyLo outperforms KAISA, ADAM and SGD with a large factor for U-Net and ResNet-32 models as shown in Figure 5b



(a) ResNet-50 (b) U-Net (c) ResNet-32

Fig. 5: Test accuracy vs time for HyLo, KAISA, ADAM and SGD. The dotted line is the target accuracy.

and Figure 5c. For U-Net, the time-to-convergence for HyLo is 480s which is 2.4×, 1.6× and 2.4× faster than ADAM , KAISA and SGD respectively. For ResNet-32, HyLo reduces the training time by factors 2.1×, 1.6× and 1.2× compared to KAISA, SGD and ADAM. Figure 6c and Figure 6b compare the per-epoch accuracy of HyLo with KAISA, SGD and ADAM on ResNet-32 and U-Net. For ResNet-32, HyLo shows a better convergence compared to SGD and ADAM and outperforms KAISA in per-epoch accuracy.

Performance analysis. We compare the cost of different steps involved in HyLo and KAISA to show the advantage of HyLo when distributed. HyLo and KAISA on ResNet-50 are trained on ImageNet-1k on 64 GPUs and ResNet-32 for CIFAR-10 is on 32 GPUS. Even though HyLo is a hybrid approach, we intentionally report separate times for its KID and KIS methods to further examine their computation and communication costs. The computation time consists of the time for factorization and inversion steps and the communication time includes the gather and broadcast steps.

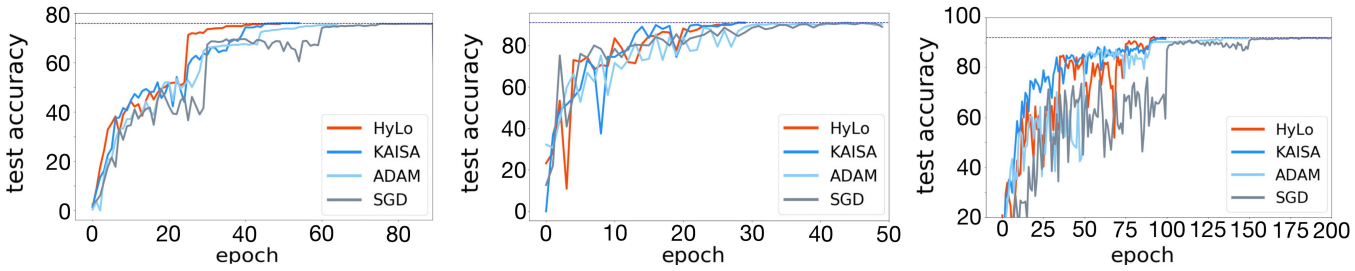
Computation cost. HyLo reduces the factorization time compared to KAISA by 27× for KID and 350× for KIS on ResNet-50, as shown in Figure 7a. The large speedup of KIS compared to KAISA is due to its low-cost norm-based sampling which is efficiently computed with algorithm 3. The KID iterations involve interpolative decomposition and hence have a higher execution time. HyLo also reduces the inversion time to 3ms which is ⍰135× less than KAISA with 410ms. As shown in Figure 2, ResNet-50 has large layers and hence KAISA's inversion becomes inefficient. Figure 7b shows that the speedups gained from HyLo are more significant for U-Net model in which the inversion cost is reduced to 1.5ms which is 600× less than KAISA.

HyLo also reduces the computation time when the model has layers with smaller dimensions. As shown in Figure 7c, the time of the factorization and inversion steps for KID are 6ms and 7ms which are 9× and 47× faster than that of KAISA. The speedups gained on ResNet-32 are relatively smaller compared to ResNet-50. This correlates with the observation in Figure 2 (which shows the layer dimension distribution) that ResNet-50 has larger layers.

Communication cost. The time for communicating the factors in the gather step is reduced by a factor of 9.4× for KID and 10.7× for KIS compared to KAISA on ResNet-50, as shown in Figure 7a. The KID method transfers three matrices, i.e. KID-factors, per layer and hence has a slightly higher communication time in the gather step while KIS only communicates two matrices per layer. Finally, we observe that the dominant communication time is in the broadcast step, with 18ms for KID and 13ms for KIS which are 36× and 48× less than that of KAISA with 664ms. Figure 7b shows the communication time breakdown of HyLo and KAISA on U-Net. The time for the gather step in HyLo is ⍰1ms which is ⍰20× less than KAISA. HyLo reduces the communication cost of broadcast step 8× compared to KAISA. Figure 7c shows the communication time breakdown of HyLo and KAISA for ResNet-32. The gather time is reduced compared to KAISA by factors 2.5× and 4× for KID and KIS methods. Both KID and KIS have similar times in the broadcast step and improve over KAISA by a factor of 2.1×.

