DISTRIBUTED SECOND-ORDER OPTIMIZATION USING KRONECKER-FACTORED APPROXIMATIONS

Jimmy Ba
University of Toronto
jimmy@psi.toronto.edu

Roger Grosse
University of Toronto
rgrosse@cs.toronto.edu

James Martens
University of Toronto
and Google DeepMind
jmartens@cs.toronto.edu

ABSTRACT

As more computational resources become available, machine learning researchers train ever larger neural networks on millions of data points using stochastic gradient descent (SGD). Although SGD scales well in terms of both the size of dataset and the number of parameters of the model, it has rapidly diminishing returns as parallel computing resources increase. Second-order optimization methods have an affinity for well-estimated gradients and large mini-batches, and can therefore benefit much more from parallel computation in principle. Unfortunately, they often employ severe approximations to the curvature matrix in order to scale to large models with millions of parameters, limiting their effectiveness in practice versus well-tuned SGD with momentum. The recently proposed K-FAC method (Martens and Grosse, 2015) uses a stronger and more sophisticated curvature approximation, and has been shown to make much more per-iteration progress than SGD, while only introducing a modest overhead. In this paper, we develop a version of K-FAC that distributes the computation of gradients and additional quantities required by K-FAC across multiple machines, thereby taking advantage of method's superior scaling to large mini-batches and mitigating its additional overheads. We provide a Tensorflow implementation of our approach which is easy to use and can be applied to many existing codebases without modification. Additionally, we develop several algorithmic enhancements to K-FAC which can improve its computational performance for very large models. Finally, we show that our distributed K-FAC method speeds up training of various state-of-the-art ImageNet classification models by a factor of two compared to Batch Normalization (Ioffe and Szegedy, 2015).

1 INTRODUCTION

Current state-of-the-art deep neural networks (Szegedy et al., 2014; Krizhevsky et al., 2012; He et al., 2015) often require days of training time with millions of training cases. The typical strategy to speed-up neural network training is to allocate more parallel resources over many machines and cluster nodes (Dean et al., 2012). Parallel training also enables researchers to build larger models where different machines compute different splits of the mini-batches. Although we have improved our distributed training setups over the years, neural networks are still trained with various simple first-order stochastic gradient descent (SGD) algorithms. Despite how well SGD scales with the size of the model and the size of the datasets, it does not scale well with the parallel computation resources. Larger mini-batches and more parallel computations exhibit diminishing returns for SGD and related algorithms.

Second-order optimization methods, which use second-order information to construct updates that accounts for the curvature of objective function, represent a promising alternative. The canonical second-order methods work by inverting a large curvature matrix (traditionally the Hessian), but this doesn't scale well to deep neural networks with millions of parameters. Various approximations to the curvature matrix have been proposed to help alleviate this problem, such as diagonal (LeCun et al., 1998; Kingma and Ba, 2014), block diagonal Le Roux et al. (2008), and low-rank ones (Moritz et al., 2016; Byrd et al., 2016). Another strategy is to use Krylov-subspace methods and efficient

matrix-vector product algorthms to avoid the inversion problem entirely (Martens, 2010; Vinyals and Povey, 2012).

The usual problem with curvature approximations, especially low-rank and diagonal ones, is that they are very crude and only model superficial aspects of the true curvature in the objective function. Krylov-subspace methods on the other hand suffer because they still rely on 1st-order methods to compute their updates.

More recently, several approximations have been proposed based on statistical approximations of the Fisher information matrix (Heskes, 2000; Grosse and Salakhutdinov, 2015; Povey et al., 2015; Desjardins et al., 2015). In the K-FAC approach (Martens and Grosse, 2015; Grosse and Martens, 2016), these approximations result in a block-diagonal approximation to the Fisher information matrix (with blocks corresponding to entire layers) where each block is approximated as a Kronecker product of two much smaller matrices, both of which can be estimated and inverted fairly efficiently. Because the inverse of a Kronecker product of two matrices is the Kronecker product of their inverse, this allows the entire matrix to be inverted efficiently.

Martens and Grosse (2015) found that K-FAC scales very favorably to larger mini-batches compared to SGD, enjoying a nearly linear relationship between mini-batch size and per-iteration progress for medium-to-large sized mini-batches. One possible explanation for this phenomenon is that second-order methods make more rapid progress exploring the error surface and reaching a neighborhood of a local minimum where gradient noise (which is inversely proportional to mini-batch size) becomes the chief limiting factor in convergence[1]. This observation implies that K-FAC would benefit in particular from a highly parallel distributed implementation.

In this paper, we propose an asynchronous distributed version of K-FAC that can effectively exploit large amounts of parallel computing resources, and which scales to industrial-scale neural net models with hunderd millions of parameters. Our method augments the traditional distributed synchronous SGD setup with additional computation nodes that update the approximate Fisher and compute its inverse. The proposed method achieves a comparable per-update runtime as a normal SGD using the same mini-batch size on a typical 4 GPU cluster. We also propose a "doubly factored" Kronecker approximation for layers whose inputs are feature maps that are normally too large to handled by the standard Kronecker-factored approximation. Finally, we empirically demonstrate that the proposed method speeds up learning of various state-of-the-art ImageNet models by a factor of two over Batch Normalization (Ioffe and Szegedy, 2015).

2 BACKGROUND

2.1 KRONECKER FACTORED APPROXIMATE FISHER

Let $\mathcal{D}W$ be the gradient of the log likelihood $\mathcal{L}$ of a neural network w.r.t. some weight matrix $W \in \mathbb{R}^{C_{in} \times C_{out}}$ in a layer, where C_{in}, C_{out} are the number of input/output units of the layer. The block of the Fisher information matrix of that layer is given by:

$$F = \underset{\mathbf{x}, y \sim P}{\mathbb{E}} \left[\text{vec}\{\mathcal{D}W\} \, \text{vec}\{\mathcal{D}W\}^\top \right], \tag{1}$$

where P is distribution over the input $\mathbf{x}$ and the network's distribution over targets y (implied by the log-likelihood objective). Throughout this paper we assume, unless otherwise stated, that expectations are taken with respect to P (and not the training distribution over y).

