

# 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 algorithms to avoid the inversion problem entirely (Martens, 2010a; 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<sup>1</sup>. 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 hundred 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-iteration 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  $\mathcal{DW}$  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:

$$F = \mathbb{E}_{\mathbf{x}, y \sim P} [\text{vec}\{\mathcal{DW}\} \text{vec}\{\mathcal{DW}\}^\top], \quad (1)$$

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 derivatives as  $\mathcal{D}s = \frac{d\mathcal{L}}{ds} \in \mathbb{R}^{C_{out}}$ . Note that the gradients of the weights is the outer product of the input activation and back-propagated derivatives  $\mathcal{DW} = \mathcal{A}\mathcal{D}s^\top$ . K-FAC approximates the Fisher block as a

<sup>1</sup>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:

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

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 interpreted 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  $\lambda I$  to  $\hat{F}$  is approximated by adding  $\pi_{\mathcal{A}}\lambda^{\frac{1}{2}}I$  to  $\mathbb{E} [\mathcal{A}\mathcal{A}^\top]$  and  $\pi_{\mathcal{D}s}\lambda^{\frac{1}{2}}I$  to  $\mathbb{E} [\mathcal{D}s\mathcal{D}s^\top]$ , 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  $\lambda$ .)

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:

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

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-iteration progress in practice. All in all, the per-iteration 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 FISHER BLOCK INVERSION

Computing the parameter updates as per Eq.3 requires the estimated gradients to be multiplied by the inverse of the smaller Kronecker factors. This requires periodically computing (typically) either inverses or eigendecompositions of each of these factors. While these factors typically have sizes only in the hundreds or low thousands, very deep networks may have hundreds of such matrices

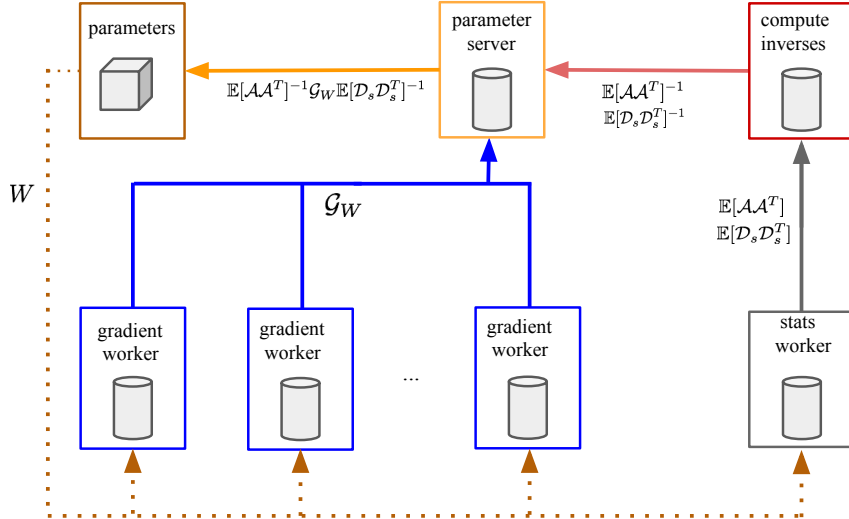


Figure 1: The diagram illustrate the distributed computation of K-FAC. Gradient workers (blue) computes the gradient w.r.t. the loss function. Stats workers (grey) computes the sampled second-order statistics. Additional workers (red) compute inverse Fisher blocks. The parameter server (orange) uses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

(2 or more for each layer). 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, inverting the approximate Fisher blocks represents a significant computational cost.

It has been observed that refreshing the inverse of the Fisher blocks only occasionally and uses stale values otherwise has only a small detrimental effect on average per-iteration progress, perhaps because the curvature changes relatively slowly (Martens and Grosse, 2015). We push this a step further by computing the inverses *asynchronously* while the network is still training. Because the required linear algebra operations are CPU-bound while the rest of our computations are GPU-bound, we perform them 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-iteration optimization performance. In our experiments, we found that computing the inverses asynchronously usually offered a 40-50% speed-up to the overall wall-clock time of the K-FAC algorithm.