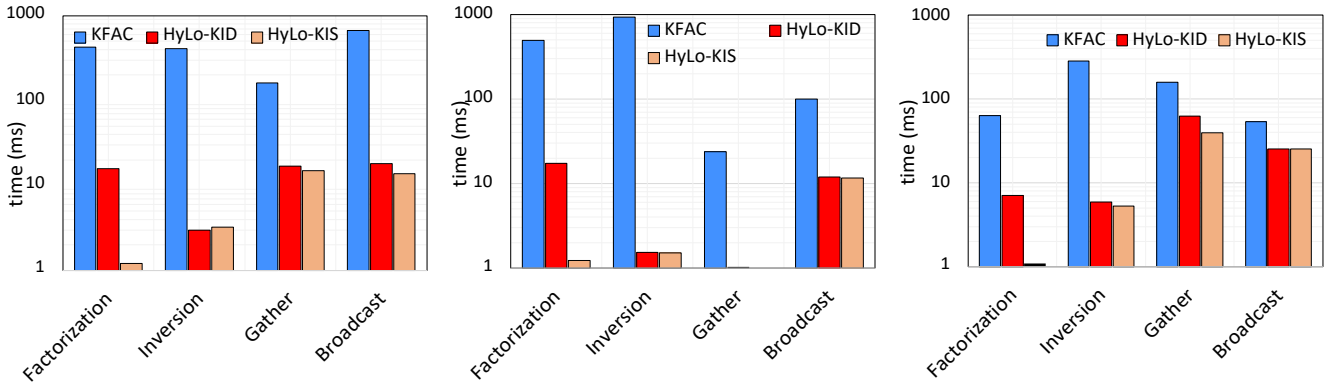
Scaling. To demonstrate the scalability of HyLo, Similar to [8], we report the projected end-to-end training time speedup for HyLo over SGD in Figure 8a. We measure the average time-per-epoch for HyLo and SGD; the number of GPUs is varied from 8 to 64 for ResNet-50 and 4 to 32 for ResNet-32 and U-Net. Also, to show the effect of the kernel matrix rank on the scaling, r is set (See Algorithm 1) to 10%, 20%, and 40% of the global batch size. For ResNet-50, we project the training time to 90 epochs in SGD and 50 epochs in HyLo, for ResNet-32 the training time is projected to 200 and 100 epochs for SGD and HyLo respectively, and for U-Net the training time is projected to 30 and 50 epochs for HyLo and SGD. The frequency of HyLo updates is scaled inversely with the number of GPUs to keep the number of updates per training sample constant. The speedup of HyLo improves over SGD with an increasing number of GPUs. HyLo achieves ⍰1.9× speedup for ResNet-32 on 32 GPUs, ⍰1.7× speedup for ResNet-50 on 64 GPUs, and ⍰1.3× speedup for U-Net on 32 GPUs. For a more accurate scalability analysis, in Figure 9 we show the running time of HyLo vs its single-GPU time for varying numbers of GPUs on ResNet-50, ResNet-32, and U-Net models. Figure 9 shows that HyLo scales superlinearly for ResNet-50 and U-Net and linearly for ResNet-32.

Analysis of rank and the switching method. HyLo is built on the assumption that the Kernel matrix has a low-rank structure, this property allows us to replace the per-sample inputs and gradients matrices with KID/KIS-factors.



(a) ResNet-50 (b) U-Net (c) ResNet-32

Fig. 6: Test accuracy vs epoch for HyLo, KAISA, ADAM and SGD. The dotted line is the target accuracy.



(a) ResNet-50 (b) U-Net (c) ResNet-32

Fig. 7: Computation and communication time of HyLo and KAISA on ResNet-50, U-Net, and ResNet-32 models. Computation time is measured for factorization and inversion steps and communication time includes gather and broadcast times.

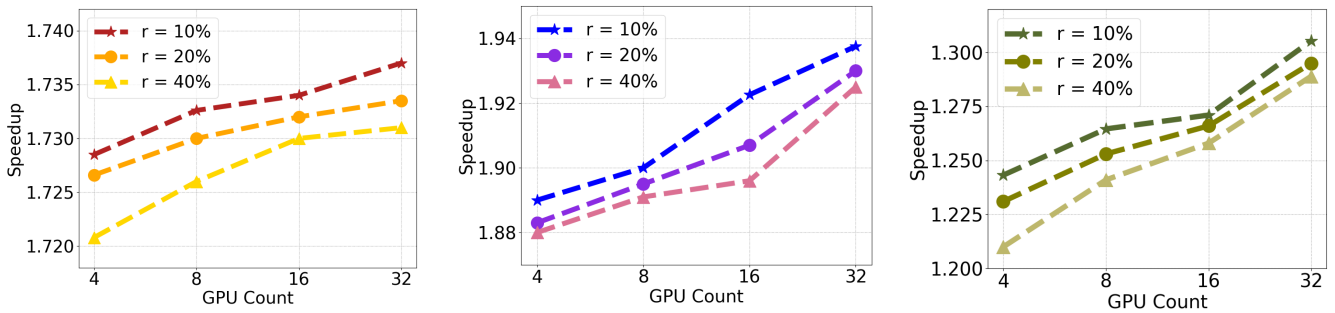
The following shows that the kernel matrix has a low-rank structure when global batch sizes are large. We also provide an analysis of our switching-based method which decides how often to choose KID versus KIS.

Analysis of kernel matrix rank. For different batch sizes, we analyze the rank distribution of the kernel matrix on ResNet-50, ResNet-32, U-Net, and DenseNet models. We compute the eigenvalue decomposition of the kernel matrix for each layer and report its numerical rank, i.e., the number of eigenvalues that contribute to 90% of their sum. The global batch size ranges from 512 to 4096 for ResNet-50 and ResNet-32. The kernel matrix maintains a low-rank structure for all global batch sizes. Figure 10a shows the rank distribution on ResNet-50, the median rank is 104, 164, 247, and 351 which respectively, show a ratio of 20%, 16%, 12%, and 8.5 % of the global batch size. Figure 10b for ResNet-32 shows that on a large global batch size of 4096 we only require 2% of the samples to approximate the matrix as the median rank is small, 89, compared to the batch size. For U-Net, the median rank is 3, which is 5% of the global batch size. For the DenseNet model, the rank distribution is similar to ResNet-32 with a median of 9.