K-FAC (Martens and Grosse, 2015; Grosse and Martens, 2016) uses an Kronecker-factored approximation to each block which we now describe. Denote the input activation vector to the layer as $\mathcal{A} \in \mathbb{R}^{C_{in}}$, the pre-activation inputs as $s = W\mathcal{A}$ and the back-propagated loss derivativesas $\mathcal{D}s = \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}s} \in \mathbb{R}^{C_{out}}$. Note that the gradients of the weights is the outer product of the input activation and back-propagated derivatives $\mathcal{D}W = \mathcal{A}\mathcal{D}s^\top$. K-FAC approximates the Fisher block as a

[1]Mathematical evidence for this idea can be found in Martens (2014), where it is shown that (convex quadratic) objective functions decompose into noise-dependent and independent terms, and that second-order methods make much more rapid progress optimizing the noise-independent term compared to SGD, while have no effect on the noise-dependent term (which shrinks with the size of the mini-batch)

Kronecker product of the second-order statistics of the input and the backpropagated derivatives:

$$F = \mathbb{E}\left[\text{vec}\{\mathcal{D}W\}\,\text{vec}\{\mathcal{D}W\}^\top\right] = \mathbb{E}\left[\mathcal{A}\mathcal{A}^\top \otimes \mathcal{D}s\mathcal{D}s^\top\right] \approx \mathbb{E}\left[\mathcal{A}\mathcal{A}^\top\right] \otimes \mathbb{E}\left[\mathcal{D}s\mathcal{D}s^\top\right] \triangleq \hat{F}. \quad (2)$$

This approximation can be interpreted as making the assumption that the second-order statistics of the activations and the backpropagated derivatives are uncorrelated.

2.2 Approximate Natural Gradient using K-FAC

The natural gradient (Amari, 1998) is defined as the inverse of the Fisher times the gradient. It is traditionally interpetted as the direction in parameter space that achieves the largest (instantaneous) improvement in the objective per unit of change in the output distribution of the network (as measured using the KL-divergence). Under certain conditions, which almost always hold in practice, it can also be interpreted as a second-order update computed by minimizing a local quadratic approximation of the log-likelihood objective, where the Hessian is approximated using the Fisher (Martens, 2014).

To compute the approximate natural gradient in K-FAC, one multiplies the gradient for the weights of each layer by the inverse of the corresponding approximate Fisher block $\hat{F}$ for that layer. Denote the gradient of the loss function with respect to the weights W by $\mathcal{G}_W \in \mathbb{R}^{C_{in} \times C_{out}}$. We will assume the use of the factorized Tikhonov damping approach described by Martens and Grosse (2015), where the addition of the damping term λI to $\hat{F}$ is approximated by adding $\pi_{\mathcal{A}}\lambda^{\frac{1}{2}} I$ to $\mathbb{E}\left[\mathcal{A}\mathcal{A}^\top\right]$ and $\pi_{\mathcal{D}s}\lambda^{\frac{1}{2}} I$ to $\mathbb{E}\left[\mathcal{D}s\mathcal{D}s^\top\right]$, where $\pi_{\mathcal{A}}$ and $\pi_{\mathcal{D}s}$ are adjustment factors that are described in detail and generalized in Sec. 4.1. (Note that one can also include the contribution to the curvature from any L2 regularization terms with λ.)

By exploiting the basic identities $(A \otimes B)^{-1} = (A^{-1} \otimes B^{-1})$ and $(A \otimes B)\,\text{vec}(C) = \text{vec}(BCA^\top)$, the approximate natural gradient update $\mathbf{v}$ can then be computed as:

$$\mathbf{v} = \left(\hat{F} + \lambda I\right)^{-1} \text{vec}\{\mathcal{G}_W\} \approx \text{vec}\left\{\left(\mathbb{E}\left[\mathcal{A}\mathcal{A}^\top\right] + \pi_{\mathcal{A}}\lambda^{\frac{1}{2}} I\right)^{-1} \mathcal{G}_W \left(\mathbb{E}\left[\mathcal{D}s\mathcal{D}s^\top\right] + \pi_{\mathcal{D}s}\lambda^{\frac{1}{2}} I\right)^{-1}\right\}, \quad (3)$$

which amounts of several matrix inversion of multiplication operations involving matrices roughly the same size as the weight matrix W.

3 Distributed Optimization using K-FAC

Stochastic optimization algorithms benefit from low-variance gradient estimates (as might be obtained from larger mini-batches). Prior work suggests that approximate natural gradient algorithms might benefit more than standard SGD from reducing the variance (Martens and Grosse, 2015; Grosse and Martens, 2016). One way to efficiently obtain low-variance gradient estimates is to parallelize the gradient computation across many machines in a distributed system (thus allowing large mini-batches to be processed efficiently). Because the gradient computation in K-FAC is identical to that of SGD, we parallelize the gradient computation using the standard synchronous SGD model.

However, K-FAC also introduces other forms of overhead not found in SGD — in particular, estimation of second-order statistics and computation of inverses or eigenvalues of the Kronecker factors. In this section, we describe how these additional computations can be performed asynchronously. While this asynchronous computation introduces an additional source of error into the algorithm, we find that it does not significantly affect the per-update progress in practice. All in all, the per-update wall clock time of our distributed K-FAC implementation is only 5-10% higher compared to synchronous SGD with the same mini-batch size.

3.1 Asynchronous eigendecomposition

Computing approximate natural gradient updates in Eq.3 requires the estimated gradients to be multiplied by the inverse of the smaller Kronecker factors. This requires periodically computing either inverses or eigendecompositions of each of these factors. While these factors typically have sizes only in the hundreds or low thousands for most neural networks, a given network may have hundreds

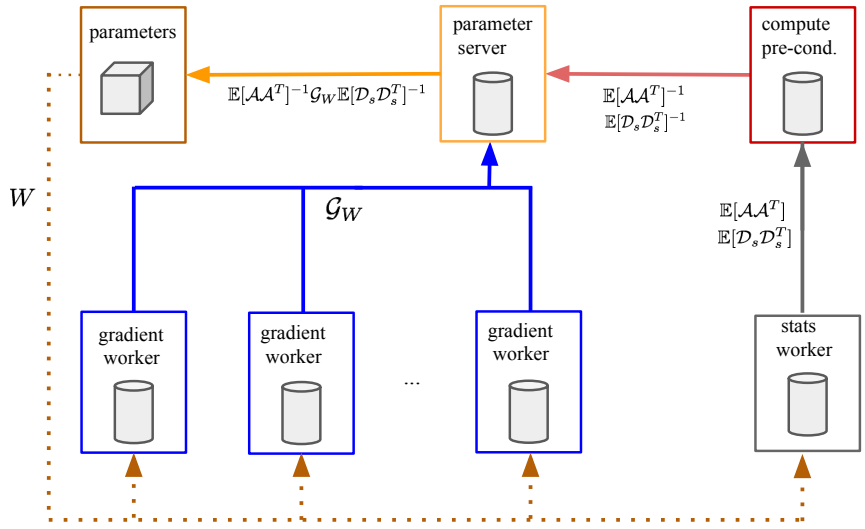

Figure 1: The diagram illustrate the distributed computation of K-FAC. Gradient workers (blue) computes the gradient w.r.t. the loss function. Stats workers (grey) computes the sampled second-order statistics. Additional workers compute inverse Fisher blocks via either eigendecomposition or matrix inversion are shown in red. The parameter server (orange) uses gradients and their inverse Fisher blocks to compute parameter updates.

of such matrices. Furthermore, matrix inversion and eigendecomposition see little benefit from GPU computation, so they can be more expensive than standard neural network operations. For these reasons, computing inverses or eigendecompositions represents a significant computational cost.