### 3.2 ASYNCHRONOUS STATISTICS COMPUTATION

The other major source of computational overhead in K-FAC 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 forward pass computations to be shared between the gradient and statistics computations. By computing the gradients and statistics on separate mini-batches, we can enable a higher degree of parallelism, at the expense of slightly more total computational operations. Under this scheme, the statistics estimation is independent of the gradient computation, so it can be done on one or more separate worker nodes with their own independent data shards. These worker nodes receive parameters from the parameter server (just as in synchronous SGD) and communicate statistics back to the parameter server. In our experiments, we assigned at most one worker to computing statistics.

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  $9216 \times 9216$ , which is too expensive to be explicitly 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  $9216 \times 4096$  weight matrix in the first fully connected layer is equivalent to a filterbank of 4096 filters with kernel size  $6 \times 6$  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:

$$\mathbb{E}[\text{vec}\{DW\} \text{vec}\{DW\}^\top] = \mathbb{E}[\text{vec}\{A\} \text{vec}\{A\}^\top \otimes DsDs^\top], \quad A \in \mathbb{R}^{\mathcal{T} \times C_{in}}. \quad (4)$$

We begin by making the following rank-1 approximation:

$$A \approx \mathcal{K}\Psi^\top, \quad (5)$$

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  $\sigma_1, u_1, v_1$  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  $\Psi$  captures the pattern across the filter responses. Under the rank-1 approximation of  $A$  we have:

$$\mathbb{E}[\text{vec}\{A\} \text{vec}\{A\}^\top \otimes DsDs^\top] \approx \mathbb{E}[\text{vec}\{\mathcal{K}\Psi^\top\} \text{vec}\{\mathcal{K}\Psi^\top\}^\top \otimes DsDs^\top] \quad (6)$$

$$= \mathbb{E}[\mathcal{K}\mathcal{K}^\top \otimes \Psi\Psi^\top \otimes DsDs^\top]. \quad (7)$$

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

$$\mathbb{E}[\text{vec}\{DW\} \text{vec}\{DW\}^\top] \approx \mathbb{E}[\mathcal{K}\mathcal{K}^\top] \otimes \mathbb{E}[\Psi\Psi^\top] \otimes \mathbb{E}[DsDs^\top]. \quad (8)$$

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  $\mathcal{R}_i : \mathbb{R}^{d_1 \times d_2 \times d_3} \rightarrow \mathbb{R}^{d_j d_k \times d_i}$  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 inverse double-factored Kronecker approximated Fisher block as:

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

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  $\lambda I$  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  $\lambda I$  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  $\pi_a$ ,  $\pi_b$  and  $\pi_c$  satisfying  $\pi_a \pi_b \pi_c = 1$ . The error associated with this approximation is:

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

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

Following Martens and Grosse (2015), we choose  $\pi_a$ ,  $\pi_b$  and  $\pi_c$  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  $\pi_a$

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

where the  $d$ ’s are the number of rows (equiv. columns) of the corresponding Kronecker factor matrices. The corresponding formulae for  $\pi_b$  and  $\pi_c$  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:

$$D_{\text{KL}}[q||p] \approx \frac{1}{2} \mathbf{v}^\top F \mathbf{v} \quad (13)$$

This quantity can be computed with a curvature-vector product (Schraudolph, 2002). Observe that choosing a step size of  $\eta$  will produce an update with squared Fisher norm  $\eta^2 \mathbf{v}^\top F \mathbf{v}$ . Instead of using a learning rate schedule, we choose  $\eta_k$  in each iteration such that the squared Fisher norm is at most some value  $c_k$ :

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

Here  $k$  controls the decay schedule of the squared Fisher norm threshold.  $k$  is either incremented in each iteration or periodically every half an epoch as in our ImageNet experiments. 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  $c_k = c_0 \zeta^k$ , where  $c_0$  and  $\zeta$  are tunable parameters. Shrinking the maximum changes in the model prediction after each update is analogous to shrink the trust region of the second-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  $\mathbf{v}^\top \hat{F} \mathbf{v} = \mathbf{v}^\top \hat{F} (\hat{F}^{-1} \mathcal{G}_W) = \mathbf{v}^\top \mathcal{G}_W$ . 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. We found this outperform simple exponential learning rate decay on ImageNet experiments (see Appendix B).

## 6 EXPERIMENTS

We explored the properties and evaluated the performance of distributed K-FAC as applied to training convolutional neural networks on image classification tasks using the CIFAR-10 and ImageNet datasets.