Analysis of switching method. We provide an analysis of how often HyLo chooses KID versus KIS. We compute the norm of the gradient at each epoch on the ResNet-32 model and report in Figure 11 for various layers. The gradient norm changes rapidly in the initial epochs and also after the learning rate decays at epochs 35 and 75. Hence, HyLo chooses KID over KIS in 20% of epochs, i.e., 1-10, 35-39, and 75-79. A similar analysis leads to choosing KID in 30% of the epochs on ResNet-50, in particular epochs 1-10 and 25-29.

In order to analyze the impact of KID and KIS updates, we measure the normalized gradient error for ResNet-50 and ResNet-32 and report it in Figure 12. The normalized gradient error measures the approximation error of KID and KIS and is defined as $\epsilon = \|\hat{g}_t - g_t\|/\|g_t\|$, where $\hat{g}_t$ is the gradient computed with KID/KIS and g_t is the gradient computed without the KID/KIS approximations. As seen in Figure 12, the KID error is around an order of magnitude smaller than KIS because its approximation provides a tighter error bound for the Kernel matrix K [32]. For ResNet-32, the gap between the KID and KIS error is much larger because it has a smaller numerical rank (the upper bounds for the approximation error can be found in Theorem 1 in [32] and Theorem 3.1 in [33]).

To further demonstrate the usefulness of the switching strategy, we compare the performance of HyLo using two different settings: 1) we alternate between KIS and KID randomly at each epoch with a probability of 50%, hence called the Random method, 2) the original HyLo with its switching method as described in section III. From Table III



(a) Speedup on ResNet-50. (b) Speedup on ResNet-32. (c) Speedup on U-Net.

Fig. 8: The speedup of HyLo over SGD for a) ResNet-50 b) ResNet-32 c) U-Net on different number of GPUs, r is the rank.

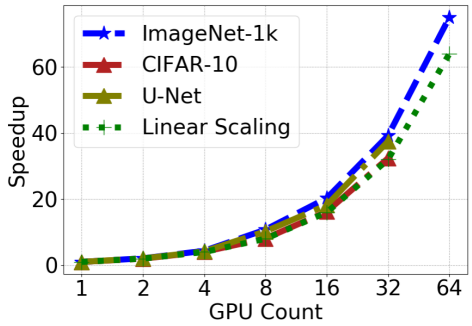


Fig. 9: HyLo's scalability on ResNet-50, ResNet-32 and U-Net.

TABLE III: Comparison of test accuracy and time-to-convergence between HyLo and Random.

Model	Accuracy(%)		Time(s)	
	HyLo	Random	HyLo	Random
ResNet-50	75.6	75.29	3960	4258
ResNet-32	92.32	91.68	260	497
U-Net	91	90.2	410	445

the Random method reaches a similar accuracy to HyLo for ResNet-50 and gives a lower accuracy on ResNet-32 and U-Net. However, Random is 7.5%, 91%, and 8.5% slower than HyLo for ResNet-50, ResNet-32, and U-Net respectively because it is using more KID updates which are slower than KIS but they have a better accuracy (see Figure 12).

Analysis of memory footprint. Table IV provides the memory footprint of all methods. HyLo's memory usage is 2× and 20× lower than KAISA on ResNet-50 and U-Net, respectively, because the KID/KIS-factors are smaller than Kronecker factors in KAISA. HyLo's memory footprint is 3× less than ADAM on U-Net because the global batch size is much smaller than the layer dimension in U-Net. Compared to SGD, all methods have a higher memory footprint because SGD only stores the gradient.

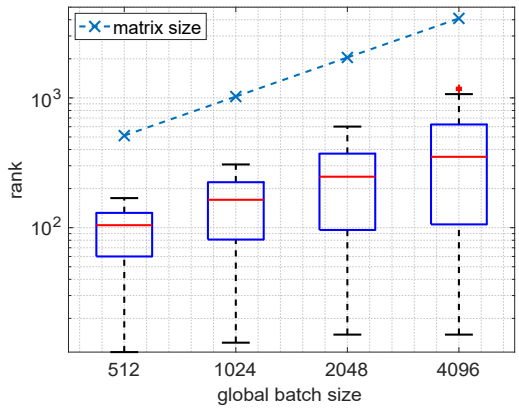
TABLE IV: Memory overhead for different methods.

	HyLo	KAISA	ADAM	SGD
ResNet-50	317.3 MB	713.9 MB	306.7 MB	102.2 MB
ResNet-32	35.5 MB	34.9 MB	5.6 MB	1.9 MB
U-Net	31.5 MB	603.2 MB	93.2 MB	31.1 MB

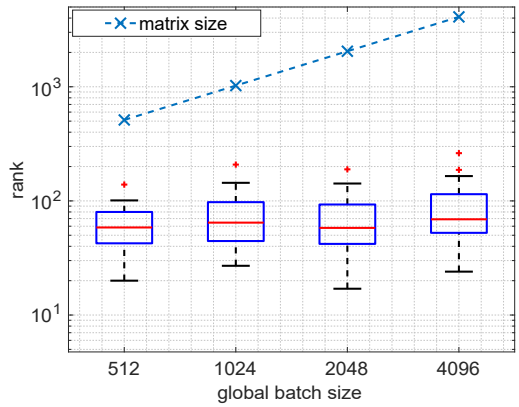
VI. Related Work

Second-order methods specifically Natural Gradient Descent (NGD) [1], [34]–[36] can accelerate training by improving the convergence rate of DNNs, specifically, exact NGD [37] methods have demonstrated improved convergence compared to first-order techniques such as SGD [15]. Zhang et al. [37] extend NGD to deep nonlinear networks with non-smooth activations and show that NGD converges to the global optimum with a linear rate. However, their method fails to scale to large or even moderate size models primarily because it relies heavily on backpropagating Jacobian matrices, which scale with the network's output dimension [38]. In [13] the authors use Woodbury identity for the inversion of the Fisher matrix and propose a unified framework for subsampled Gauss-Newton and NGD methods. Their framework is targeted at fully-connected networks and relies on empirical Fisher. This requires extra forward-backward passes to perform parameter updates [38], [39].