It has been observed that the eigendecompositions can be computed with a significant lag (potentially hundreds of iterations), perhaps because the curvature changes relatively slowly (Martens and Grosse, 2015). We push this a step further by computing the eigendecompositions *asynchronously* while the network is still training. Because the eigendecompositions are CPU-bound while the rest of our computations are GPU-bound, we perform the eigendecompositions on the CPU with little effective overhead. Our curvature statistics are somewhat more stale as a result, but this does not appear to significantly affect per-update optimization performance. In our experiments, we found that asynchronous eigenvalue computation usually offered a 40-50% speed-up to the overall wall-clock of the K-FAC algorithm.

3.2 Asynchronous statistics computation

The other source of computation is the estimation of the second-order statistics of the activations and derivatives, which are needed for the Kronecker factors. In the standard K-FAC algorithm, these statistics are computed on the same mini-batches as the gradients, allowing the

In cases where it is undesirable to devote separate worker nodes to computing statistics, we also introduce a fast approximation to the statistics for convolution layers (see Appendix A).

4 Doubly-factored Kronecker approximation for large convolution layers

Computing the standard Kronecker factored Fisher approximation for a given layer involves operations on matrices whose dimension is the number of input units or output units. The cost of these operations is reasonable for most fully-connected networks because the number of units in each layer rarely exceeds a couple thousand. Large convolutional neural networks, however, often include a fully-connected layer that "pools" over a large feature map before the final softmax classification. For instance, the output of the last pooling layer of AlexNet is of size $6 \times 6 \times 256 = 9216$, which then provides inputs to the subsequent fully connected layer of 4096 ReLUs. VGG models also

share a similar architecture. For the standard Kronecker-factored approximation one of the factors will be a matrix of size 9216×9216, which is too expensive to be expicitly inverted as often as is needed during training.

In this section we propose a "doubly-factored" Kronecker approximation for layers whose input is a large feature map. Specifically, we approximate the second-order statistics matrix of the inputs as itself factoring as a Kronecker product. This gives an approximation which is a Kronecker product of three matrices.

Using the AlexNet example, the 9216×4096 weight matrix in the first fully connected layer is equivalent to a filterbank of 4096 filters with kernel size 6×6 on 256 input channels. Let A be a matrix of dimension $\mathcal{T}$-by-C_{in} representing the input activations (for a single training case), where $\mathcal{T} = K_w \times K_h$ is the feature map height and width, and C_{in} is the number of input channels. The Fisher block for such a layer can be written as:

$$\mathbb{E}[\mathrm{vec}\{\mathcal{D}W\}\,\mathrm{vec}\{\mathcal{D}W\}^\top] = \mathbb{E}[\mathrm{vec}\{A\}\,\mathrm{vec}\{A\}^\top \otimes \mathcal{D}s\mathcal{D}s^\top], \quad A \in \mathbb{R}^{\mathcal{T} \times C_{in}}. \tag{4}$$

We begin be making the following rank-1 approximation:

$$A \approx \mathcal{K}\Psi^\top, \tag{5}$$

where $\mathcal{K} \in \mathbb{R}^{\mathcal{T}}$, $\Psi \in \mathbb{R}^{C_{in}}$ are the factors along the spatial location dimension and the input channel dimension. The optimal solution of a low-rank approximation under the Frobenius norm is given by the singular value decomposition. The activation matrix A is small enough that its SVD can be computed efficiently. Let σ_1, u_1, v_1 be the first singular value and its left and right singular vectors of the activation matrix A. The factors of rank-1 approximation are then chosen to be $\mathcal{K} = \sqrt{\sigma_1}u_1$ and $\Psi = \sqrt{\sigma_1}v_1$. $\mathcal{K}$ captures the activation patterns across spatial locations in a feature map and Ψ is captures the pattern across the filter responses. Under the rank-1 approximation of A we have:

$$\mathbb{E}[\mathrm{vec}\{A\}\,\mathrm{vec}\{A\}^\top \otimes \mathcal{D}s\mathcal{D}s^\top] \approx \mathbb{E}[\mathrm{vec}\{\mathcal{K}\Psi^\top\}\,\mathrm{vec}\{\mathcal{K}\Psi^\top\}^\top \otimes \mathcal{D}s\mathcal{D}s^\top] \tag{6}$$

$$= \mathbb{E}[\mathcal{K}\mathcal{K}^\top \otimes \Psi\Psi^\top \otimes \mathcal{D}s\mathcal{D}s^\top]. \tag{7}$$

We further assume the second order statistics are three-way independent between the loss derivatives $\mathcal{D}s$, the activations along the input channels Ψ, and the activations along spatial locations $\mathcal{K}$:

$$\mathbb{E}[\mathrm{vec}\{\mathcal{D}W\}\,\mathrm{vec}\{\mathcal{D}W\}^\top] \approx \mathbb{E}[\mathcal{K}\mathcal{K}^\top] \otimes \mathbb{E}[\Psi\Psi^\top] \otimes \mathbb{E}[\mathcal{D}s\mathcal{D}s^\top]. \tag{8}$$

The final approximated Fisher block is a Kronecker product of three small matrices. And note that although we assumed the feature map activations have low-rank structure, the resulting approximated Fisher is not low-rank.

The approximate natural gradient for this layer can then be computed by multiplying the inverses of the each smaller matrices against the respective dimensions of the gradient tensor. We define a function $\mathcal{R}_i : \mathbb{R}^{d_1 \times d_2 \times d_3} \to \mathbb{R}^{d_j d_k \times d_i}$ that constructs a matrix from a 3D tensor by "reshaping" it so that the desired target dimension $i \in \{1, 2, 3\}$ maps to columns, while the remaining dimensions (j and k) are "folded together" and map to the rows. Given the gradient of the weights, $\mathcal{G}_W \in \mathbb{R}^{\mathcal{T} \times C_{in} \times C_{out}}$ we can compute the matrix-vector product with the double-factored Kronecker approximated Fisher block as:

$$\mathcal{R}_3^{-1}\left(\mathbb{E}[\mathcal{D}s\mathcal{D}s^\top]^{-1}\mathcal{R}_3\left(\mathcal{R}_2^{-1}\left(\mathbb{E}[\Psi\Psi^\top]^{-1}\mathcal{R}_2(\mathcal{R}_1^{-1}(\mathbb{E}[\mathcal{K}\mathcal{K}^\top]^{-1}\mathcal{R}_1(\mathcal{G}_W))))\right)\right). \tag{9}$$

which is a nested applications of the reshape function $\mathcal{R}(\cdot)$ at each of the dimension of the gradient tensor.