Due to computational resource constraints, we used a single GPU server with 8 Nvidia K20 GPUs to simulate a large distributed system. The GPUs were used as gradient workers that computed the gradient over a large mini-batch, with the CPUs acting as a parameter server. The Fisher block inversions were performed on the CPUs in parallel using as many threads as possible. The second-order statistics required for the various Fisher block approximations were either computed by the gradient workers (CIFAR-10 experiments), or asynchronously using a separate dedicated “stats worker” (ImageNet experiments).

Meta-parameters such as learning rates, damping parameters, and the decay-rate for the second-order statistics, were optimized carefully by hand for each method. The momentum was fixed to 0.9.

Similarly to Martens and Grosse (2015), we applied an exponentially decayed Polyak averaging scheme to the sequence of output iterates produced by each method. We found this improved their convergence rate in the later stages of optimization, and reduced or eliminated the need to decay the learning rates.

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 Fisher block 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 C for details. Because TensorFlow’s parallel primitives were designed with scalability in mind, it should be possible to scale our implementation to a larger distributed system of hundreds of workers.

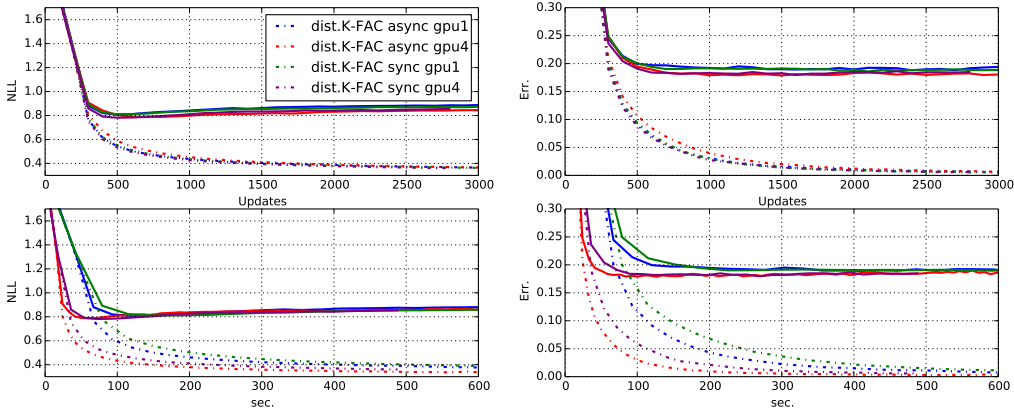


Figure 2: The results from our CIFAR-10 experiment looking at the effectiveness of asynchronously computing the approximate Fisher inverses. *gpu* indicates the number of gradient workers. Dashed lines denote training curves and solid lines denote test curves. Top row: cross entropy loss and classification error vs the number of updates. Bottom row: cross entropy loss and classification error vs wallclock time.

### 6.1 CIFAR-10 CLASSIFICATION AND ASYNCHRONOUS FISHER BLOCK INVERSION

In our first experiment we evaluated the effectiveness of asynchronously computing the approximate Fisher inverses (as described in Section 3.1) on a simple convnet training task. We considered the effect that this has on both the quality of the updates, as measured by per-iteration progress on the objective, and on the average per-iteration wall-clock time.

The task is to train a basic convolutional network model on the CIFAR-10 image classification dataset (Krizhevsky and Hinton, 2009). The model has 3 convolutional layers of 32-32-64 filters, each with a receptive field size of 5x5, followed by a softmax layer that predicts 10 classes. This is a similar but not identical CIFAR-10 model that was used by Grosse and Martens (2016). All the CIFAR-10 experiments use a mini-batch size of 512.

The baseline method is a simple synchronous version of distributed K-FAC with a fixed learning rate, and up to 4 GPU workers computing gradients and second-order statistics, which recomputes the inverses of the approximate Fisher blocks once every 20 iterations. This baseline method behaves similarly to the implementation of K-FAC in Grosse and Martens (2016), while being potentially faster due to its greater use of parallelism. We compare this baseline to a version of distributed K-FAC where the approximate Fisher blocks are inverted asynchronously while gradient workers keep making progress using staled approximate Fisher(), which is otherwise configured identically). Note that under this scheme, the inverses of the approximate Fisher blocks are updated all at once whenever all the approximate Fisher blocks are inverted, which could be more often than every 20 iterations.