Approximate NGD approaches such as [40]–[45] attempt to improve the overall execution time of NGD with FIM inverse approximation. For example, KFAC [40] approximates each block inverse using the Kronecker product of two smaller matrices, i.e. Kronecker factors. However, these factors have large sizes for wide layers and hence their inversion is expensive. EKFAC [41] improves the approximation used in KFAC by rescaling the Kronecker factors with a diagonal matrix obtained via costly singular value decompositions. Other work such as KBFGS [42] further estimates the inverse of Kronecker factors using low-rank BFGS type updates. WoodFisher [46] estimates the empirical FIM block inverses using rank-one updates, however, this estimation will not contain enough useful curvature information to produce a good search direction [3]. Goldfarb et al. [42] follow the framework for stochastic quasi-



(a) ResNet-50



(b) ResNet-32

Fig. 10: Distribution of the rank of kernel matrix on a) ResNet-50 and b) ResNet-32. The kernel matrix has a low-rank structure for all global batch sizes.

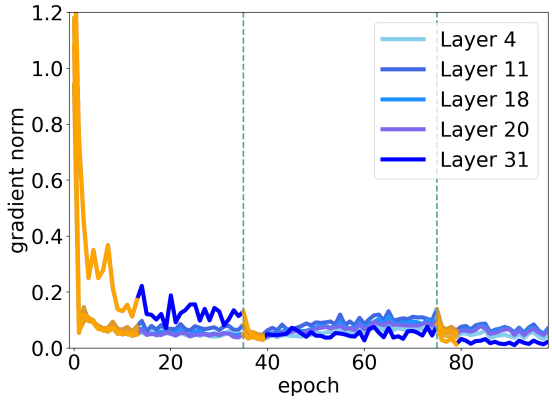
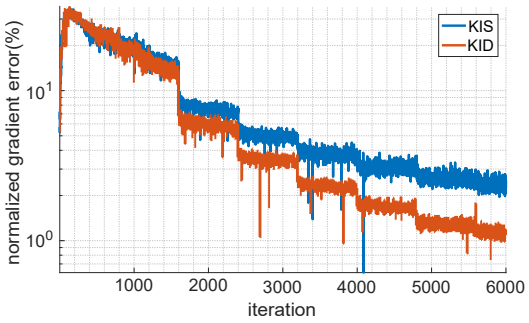


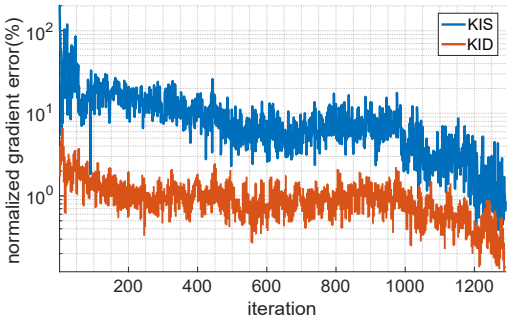
Fig. 11: Gradient norms throughout ResNet-32 training.

Newton methods and prove that KBFGS converges with a sublinear rate for a network with bounded activation functions.

Distributed implementations of KFAC methods have been recently explored in [7]–[9], [11], [12], [47]. Ba et al. [11]



(a) ResNet-50



(b) ResNet-32

Fig. 12: Normalized gradient error for KID and KIS.

implement an asynchronous distributed KFAC on a parameter server. Their work assigns the factorization step in KFAC to some of the workers while others compute inversions. Their approach leads to a 5.9% accuracy loss on ImageNet-1k compared to state-of-the-art target accuracies. Osawa et al. [12] perform the factorization step on all workers and compute the factor inverses using a model-parallel approach. They apply mixed-precision computations (FP32 and FP16) and use symmetry-aware communication to reduce the volume of transferred data by only sending the upper triangular part of Kronecker factors. Pauloski et al. [9] use a simi-lar approach to [12] and implement KFAC layer-wise, their method achieves the MLPerf target accuracy and improves training time over SGD. This work was further improved in [7], by applying a custom 21-bit floating point format for collective communications amongst GPUs and overlapping the gather/broadcast steps with the backward pass computations. KAISA [8] combines previous distributed KFAC strategies into one hybrid approach using a tunable memory footprint approach to balance the memory and communication costs. Ma et al. [47] provides an extensive analysis of distributed KFAC and shows that KFAC does not exhibit good scalability behavior and loses accuracy on large batch sizes.

VII. Conclusion

In this work, we propose a distributed algorithm and implementation for Natural Gradient Decent methods which we call HyLo. HyLo is based on a gradient-based switching approach

that decides when to use a Khatri-Rao-based interpolative decomposition method and when to use importance sampling. As a result, the computation and communication costs associated with computing the Fisher matrix inverse in NGD methods are significantly reduced on distributed platforms. HyLo also supports convolutional neural networks as we extend the SNGD formulation to support CNNs. Our results show that HyLo outperforms state-of-the-art implementations of NGD, such as KFAC, KAISA, EKFAC, KBFGS-L, on both single-GPU and multi-GPU platforms.