The doubly factored Kronecker approximation provides a computational feasible alternative to the standard Kronecker-factored approximation for layers that have a number of parameters in the order of hundreds of millions. For example, inverting it for the first fully connected layer of AlexNet takes about 15 seconds on an 8 core Intel Xeon CPU, and such time is amortized in our asynchronous algorithm.

Unfortunately, the homogeneous coordinate formulation is no longer applicable under this new approximation. Instead, we lump the bias parameters together and associate a full Fisher block with them, which can be explicitly computed and inverted since the number of bias parameters per layer is small.

4.1 FACTORED TIKHONOV DAMPING FOR THE DOUBLE-FACTORED KRONECKER APPROXIMATION

In second-order optimization methods, "damping" performs the crucial task of correcting for the inaccuracies of the local quadratic approximation of the objective that is (perhaps implicitly) optimized when computing the update (Martens and Sutskever, 2012; Martens, 2014, e.g.). In the well-known Tikhonov damping/regularization approach, one adds a multiple of the identity λI to the Fisher before inverting it (as one also does for L2-regularization / weight-decay), which roughly corresponds to imposing a spherical trust-region on the update.

The inverse of a Kronecker product can be computed efficiently as the Kronecker product of the inverse of its factors. Adding a multiple of the identity complicates this computation (although it can still be performed tractably using eigendecompositions). The "factored Tikhonov damping" technique proposed in (Martens and Grosse, 2015) is appealing because it preserves the Kronecker structure of the factorization and thus the inverse can still be computed by inverting each of the smaller matrices (and avoiding the more expensive eigendecomposition operation). And in our experiments with large ImageNet models, we also observe the factored damping seems to perform better in practice. In this subsection we derive a generalized version of factored Tikhonov damping for the double-factored Kronecker approximation.

Suppose we wish to add λI to our approximate Fisher block $A \otimes B \otimes C$. In the factored Tikhonov scheme this is approximated by adding $\pi_a \lambda^{\frac{1}{3}} I$, $\pi_b \lambda^{\frac{1}{3}} I$, and $\pi_c \lambda^{\frac{1}{3}} I$ to A, B and C respectively, for non-negative scalars π_a, π_b and π_c satisfying $\pi_a \pi_b \pi_c = 1$. The error associated with this approximation is:

$$(A + \pi_a \lambda^{\frac{1}{3}} I) \otimes (B + \pi_b \lambda^{\frac{1}{3}} I) \otimes (C + \pi_c \lambda^{\frac{1}{3}} I) - (A \otimes B \otimes C + \lambda I) \tag{10}$$

$$\begin{aligned} =& \pi_c \lambda^{\frac{1}{3}} I \otimes A \otimes B + \pi_b \lambda^{\frac{1}{3}} I \otimes A \otimes C + \pi_a \lambda^{\frac{1}{3}} I \otimes B \otimes C \\ &+ \pi_c \lambda^{\frac{i}{3}} I \otimes \pi_b \lambda^{\frac{1}{3}} I \otimes A + \pi_c \lambda^{\frac{1}{3}} I \otimes \pi_a \lambda^{\frac{1}{3}} I \otimes B + \pi_a \lambda^{\frac{1}{3}} I \otimes \pi_b \lambda^{\frac{1}{3}} I \otimes C \end{aligned} \tag{11}$$

Following Martens and Grosse (2015), we choose π_a, π_b and π_c by taking the nuclear norm in Eq. 11 and minimizing its triangle inequality-derived upper-bound. Note that the nuclear norm of Kronecker products is the product of the nuclear norms of each individual matrices: $\|A \otimes B\|_* = \|A\|_* \|B\|_*$. This gives the following formula for the value of π_a

$$\pi_a = \sqrt[3]{\left(\frac{\|A\|_*}{d_A}\right)^2 \left(\frac{\|B\|_*}{d_B} \frac{\|C\|_*}{d_C}\right)^{-1}}. \tag{12}$$

where the d's are the number of rows (equiv. columns) of the corresponding Kronecker factor matrices. The corresponding formulae for π_b and π_c are analogous. Intuitively, the Eq. 12 rescales the contribution to each factor matrix according the geometric mean of the ratio of its norm vs the norms of the other factor matrices. This results in the contribution being upscaled if the factor's norm is larger than averaged norm, for example. Note that this formula generalizes to Kronecker products of arbitrary numbers of matrices as the geometric mean of the norm ratios.

5 STEP SIZE SELECTION

Although Grosse and Martens (2016) found that Polyak averaging (Polyak and Juditsky, 1992) obviated the need for tuning learning rate schedules on some problems, we observed the choice of learning rate schedules to be an important factor in our ImageNet experiments (perhaps due to higher stochasticity in the updates). On ImageNet, it is common to use a fixed exponential decay schedule (Szegedy et al., 2014; 2015). As an alternative to learning rate schedules, we instead use curvature information to control the amount by which the predictive distribution is allowed to change after each update. In particular, given a parameter update vector $\mathbf{v}$, the second-order Taylor approximation to the KL divergence between the predictive distributions before and after the update is given by the (squared) Fisher norm:

$$\mathrm{D}_{\mathrm{KL}}[q||p] \approx \frac{1}{2} \mathbf{v}^\top F \mathbf{v} \tag{13}$$

This quantity can be computed with a curvature-vector product (Schraudolph, 2002). Observe that choosing a step size of η will produce an update with squared Fisher norm $\eta^2 \, \mathbf{v}^\top F \mathbf{v}$. Instead of

using a schedule learning rate schedule, we choose η_k in each iteration such that the squared Fisher norm is at most some value c_k:

$$\eta_k = \min\left(\eta_{\max}, \sqrt{\frac{c_k}{\mathbf{v}_k^\top F \mathbf{v}_k}}\right) \tag{14}$$

Grosse and Martens (2016) used this method to clip updates at the start of training, but we found it useful to use it throughout training. We use an exponential decay schedule $c_k = c_0 \lambda^k$, where c_0 and λ are tunable parameters. Shrinking the maximum changes in the model prediction after each update is analogous to shrink the trust region of the 2-nd order optimization. In practice, computing curvature-vector products after every update introduces significant computational overhead, so we instead used the approximate Fisher $\hat{F}$ in place of F, which allows the approximate Fisher norm to be computed efficiently as $\mathbf{v}^\top \hat{F} \mathbf{v} = \mathbf{v}^\top \mathcal{G}_W$. The maximum step size $\eta_{\max}$ was set to a large value, and in practice this maximum was reached only at the beginning of training, when F was small in magnitude.