The results of this first experiment are plotted in Fig. 2. We found that the asynchronous version iterated about 1.5 times faster than the synchronous version, while its per-iteration progress remained comparable. The plots show that the asynchronous version is better at taking advantage of parallel computation and displayed an almost linear speed-up as the number of gradient workers increases to 4. In terms of the wall-clock performance, the asynchronous version of distributed K-FAC is able to make 700 gradient iterations in just 1 minute and the test errors of the simple CIFAR-10 model plateau at 19%.

### 6.2 IMAGENET CLASSIFICATION

In our second set of experiments we benchmarked distributed K-FAC against several other popular approaches, and considered the effect of mini-batch size on per-iteration progress. To do this we trained various off-the-shelf convnet architectures for ImageNet classification (Russakovsky et al., 2015): AlexNet (Krizhevsky et al., 2012), GoogLeNet InceptionV1 (Szegedy et al., 2014) and the 50-layer Residual network (He et al., 2015).

We used a less extensive dataset augmentation/pre-processing pipeline than is typically used for ImageNet, as the purpose of this paper is not to achieve state-of-the-art ImageNet results, but rather



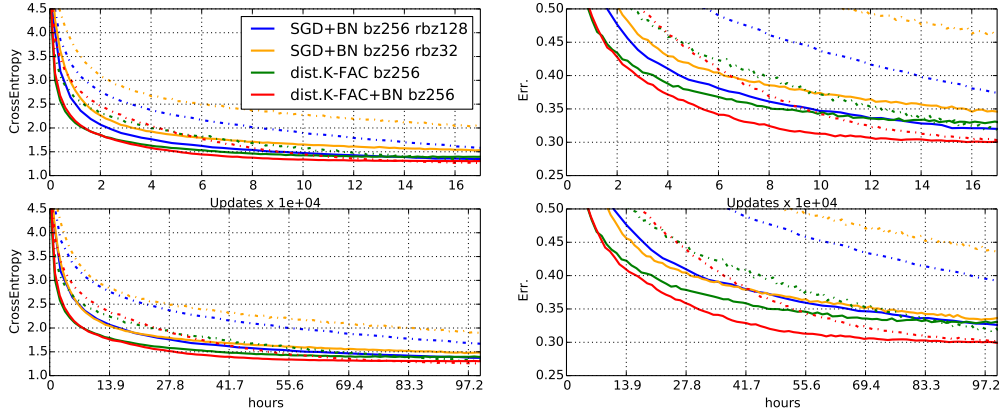


Figure 3: Optimization performance of distributed K-FAC and SGD training GoogLeNet on ImageNet. Dashed lines denote training curves and solid lines denote validation curves.  $bz$  indicates the size of mini-batches.  $rbz$  indicates the size of chunks used to assemble the BN updates. Top row: cross entropy loss and classification error v.s. the number of updates. Bottom row: cross entropy loss and classification error vs wallclock time (in hours). All methods used 4 GPUs, with distributed K-FAC using the 4-th GPU as a dedicated asynchronous stats worker.

to evaluate the optimization performance of distributed K-FAC. In particular, the dataset consists of  $224 \times 224$  images and during training the original images are first resized to  $256 \times 256$  and then randomly cropped back down to  $224 \times 224$  before being fed to the network.

For all of our ImageNet experiments we used the cheaper Kronecker factorization from Appendix A and KL-based step sized selection method described in Section 5 with parameters  $c_0 = 0.01$  and  $\zeta = 0.96$ . The SGD baselines use exponential decay schedule with a decay rate of 0.96. Instead of decaying the learning rate and the KL-based step size in each iteration, we apply decay every half an epoch in our K-FAC and batch normalized SGD experiments. This scheme corresponds to increment  $k$  every half an epoch in Section 5. It is consistent with the experiment setups from (Ioffe and Szegedy, 2015) The learning rate of the plain SGD optimizer decays every two epochs.

### 6.2.1 GOOGLELENET AND BATCH NORMALIZATION

Batch Normalization (Ioffe and Szegedy, 2015) is a reparameterization of neural networks that can make them easier to train with first-order methods, and has been successfully applied to large ImageNet models. It can be thought of as a modification of the units of a neural network so that each one centers and normalizes its own raw input (using the statistics of the current mini-batch), and then applies a separate shift and scaling operation to the result via its own local “bias” and “gain” parameters. It is closely related to centering techniques (Schraudolph, 1998), with the main difference being that it makes the centering and normalization operations part of the model instead of the optimization algorithm (and thus “backprops” through them when computing the gradient).