Acknowledgements

We would like to thank the Digital Research Alliance of Canada and Scinet (www.alliancecan.ca) for providing us with critical resources to conduct this research. Large-scale experiments of the work would have not been possible without their support. This work was also supported in part by NSERC Discovery Grants (RGPIN-06516, DGECR00303), the Canada Research Chairs program, the Ontario Early Researcher award, the Government of Canada's New Frontiers in Research Fund NFRFE-2019-00503, the ONR N00014-21-1-2244, the U.S. NSF awards CCF-1814888 and DMS-2053485 along with the Extreme Science and Engineering Discovery Environment (XSEDE) [48] which is supported by NSF grant number ACI-1548562.

References

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

[2] N. Ay, "On the locality of the natural gradient for learning in deep bayesian networks," Information Geometry, pp. 1–49, 2020.

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

[4] T. George, C. Laurent, X. Bouthillier, N. Ballas, and P. Vincent, "Fast approximate natural gradient descent in a Kronecker-factored eigenbasis," arXiv preprint arXiv:1806.03884, 2018.

[5] D. Goldfarb, Y. Ren, and A. Bahamou, "Practical quasi-Newton methods for training deep neural networks," arXiv preprint arXiv:2006.08877, 2020.

[6] D. C. Liu and J. Nocedal, "On the limited memory bfgs method for large scale optimization," Mathematical programming, vol. 45, no. 1, pp. 503–528, 1989.

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

[8] J. G. Pauloski, Q. Huang, L. Huang, S. Venkataraman, K. Chard, I. Foster, and Z. Zhang, "Kaisa: an adaptive second-order optimizer framework for deep neural networks," in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2021, pp. 1–14.

[9] J. G. Pauloski, Z. Zhang, L. Huang, W. Xu, and I. T. Foster, "Convolutional neural network training with distributed K-FAC," in SC20: International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE, 2020, pp. 1–12.

[10] S. Shi, L. Zhang, and B. Li, "Accelerating distributed K-FAC with smart parallelism of computing and communication tasks," in 2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS). IEEE, 2021, pp. 550–560.

[11] J. Ba, R. Grosse, and J. Martens, "Distributed second-order optimization using kronecker-factored approximations," 2016.

[12] K. Osawa, Y. Tsuji, Y. Ueno, A. Naruse, R. Yokota, and S. Matsuoka, "Large-scale distributed second-order optimization using kronecker-factored approximate curvature for deep convolutional neural networks," in Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2019, pp. 12 359–12 367.

[13] Y. Ren and D. Goldfarb, "Efficient subsampled Gauss-Newton and natural gradient methods for training neural networks," arXiv preprint arXiv:1906.02353, 2019.

[14] M. Yang, D. Xu, Z. Wen, M. Chen, and P. Xu, "Sketchy empirical natural gradient methods for deep learning," 2020. [Online]. Available: https://arxiv.org/abs/2006.05924

[15] H. Robbins and S. Monro, "A stochastic approximation method," The annals of mathematical statistics, pp. 400–407, 1951.

[16] L. Bottou, "Stochastic gradient descent tricks," in Neural networks: Tricks of the trade. Springer, 2012, pp. 421–436.

[17] F. Kunstner, L. Balles, and P. Hennig, "Limitations of the empiri-cal fisher approximation for natural gradient descent," arXiv preprint arXiv:1905.12558, 2019.

[18] T. George, C. Laurent, X. Bouthillier, N. Ballas, and P. Vincent, "Fast approximate natural gradient descent in a kronecker factored eigenbasis," in Advances in Neural Information Processing Systems, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., vol. 31. Curran Associates, Inc., 2018. [Online]. Available: https://proceedings.neurips.cc/paper/2018/file/48000647b315f6f00f913caa757a70b3-Paper.pdf

[19] G. H. Golub and C. F. Van Loan, Matrix computations. JHU press, 2013.

[20] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014. [Online]. Available: https://arxiv.org/abs/1412.6980

[21] P. Drineas and M. W. Mahoney, "Randnla: randomized numerical linear algebra," Communications of the ACM, vol. 59, no. 6, pp. 80–90, 2016.

[22] S. Agarwal, H. Wang, K. Lee, S. Venkataraman, and D. Papailiopou-los, "Accordion: Adaptive gradient communication via critical learning regime identification," arXiv preprint arXiv:2010.16248, 2020.

[23] D. Goldfarb, Y. Ren, and A. Bahamou, "Practical quasi-newton methods for training deep neural networks," 2020. [Online]. Available: https://arxiv.org/abs/2006.08877

[24] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in Advances in Neural Information Processing Systems, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds., vol. 25. Curran Associates, Inc., 2012. [Online]. Available: https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

[25] A. Krizhevsky, "Learning multiple layers of features from tiny images," Tech. Rep., 2009.

[26] M. Buda, "Brain mri segmentation," May 2019. [Online]. Available: https://www.kaggle.com/datasets/mateuszbuda/lgg-mri-segmentation

[27] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms," arXiv preprint arXiv:1708.07747, 2017.