6 EXPERIMENTS

To show the effectiveness of asynchronous distributed training, we first evaluate the performance of distributed K-FAC on CIFAR-10 comparing to the synchronous K-FAC implementation. K-FAC is used in those experiments to pre-condition an SGD optimizer with momentum of 0.9. An exponentially decayed Polyak averaging (Polyak and Juditsky, 1992) is applied to the model parameters (in addition to the means and the variances from batch normalization) for all experiments. We used the similar model and experiment setup from (Grosse and Martens, 2016). In our ImageNet experiments, faster Kronecker factorization from Appendix A and KL based learning rate schedule are applied. Lastly, we compare the scalability of distributed K-FAC and SGD for large mini-batches and more computation resources.

Constrained by resources, we uses a single GPU server to simulate a large distributed system. Some of the GPUs are used as gradient workers that computes the gradient of a large mini-batches while the CPUs acts as a parameter server. The asynchronous eigendecompostion are computed on the local CPUs of the server. Optionally, second-order statistics can be computed on some other GPUs asynchronously that is different from the gradient workers in the GPU server.

We chose to base our distributed K-FAC implementation on the TensorFlow framework (Abadi et al., 2016) because it provides well-engineered and scalable primitives for distributed computation. The K-FAC approximations can be derived automatically using TensorFlow's computation graph formalism, allowing our optimization code to be shared between all network architectures we consider; see Appendix B for details.

6.1 ANALYZE ASYNCHRONOUS UPDATES ON CIFAR-10

To verify our claim that computing eigendecomposition asychronously does not degrade the per update progress, we performed some experiments on the smaller model. Our CIFAR-10 model has 3 convolutional layers of 32-32-64 filters with receptive field size of 5x5 followed by a softmax layer that predicts 10 classes. We first compares the synchronous K-FAC with distributed K-FAC using a single gradient worker. The difference of the two model is that the eigenvaluedecomposition is computed asynchronously for the proposed distributed K-FAC algorithm. In Fig. 2, distributed K-FAC is 1.5 times faster than the synchronous K-FAC while its per-update progress is comparable to the synchronous implementation. As the number of gradient workers increases to 4, the distributed K-FAC reaches 19% test error in 1 minute. Even though the pre-conditioners in distributed K-FAC are computed from delayed statistics. we observe in the top row plots from Fig. 2, such delayed pre-conditioner has little effect on the convergence speed of the approximate natural gradient algorithm.

6.2 IMAGENET CLASSIFICATION

We uses ImageNet 2012 classification dataset (Russakovsky et al., 2015) to evaluate distributed K-FAC. ImageNet dataset consists of 1.28 million training images from 1000 classes. We apply distributed K-FAC to the following off-the-shelf ImageNet models: AlexNet (Krizhevsky et al.,

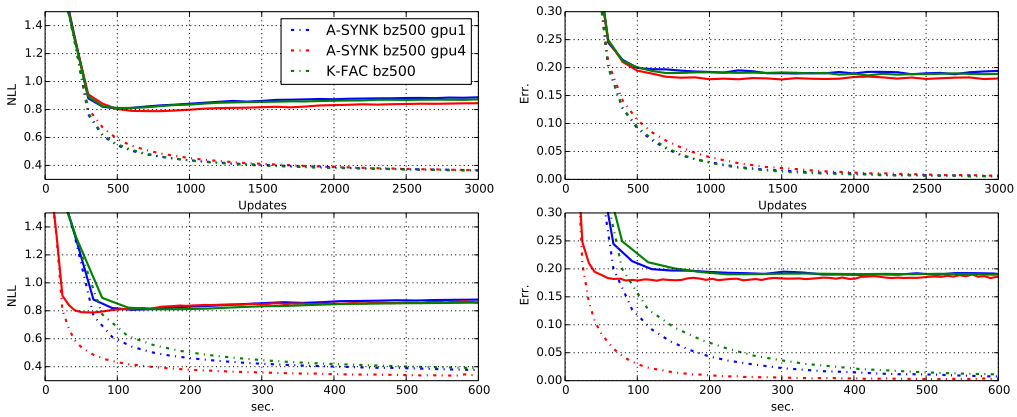

Figure 2: Optimization performance of distributed K-FAC, K-FAC on CIFAR-10. *bz* represents the size of mini-batches and the *gpu* number denotes the number of gradient workers for distributed K-FAC. Dashed lines denote training curves and solid lines denote test curves. Top row: cross entropy loss and validation error v.s. the number of updates. Bottom row: cross entropy loss and validation error v.s. wallclock time (in hours). The models are trained using 4 of Nvidia K80 GPUs.

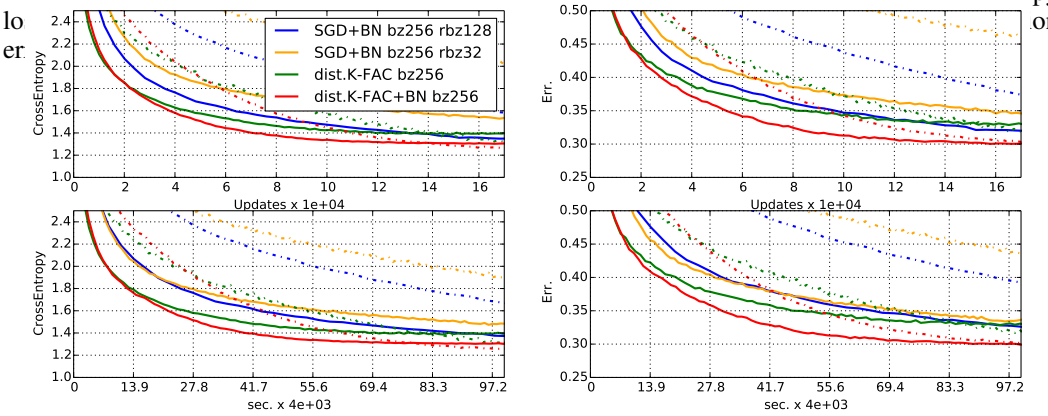

Figure 3: Optimization performance of distributed K-FAC and SGD training GoogLeNet on ImageNet. Dashed lines denote training curves and solid lines denote test curves. *bz* represents the size of mini-batches. *rbz* represents the size of mini-batches used to compute batch normalization mini-batch statistics during training. Top row: cross entropy loss and validation error v.s. the number of updates. Bottom row: cross entropy loss and validation error v.s. wallclock time (in hours). The models are trained using 4 of Nvidia K80 GPUs.