Without any changes to the algorithm, distributed K-FAC can be used to train neural networks that have Batch Normalization (BN) layers. The weight-matrix gradient for such layers has the same structure as it does for standard layers, and so Fisher blocks can be approximated using the same set of techniques. The per-unit gain and bias parameters cause a minor complication, but because they are relatively few in number one can compute an exact Fisher block for each of them (whose size will be the same as the associated factor in the Kronecker product approximation for that layer).

Computing updates for BN networks over large mini-batches is usually done by splitting the mini-batch into chunks of size 32, computing the gradients separately for these chunks (using only the data in the chunk to compute the mean and variance statistics), and then summing them together. Using small sample sets to compute the statistics like this introduces additional stochasticity into the BN update that acts as a regularizer, but can also hurt optimization performance. To help decouple the effect of regularization and optimization, we also compared to a BN baseline that uses larger chunks. We found can give a factor of 2 speed-up in optimization performance versus the standard BN baseline. In our figures  $rbz$  will indicate the chunk size, which defaults 32 if left unspecified.

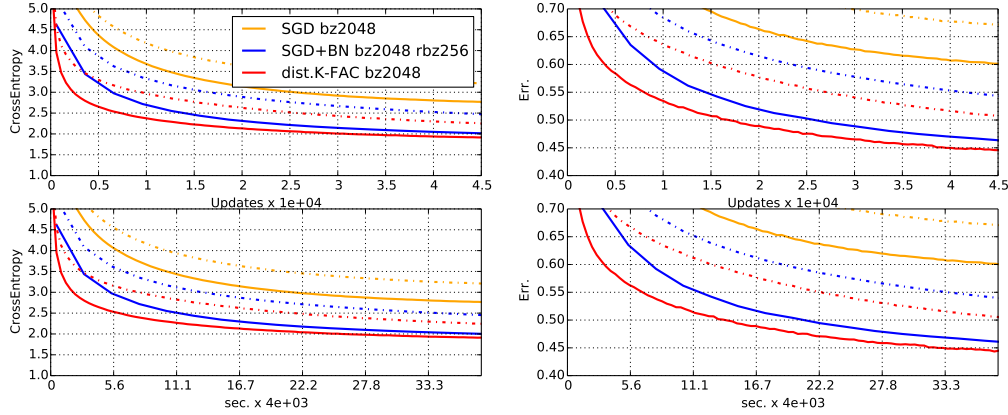


Figure 4: Optimization performance of distributed K-FAC and SGD training AlexNet on ImageNet. Dashed lines denote training curves and solid lines denote validation curves. *bz* indicates the size of the mini-batches. *rbz* indicates the size of chunks used to assemble the BN updates. Top row: cross entropy loss and validation error vs the number of updates. Bottom row: cross entropy loss and validation error vs wallclock time (in hours). All methods used 8 GPUs, with distributed K-FAC using the 8-th GPU as a dedicated asynchronous stats worker.

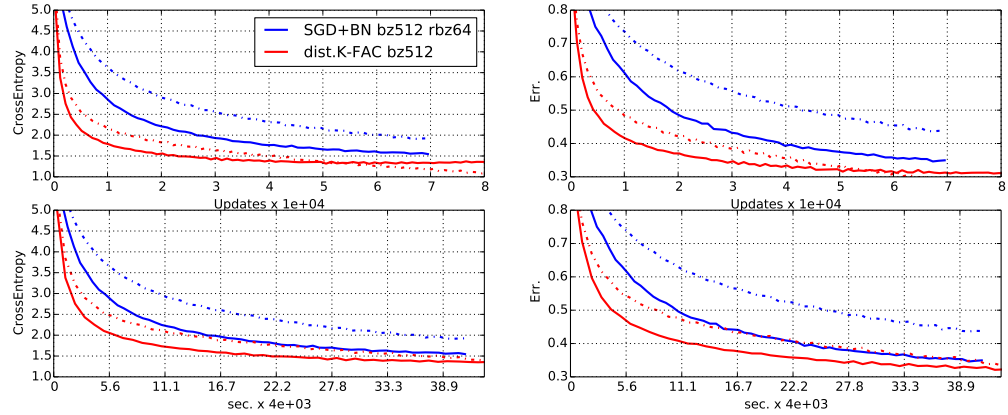


Figure 5: Optimization performance of distributed K-FAC and SGD training ResNet50 on ImageNet. The dashed lines are the training curves and solid lines are the validation curves. *bz* indicates the size of mini-batches. *rbz* indicates the size of chunks used to assemble the BN updates. Top row: cross entropy loss and classification error v.s. the number of updates. Bottom row: cross entropy loss and classification error v.s. wallclock time (in hours). All methods used 8 GPUs, with distributed K-FAC using the 8-th GPU as a dedicated asynchronous stats worker.