[28] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou, R. Chukka, C. Coleman, S. Davis, P. Deng, G. Diamos, J. Duke, D. Fick, J. S. Gardner, I. Hubara, S. Idgunji, T. B. Jablin, J. Jiao, T. S. John, P. Kanwar, D. Lee, J. Liao, A. Lokhmotov, F. Massa, P. Meng, P. Micikevicius, C. Osborne, G. Pekhimenko, A. T. R. Rajan, D. Sequeira, A. Sirasao, F. Sun, H. Tang, M. Thomson, F. Wei, E. Wu, L. Xu, K. Yamada, B. Yu, G. Yuan, A. Zhong, P. Zhang, and Y. Zhou, "Mlperf inference benchmark," 2019. [Online]. Available: https://arxiv.org/abs/1911.02549

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

[30] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," 2016. [Online]. Available: https://arxiv.org/abs/1608.06993

[31] "Compute Canada," www.computecanada.ca.

[32] P. Drineas, R. Kannan, and M. W. Mahoney, "Fast monte carlo algorithms for matrices i: Approximating matrix multiplication," SIAM Journal on Computing, vol. 36, no. 1, pp. 132–157, 2006.

[33] D. J. Biagioni, D. Beylkin, and G. Beylkin, "Randomized interpolative decomposition of separated representations," Journal of Computational Physics, vol. 281, pp. 116–134, 2015.

[34] S.-i. Amari, "Neural learning in structured parameter spaces - natural riemannian gradient," in Advances in Neural Information Processing Systems, M. C. Mozer, M. Jordan, and T. Petsche, Eds., vol. 9. MIT Press, 1997. [Online]. Available: https://proceedings.neurips.cc/paper/1996/file/39e4973ba3321b80f37d9b55f63ed8b8-Paper.pdf

[35] T. Cai, "A Gram-Gauss-Newton method learning overparameterized deep neural networks for regression problems," Machine learning, no. 1/28, 2019.

[36] R. Karakida and K. Osawa, "Understanding approximate Fisher information for fast convergence of natural gradient descent in wide neural networks," in Advances in Neural Information Processing Systems, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, Eds., vol. 33. Curran Associates, Inc., 2020, pp. 10 891–10 901. [Online]. Available: https://proceedings.neurips.cc/paper/2020/file/7b41bfa5085806dfa24b8c9de0ce567f-Paper.pdf

[37] G. Zhang, J. Martens, and R. B. Grosse, "Fast convergence of natural gradient descent for over-parameterized neural networks," in Advances in Neural Information Processing Systems, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32. Curran Associates, Inc., 2019. [Online]. Available: https://proceedings.neurips.cc/paper/2019/file/1da546f25222c1ee710cf7e2f7a3ff0c-Paper.pdf

[38] F. Dangel, F. Kunstner, and P. Hennig, "Backpack: Packing more into backprop," in International Conference on Learning Representations, 2019.

[39] F. Kunstner, P. Hennig, and L. Balles, "Limitations of the empirical Fisher approximation for natural gradient descent," in Advances in Neural Information Processing Systems, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32. Curran Associates, Inc., 2019. [Online]. Available: https://proceedings.neurips.cc/paper/2019/file/46a558d97954d0692411c861cf78ef79-Paper.pdf

[40] R. Grosse and J. Martens, "A Kronecker-factored approximate Fisher matrix for convolution layers," in International Conference on Machine Learning. PMLR, 2016, pp. 573–582.

[41] T. George, C. Laurent, X. Bouthillier, N. Ballas, and P. Vincent, "Fast approximate natural gradient descent in a kronecker factored eigenbasis," in Advances in Neural Information Processing Systems, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., vol. 31. Curran Associates, Inc., 2018. [Online]. Available: https://proceedings.neurips.cc/paper/2018/file/48000647b315f6f00f913caa757a70b3-Paper.pdf

[42] D. Goldfarb, Y. Ren, and A. Bahamou, "Practical Quasi-Newton methods for training deep neural networks," in Advances in Neural Information Processing Systems, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, Eds., vol. 33. Curran Associates, Inc., 2020, pp. 2386–2396. [Online]. Available: https://proceedings.neurips.cc/paper/2020/file/192fc044e74dffea144f9ac5dc9f3395-Paper.pdf

[43] A. Botev, H. Ritter, and D. Barber, "Practical Gauss-Newton optimisation for deep learning," in International Conference on Machine Learning. PMLR, 2017, pp. 557–565.

[44] T. Heskes, "On "natural" learning and pruning in multilayered perceptrons," Neural Computation, vol. 12, no. 4, pp. 881–901, 2000.

[45] G. Desjardins, K. Simonyan, R. Pascanu, and k. kavukcuoglu, "Natural neural networks," in Advances in Neural Information Processing Systems, C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, Eds., vol. 28. Curran Associates, Inc., 2015. [Online]. Available: https://proceedings.neurips.cc/paper/2015/file/2de5d16682c3c35007e4e92982f1a2ba-Paper.pdf

[46] S. P. Singh and D. Alistarh, "Woodfisher: Efficient second-order approximation for neural network compression," Advances in Neural Information Processing Systems, vol. 33, 2020.

[47] L. Ma, G. Montague, J. Ye, Z. Yao, A. Gholami, K. Keutzer, and M. Mahoney, "Inefficiency of K-FAC for large batch size training," in Proceedings of the AAAI Conference on Artificial Intelligence, vol. 34, no. 04, 2020, pp. 5053–5060.

[48] J. Towns, T. Cockerill, M. Dahan, I. Foster, K. Gaither, A. Grimshaw, V. Hazlewood, S. Lathrop, D. Lifka, G. D. Peterson, R. Roskies, J. R. Scott, and N. Wilkins-Diehr, "Xsede: Accelerating scientific discovery," Computing in Science & Engineering, vol. 16, no. 5, pp. 62–74, Sept.-Oct. 2014. [Online]. Available: doi.ieeecomputersociety.org/10.1109/MCSE.2014.80

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

1 ABSTRACT

This artifact provides information to reproduce results shown in the paper: "HyLo: A Hybrid Low-Rank Natural Gradient Descent Method". We explain how to run HyLo and compare it with KAISA and SGD/ADAM. We provide one repository for HyLo and use the code provided by KAISA for comparisons.