2012), GoogLeNet IncepitonV1 (Szegedy et al., 2014) and 50-layer residual network (He et al., 2015). The images are resized to 256x256 images. The only random cropping is used to generate the final 224x224 input images to the models. We use a less extensive image pre-processing pipeline as the purpose of this paper to not to show the state-of-the-art ImageNet results but rather demonstrating distributed K-FAC as an optimization technique has desirable scaling behavior. In the following experiments, we uses initial KL threshold, $c_0 = 0.01$ and the exponential decay of c is similar to the scheme used in (Szegedy et al., 2014) that decays the threshold by a factor of 0.96 after half of an epoch. Factorized damping of $\epsilon = 0.001$ and the fast Kronecker approximation scheme from Appendix A are used.

Batch Normalization (Ioffe and Szegedy, 2015) uses the distribution of the a hidden unit over a mini-batch to compute the mean and variance that used to normalize the hidden units. It has been shown to help speed-up SGD and other first order optimization techniques for training large ImageNet models. Batch Normalization has the advantage of simplicity and negligible computation overhead. Without any changes to the algorithm, distributed K-FAC can be used to optimize a batch-normalized neural network. The Fisher of the weight matrices in each batch-normalized layers are approximated with K-FAC while separate block diagonal Fishers are used for gain and bias parameters.

In Fig. 3, we compared distributed K-FAC on GoogLeNet with and without batch normalization. We observe that distributed K-FAC makes similar progress per-update on the training cross entropy with and without batch normalization. Moreover, distributed K-FAC is 3.5 times faster than batch normalization (orange line). Batch normalization, however, does help distributed K-FAC generalize better, likely due to its well-known regularizing effect. The batch statistics of a batch normalization

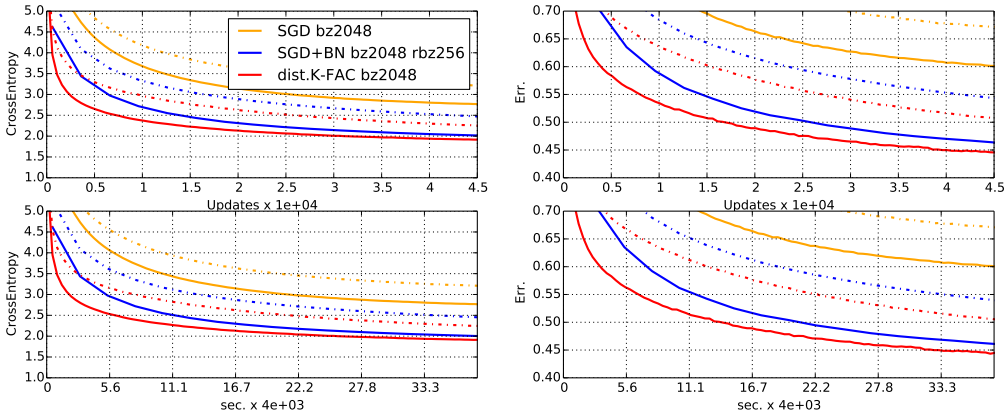

Figure 4: Optimization performance of distributed K-FAC and SGD training AlexNet on ImageNet. Dashed lines denote training curves and solid lines denote test curves. *bz* represents the size of mini-batches. *rbz* represents the size of mini-batches used to compute batch normalization mini-batch statistics during training. Top row: cross entropy loss and validation error v.s. the number of updates. Bottom row: cross entropy loss and validation error v.s. wallclock time(in hours). The models are trained using 8 of Nvidia K80 GPUs.

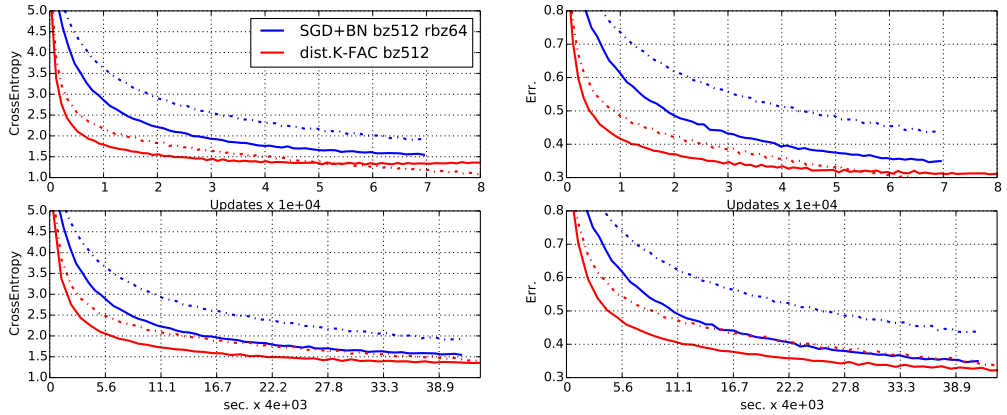

Figure 5: Optimization performance of distributed K-FAC and SGD training ResNet50 on ImageNet. The dashed lines are the training curves and solid lines are the test curves. *bz* represents the size of mini-batches. *rbz* represents the size of mini-batches used to compute batch normalization mini-batch statistics during training. Top row: cross entropy loss and validation error v.s. the number of updates. Bottom row: cross entropy loss and validation error v.s. wallclock time(in hours). The models are trained using 8 of Nvidia K80 GPUs.

layer are typically computed using a mini-batch of 32. The small sample size introduces additional stochasticity that acts as a regularizer but hurt optimization performance. To decouple the effect of regularization and improving optimization, we consider a stronger batch normalization baseline that computes the batch statistics over a mini-batch of 128. We notice a factor of two speed-up for our batch norm baseline that larger mini-batch to compute the statistics. For the simplicity of our discussion, distributed K-FAC is not combined with batch normalization in the the rest of the experiments as we are chiefly interested in evaluating optimization performance, not regularization. Also the rest of the batch normalization results are shown using our stronger baseline that computes batch statistics using large mini-batches and is noted by *rbz* in the figures.

To demonstrate distributed K-FAC can efficiently optimize models with hundreds of millions of parameters, we train AlexNet using distributed K-FAC and compare to SGD with batch normalization. Because the first fully connected layers in AlexNet has hundreds of millions of parameters, the double-factored Kronecker approximation is used. The results of AlexNet are shown on Figure4.

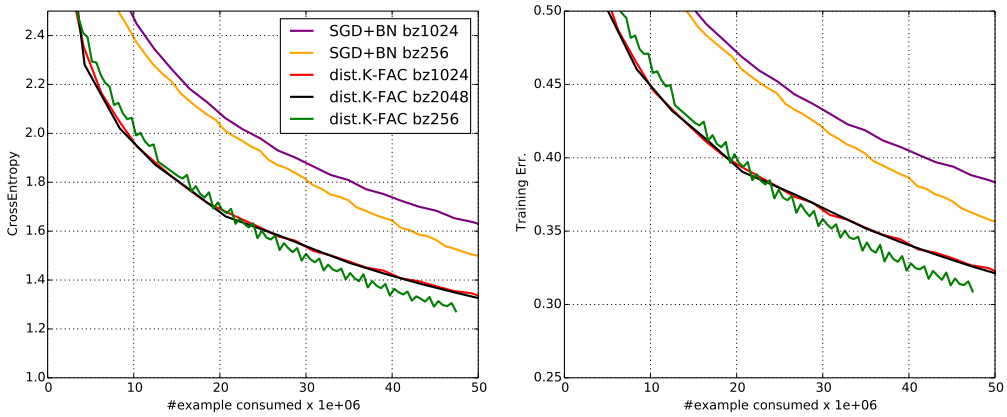

Figure 6: The comparison of distributed K-FAC and SGD on per training case progress on training loss and errors. The experiments were conducted using GoogLeNet with various mini-batch sizes.

Note that computing eigendecomposition for all the Kronecker factors in AlexNet takes around 6 minutes. Distributed K-FAC is still able to make rapid progress using a very stale approximate Fisher.

Natural gradient was previously shown to be effective for very deep auto-encoders (Martens and Grosse, 2015). We show that distributed K-FAC can efficiently scale to very deep convolutional neural networks and speed up its training, We compare distributed K-FAC and SGD on training the 50 layer ResNet architecture (He et al., 2015). The result from Fig. 5 shows that distributed K-FAC provides significant speed-up during the early stage of training comparing to SGD. (ResNet experiments are still running and Fig. 5 will be updated once the experiments are done)

To explore how well distributed K-FAC scales to more parallel computational resources and larger mini-batches, we trained GoogLeNet with mini-batch size of $\{256, 1024, 2048\}$. The results are shown in a per training case progress plot in Fig. 6. It is a rough estimates of how well the algorithm can scale up to more computational resources. Comparing to SGD with batch normalization, distributed K-FAC makes almost equal progress on each training case regardless of the mini-batch sizes. That is distributed K-FAC is able to speed-up the training progress proportional to the parallel computational resources it is given. On the other hand, SGD starts to show diminishing returns on per training case progress with the batch size of 1024.

7 Discussion

We have introduced distributed K-FAC, an asynchronous distributed natural gradient algorithm that augments distributed synchronous SGD with K-FAC for tractable Fisher approximation. We also proposed a crude Kronecker approximation to convolutional layers and a double-factored Kronecker approximation that allows distributed K-FAC to scale up to large models with hundreds of millions of parameters. In our simulation, computing distributed K-FAC updates has little overhead and has shown comparable runtime as SGD on the same mini-batch sizes and it usually speeds up SGD training by a factor of two. Moreover, distributed K-FAC scales well over the distributed computation resource and, unlike SGD, has little diminish of returns from using larger mini-batches. As more parallel computation resources become available in deep learning, we believe the benefit of using asynchronous distributed second-order method such as distributed K-FAC will be more pronounced.

References

James Martens and Roger Grosse. Optimizing neural networks with kronecker-factored approximate curvature. In *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, pages 2408–2417, 2015.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of The 32nd International Conference on Machine Learning*, pages

448–456, 2015.

Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *arXiv preprint arXiv:1409.4842*, 2014.

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.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*, 2015.

Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231, 2012.

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Nicolas Le Roux, Pierre-Antoine Manzagol, and Yoshua Bengio. Topmoumoute online natural gradient algorithm. In *Advances in neural information processing systems*, pages 849–856, 2008.

Philipp Moritz, Robert Nishihara, and Michael Jordan. A linearly-convergent stochastic l-bfgs algorithm. In *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics*, pages 249–258, 2016.

Richard H Byrd, SL Hansen, Jorge Nocedal, and Yoram Singer. A stochastic quasi-newton method for large-scale optimization. *SIAM Journal on Optimization*, 26(2):1008–1031, 2016.

James Martens. Deep learning via hessian-free optimization. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 735–742, 2010.

Oriol Vinyals and Daniel Povey. Krylov subspace descent for deep learning. In *AISTATS*, pages 1261–1268, 2012.

Tom Heskes. On "natural" learning and pruning in multilayered perceptrons. *Neural Computation*, 12(4): 881–901, 2000.

Roger Grosse and Ruslan Salakhutdinov. Scaling up natural gradient by factorizing fisher information. In *Proceedings of the 32nd International Conference on Machine Learning (ICML)*, 2015.

Daniel Povey, Xiaohui Zhang, and Sanjeev Khudanpur. Parallel training of DNNs with natural gradient and parameter averaging. In *International Conference on Learning Representations: Workshop track*, 2015.

Guillaume Desjardins, Karen Simonyan, Razvan Pascanu, and Koray Kavukcuoglu. Natural neural networks. In *Advances in Neural Information Processing Systems*, pages 2071–2079, 2015.

Roger Grosse and James Martens. A kronecker-factored approximate fisher matrix for convolution layers. In *Proceedings of the 33rd International Conference on Machine Learning (ICML-16)*, 2016.

J. Martens. New insights and perspectives on the natural gradient method. 2014.

Shun-Ichi Amari. Natural gradient works efficiently in learning. *Neural computation*, 10(2):251–276, 1998.

J. Martens and I. Sutskever. Training deep and recurrent networks with Hessian-free optimization. In *Neural Networks: Tricks of the Trade*, pages 479–535. Springer, 2012.

Boris T Polyak and Anatoli B Juditsky. Acceleration of stochastic approximation by averaging. *SIAM Journal on Control and Optimization*, 30(4):838–855, 1992.

Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. *arXiv preprint arXiv:1512.00567*, 2015.

Nicol N. Schraudolph. Fast curvature matrix-vector products for second-order gradient descent. *Neural Computation*, 14(7), 2002.

Martın Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015.

James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: A cpu and gpu math compiler in python. In *Proc. 9th Python in Science Conf*, pages 1–7, 2010.

A A CHEAPER KRONECKER FACTOR APPROXIMATION FOR CONVOLUTION LAYERS

In a convolution layer, the gradient is the sum of the outer product between the receptive field input activation $\mathcal{A}_t$ and the back-propagated derivatives $\mathcal{D}s_t$ at each spatial location $t \in \mathcal{T}$. One cannot simply apply the standard Kronecker factored approximation from Martens and Grosse (2015) to each location, sum the results, and then take the inverse, as there is no known efficient algorithm for computing the inverse of such a sum.