2 DESCRIPTION

Checklist (artifact meta-information)

(1) Algorithm: HyLo, KAISA, SGD and ADAM
(2) Program: Python
(3) Datasets: ImageNet-1k, LGG MRI, CIFAR-10, CIFAR-100, Fashion-MNIST
(4) Run-time environment: Linux
(5) Experiment workflow: Download/Install HyLo and KAISA. Download the datasets. Run the test scripts.

3 SOFTWARE DETAILS

The software version (in Anaconda) for all the experiments is as follows:

(1) python==3.7.11
(2) pytorch==1.7.1
(3) cudatoolkit==10.2.89
(4) cudnn==7.6.5
(5) torchinfo==1.6.5
(6) tensorboard==2.4.1
(7) torchvision==0.8.2
(8) matplotlib==3.5.1
(9) scikit-image==0.18.3
(10) medpy==0.3.0

4 HARDWARE DETAILS

We use three systems for the experiments. SciNet Mist supercomputers and AWS P2 and P3 systems.

(1) Mist: each node has 32 IBM Power9 cores with 256GB RAM and 4 NVIDIA V100 GPUs with 32GB memory, NVLink in between.
(2) AWS-P2: we use P2.8xlarge systems which each node has 8 K80 GPUs with 12 GB memory.
(3) AWS-P3: we use P3.16xlarge equipped with 8 NVIDIA V100 with 32 GB memory connected via NVLink.

5 DATASETS

All the datasets are publicly available.

(1) ImageNet-1k is available from ILSVRC2012: https://image-net.org/challenges/LSVRC/2012/2012-downloads.php.
(2) LGG MRI dataset can be downloaded from LGG: https://www.kaggle.com/datasets/mateuszbuda/lgg-mri-segmentation.
(3) CIFAR dataset are available in CIFAR https://www.cs.toronto.edu/ kriz/cifar.html.
(4) Fashion-MNIST is provided in Fashion-MNIST https://github.com/zalandoresearch/fashion-mnist.

6 EXPERIMENTS

(1) accuracy and time-to-convergence: For the single-GPU setting, we compare the time-to-convergence and accuracy of HyLo to KFAC, EKFAC, KBFGS-L, SGD, ADAM on 1 V100 GPU for DenseNet and 3C1F. Figure 4 shows the accuracy vs. time. For the multi-GPU setting, we compare the time-to-convergence and accuracy of HyLo to KAISA and SGD. For U-Net model, instead of SGD, ADAM optimizer is used. We conduct this experiment on 16 Mist nodes for ResNet-50, 1 Mist node for U-Net and 4 AWS-P2 nodes for ResNet-32 models. Figure 5 is the accuracy vs time and Figure 6 is the accuracy vs epoch.
(2) profiling of communication-computation: We compare the detailed timing of HyLo to KAISA. We profile the code for detailed timings on AWS-P3 platform. For ResNet-50 model, we use 64 GPUs, for U-Net we use 4 GPUs and for ResNet-32 we use 32 GPUs. Each experiment is run for one epoch and
the average time is reported. The results is reported in Figure 7.
(3) speedup analysis: We use AWS-P3 for ResNet-50 and U-Net and AWS-P2 for ResNet-32. We change the number of GPUs from 4 to 32 for this experiments. We run each method for 3 epoch and project the timing for the the whole training. The speedups are reported in Figure 8.
(4) scalability analysis: We use AWS-P3 for scalability of ResNet-50 and ResNet-32 models. We change the number of GPUs from 1 to 64 for ResNet-50 model and run HyLo for 3 epochs and project the timing for the whole training. For ResNet-32, the number of GPUs are changed from 1 to 32. The scalability is reported in Figure 9.
(5) rank analysis: We report the numerical rank of the kernel matrix on 32 GPUs on Mist for HyLo. For ResNet-50 and ResNet-32, we change the global batch size from 512 to 4096 and report the rank. We report the results in Figure 10.
(6) gradient norm analysis: We compute the gradient norm of different layers for ResNet-32 for end-to-end training and report it in Figure 11.
(7) gradient error analysis: We compute the gradient error of KID and KIS for ResNet-32 and ResNet-50 and report it in Figure 12.
(8) switching method: We compare accuracy and time-to-convergence of HyLo and the random switching on ResNet-50, ResNet-32 and U-Net, as shown in Table III.
(9) memory overhead: We compare memory overhead of HyLo to KAISA, ADAM, SGD with ResNet-50, ResNet-32 and U-Net, as shown in Table IV.

AUTHOR-CREATED OR MODIFIED ARTIFACTS:

Artifact 1

Persistent ID: DOI:10.5281/zenodo.6942342,https://github.com/hylo-dso/hylo

Artifact name: HyLo source code with instructions on GitHub

Artifact 2

Persistent ID: https://cloud.sylabs.io/library/hylo/dso/artifact

Artifact name: Singularity image with required packages

Artifact 3

Persistent ID: DOI:10.5281/zenodo.6910740,https://github.com/hylo-dso/kfac-pytorch

Artifact name: KAISA source code

Reproduction of the artifact with container:

0. Clone the repository (should be on the src branch by default).

```
git clone https://github.com/hylo-dso/hylo.git
cd hylo
```

1. Download the datasets (if not available locally). Please follow the instructions from the oficial websites listed below to download the datasets. The provided scripts assume that the datasets are stored under the directory $SCRATCH and the singularity image (by default, named artifact_init.sif) is downloaded into the directory hylo.