In Grosse and Martens (2016), a Kronecker-factored approximation for convolutional layers called Kronecker Factors for Convolution (KFC) was developed. It works by introducing additional statistical assumptions about how the weight gradients are related across locations. In particular, KFC assumes spatial homogeneity, i.e. that all locations have the same statistics, and spatially uncorrelated derivatives, which (essentially) means that gradients from any two different locations are statistically independent.

In this section we introduce an arguably simpler Kronecker factored approximation for convnets that is cheaper to compute. In practice, it appears to be competitive with the original KFC approximation in terms of per-iteration progress on the objective, working worse in some experiments and better in others, while (often) improving wall-clock time due to its cheaper cost.

It works by approximating the sum of the gradients over spatial locations as the outer product of the averaged receptive field activations over locations $\mathbb{E}_t[\mathcal{A}_t]$, and the averaged back-propagated derivatives $\mathbb{E}_t[\mathcal{D}s_t]$, multipled by the number of spatial locations $|\mathcal{T}|$. In other words:

$$\mathbb{E}[\mathrm{vec}\{\mathcal{D}W\}\,\mathrm{vec}\{\mathcal{D}W\}^\top] = \mathbb{E}\left[\mathrm{vec}\{\sum_{t\in\mathcal{T}} \mathcal{A}_t \mathcal{D}s_t^\top\}\,\mathrm{vec}\{\sum_{t\in\mathcal{T}} \mathcal{A}_t \mathcal{D}s_t^\top\}^\top\right] \tag{15}$$

$$= \mathbb{E}\left[\left(\sum_{t\in\mathcal{T}} \mathcal{A}_t \otimes \mathcal{D}s_t\right)\left(\sum_{t\in\mathcal{T}} \mathcal{A}_t \otimes \mathcal{D}s_t\right)^\top\right] \tag{16}$$

$$\approx \mathbb{E}\left[\left(|\mathcal{T}|\,\mathop{\mathbb{E}}_t[\mathcal{A}_t] \otimes \mathop{\mathbb{E}}_t[\mathcal{D}s_t]\right)\left(|\mathcal{T}|\,\mathop{\mathbb{E}}_t[\mathcal{A}_t] \otimes \mathop{\mathbb{E}}_t[\mathcal{D}s_t]\right)^\top\right] \tag{17}$$

Under the approximation assumption that the second-order statistics of the average activations, $\mathbb{E}_t[\mathcal{A}_t]$, and the second-order statistics of the average derivatives, $\mathbb{E}_t[\mathcal{D}s_t]$, are uncorrelated, this becomes:

$$|\mathcal{T}|^2\,\mathbb{E}\left[\mathop{\mathbb{E}}_t[\mathcal{A}_t]\mathop{\mathbb{E}}_t[\mathcal{A}_t]^\top\right] \otimes \mathbb{E}\left[\mathop{\mathbb{E}}_t[\mathcal{D}s_t]\mathop{\mathbb{E}}_t[\mathcal{D}s_t]^\top\right] \tag{18}$$

This approximation is cheaper than the original KFC approximation because it is easier to compute a single outer product (after averaging over locations) than it is to compute an outer product at each location and then average. In the synchronous setting, for the large convolutional networks we experimented with, this trick resulted in a 20-30% decrease in overall wall clock time per iteration, with little effect on per-iteration progress.

B AUTOMATIC CONSTRUCTION OF THE K-FAC COMPUTATION GRAPHS

In recent years, deep learning libraries have moved towards the computational graph abstraction (Abadi et al., 2016; Bergstra et al., 2010) to represent neural network computations. In this section we describe how to design an algorithm that automatically extracts the information required for the Kronecker-factored approximations from such graphs.

For the convenience of our discussion, let the computation graph be a directed bipartite graph that has a set of operator nodes doing some computation and some variable nodes that holds intermediate computational results. The trainable parameters are stored in the memory that is loaded or mutated through read/write operator nodes. The gradient computation graph for the trainable parameters is usually generated via automatic differentiation. Assume the trainable parameters are grouped layer-wise as a set of weights and biases. To compute the Fisher block for that layer, we want to find all the nodes holding the gradients of the trainable parameters in a computation graph.

In a computation graph, a set of parameters has an efficient Kronecker approximation to its Fisher block if its corresponding gradient node has a matrix product or convolution operator node as its immediate parent node. For these parameters, the small Kronecker factors are the second-order statistics matrices of the inputs to the parent operator node of their gradients (typically the activities $\mathcal{A}$ and back-propagated derivatives $\mathcal{D}s$).

In analogues to generating the gradient computation graph through automatic differentiation, given an arbitrary computation graph with a set of the trainable parameters, we would like to use the existing nodes in the given graph to automatically generate K-FAC computation graph to compute approximate Kronecker factored Fisher of the parameters. One simple search strategy is to traverse the computation graph from the gradient nodes to their immediate parent nodes. Kronecker factored Fisher is applicable if the immediate parent node is either a matrix product or convolution operator. The exact block-diagonal Fisher is otherwise computed for the rest of the parameters. In a typical neural network, large weights, that are typically followed by a matrix product or convolution, can be handled by K-FAC. The rest of the parameters are often gain and bias vectors for each hidden units that is tractable to compute their block-diagonal Fisher. Note that homogeneous coordinates can be used for a larger block Fisher if the weights and biases of the same layer are annotated in the computation graph.

The second-order statistics, i.e. the auto-correlations, in a K-FAC computation graph is often shared among some parameters. That is the case when the gradients of some parameters are computed from the same activations or back-propagated derivatives. The computation can be reused for those parameters. For example, a neural network that merges the multiple modality or input streams can have shared Kronecker factors at the "merge" layer.

A neural network can be also instantiated multiple times in a computational graph with shared parameters to process different inputs. The gradient of the parameters shared across the instantiations are the sum of the individual gradients from each instantiation. Given such computation graph, the immediate parent operator node from the gradient is the summation whose inputs are computed by the same type of operators. Without additional knowledge about the computation graph, one can assume the individual gradients in the summation are independent of each other and thus K-FAC or block-diagonal Fisher is applied to each of the individual gradient nodes. The second-order statistics of the parameter is the average of the individual statistics of the summation terms.

Distributed K-FAC applies the above the strategy to automatically construct computation graphs for computing the second-order statistics of the gradients of the parameters and applying Kronecker factorization whenever possible. The resulting second-statistics are either Kronecker factors of the approximate Fisher or block-diagonal Fisher.