- ImageNet-1k (ILSVRC2012): https://image-net.org/challenges/LSVRC/2012/2012-downloads.php (ImageNet account required)
- LGG MRI: https://www.kaggle.com/datasets/mateuszbuda/lgg-mri-segmentation
- CIFAR: https://www.cs.toronto.edu/~kriz/cifar.html
- Fashion-MNIST: https://github.com/zalandoresearch/fashion-mnist

2. Set up the environment.

- Option 1: use the singularity image (for ppc64le architecture) with all the requirements. To download the singularity image from Sylabs (Sylabs account required):
  If you have not login to singularity on the cluster, generate a token on Sylabs, after running the command below, paste it to the command line interface

  ```
  singularity remote login
  ```

  Then pull the singularity image from remote:

  ```
  singularity pull --arch ppc64le \
  library://hylo/dso/artifact:init
  ```

- Option 2: a list of required packages (can be installed in an Anaconda environment)
  Please see the Software Details Section.

3. Update the scripts. .

The .sh scripts consists of 4 main parts:

a. setup of the environment (see the PRELOAD command),

b. setup of training configurations, e.g. model, dataset, hyperparameters (see the CMD command),

c. distributed system configurations, e.g. the list of allocated nodes, master node, number of nodes

note: the scripts use TCP initialization by default. The url format is tcp://$MASTER_ADDR:$MASTER_PORT, where the MASTER_PORT is set to 1234 by default.

d. launch the processes on each node with torch.distributed.launch (see the LAUNCHER command)

Things you might need to change:

a. If you are NOT using the singularity image, please update the PRELOAD command to set up the environment accordingly. For example, if you choose to install all requirements listed above in a conda environment, say, named env, then the PRELOAD command should be (or similar to):

```
PRELOAD="module load anaconda3;"
PRELOAD+="source activate env;"
```

b. On different number of GPUs, the hyperparameters are different. For example, the frequency `--freq` is scaled inversely with the number of GPUs. The learning rate, damping, target damping, weight decay might need to be adjusted.

c. The provided scripts for ResNet-32 + CIFAR-10 assume that the number of GPUs per node is 4 (`nproc_per_node=4`). Other scripts assume that `nproc_per_node=4`. If the system you are using has a different setting, please update `nproc_per_node` in the CMD and LAUNCHER command accordingly.

4. End-to-End Training

The commands run end-to-end training of the model + dataset. The accuracy and wall-clock time are written to a .csv file in the hylo directory. The checkpoint files (.pth.tar) are stored in the directory specified by the `--log-dir` arg of the main-*.py scripts.

(i) interactive mode:

Example: CIFAR-10 + ResNet-32 (Classification)

```
sh scripts/train-resnet50-imagenet-end-to-end.sh
```

Example: ImageNet-1k + ResNet-50 (Classification)

```
sh scripts/train-resnet32-cifar10-end-to-end.sh
```

Example: Brain LGG + U-Net (Segmentation)

```
sh scripts/train-unet-brain-end-to-end.sh
```

(ii) submit job to compute nodes via sbatch

Example: CIFAR-10 + ResNet-32 (Classification)

```
sbatch -p compute_full_node -N ~ 8 -t ~ 00:30:00 \
--gpus-per-node=4 --ntasks=1024 -J ~ cf-rn32 \
-o cf-rn32.o%j --mail-type=ALL  \
scripts/train-resnet32-cifar10-end-to-end.sh
```

Example: ImageNet-1k + ResNet-50 (Classification)

```
sbatch -p compute_full_node -N ~ 16 -t ~ 2:00:00 \
--gpus-per-node=4 --ntasks=2048 -J ~ img-rn50 \
-o img-rn50.o%j --mail-type=ALL \
scripts/train-resnet50-imagenet-end-to-end.sh
```

Example: Brain LGG + U-Net (Segmentation)

```
sbatch -p compute_full_node -N ~ 1 -t ~ 00:30:00 \
--gpus-per-node=4 --ntasks=128 -J ~ lgg-unet \
-o lgg-unet.o%j --mail-type=ALL \
scripts/train-unet-brain-end-to-end.sh
```

note: for the baselines, we provide sample scripts https://github.com/hylo-dso/kfac-pytorch (based on the oficial release of KAISA 1 and 2).

5. Analysis

To enable profiling, add `--profiling` to the training command, e.g.

```
sh scripts/train-resnet50-imagenet-end-to-end.sh \
--profiling
```

The outputs are written to .csv files.

To enable rank analysis, add --rank-analysis to the training command, e.g.

```
sh scripts/train-resnet32-cifar10-end-to-end.sh \
--rank-analysis
```

The outputs are written to .csv files.

To check the gradient norm trend throughout the training, add `--sngd` and --grad-norm to the training command, e.g.

```
sh scripts/train-resnet32-cifar10-end-to-end.sh \
--sngd --grad-norm
```

The outputs are written to grad-norm.csv.

To check the gradient error of KID and KIS, add --grad-norm and `--enable-id`/`--enable-is` to the training command, e.g.

```
sh scripts/train-resnet50-imagenet-end-to-end.sh \
--grad-error --enable-id
sh scripts/train-resnet50-imagenet-end-to-end.sh \
--grad-error --enable-is
```

The outputs are written to grad-error-*.txt files.