In Fig. 3, we compare distributed K-FAC to SGD on GoogLeNet with and without BN. All methods used 4 GPUs, with distributed K-FAC using the 4-th GPU as a dedicated asynchronous stats worker.

We observe that the per-iteration progress made by distributed K-FAC on the training objective is not significantly affected by the use of BN. Moreover, distributed K-FAC is 3.5 times faster than SGD with standard BN baseline (orange line) and 1.5-2 times faster than the enhanced BN baseline (blue line). BN, however, does help distributed K-FAC generalize better, likely due to its aforementioned regularizing effect.

For the simplicity of our discussion, distributed K-FAC is not combined with BN in the the rest of the experiments, as we are chiefly interested in evaluating optimization performance, not regularization, and BN doesn't seem to provide any *additional* benefit to distributed K-FAC in regards to the former.

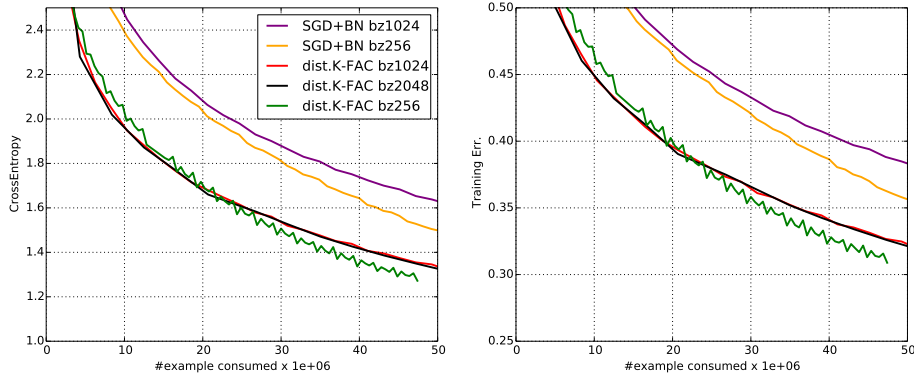


Figure 6: The comparison of distributed K-FAC and SGD on per training case progress on training loss and errors. The experiments were conducted using GoogLeNet with various mini-batch sizes.

### 6.2.2 ALEXNET AND THE DOUBLY-FACTORED KRONECKER APPROXIMATION

To demonstrate that distributed K-FAC can efficiently optimize models with very wide layers we train AlexNet using distributed K-FAC and compare to SGD+BN. The doubly-factored Kronecker approximation proposed in Section 4 is applied to first fully-connected layer of AlexNet, which has 9216 input units and is thus too wide for the standard Kronecker approximation to be feasible. Even using doubly-factored Kronecker approximation, to compute all the inverses of the Fisher blocks can still be time consuming. On an eight K-80 GPU server with 16 core Xeon 2.2Ghz CPUs, all the inverses are updated the same time and are typically refreshed every hundreds of gradient iterations by the asynchronous inverse worker running on the local CPU.

In the experiments shown in Fig. 4, the inverses of the asynchronous Fisher blocks are “lagged” behind the gradient workers for about two hundred parameter updates. The plot shows Distributed K-FAC is still able to make rapid progress using the stale Fisher inverses speeding-up AlexNet training by 1.5 times comparing to our improved batch normalization baseline.

### 6.2.3 VERY DEEP ARCHITECTURES (RESNETS)

In recent years very deep convolutional architectures have been successfully applied to ImageNet classification. These networks are particularly challenging to train because the usual difficulties associated with deep learning are especially severe. Fortunately second-order optimization is perhaps ideally suited to addressing these difficulties in a robust and principled way (Martens, 2010b).

To investigate whether distributed K-FAC can scale to such architectures and provide useful acceleration, we compared it to SGD+BN using the 50 layer ResNet architecture (He et al., 2015). The results from this experiment are plotted in Fig. 5. They show that distributed K-FAC provides significant speed-up during the early stages of training compared to SGD+BN. [Note to early readers: ResNet experiments are still running and Fig. 5 will be updated once they are finished.]

### 6.2.4 MINI-BATCH SCALING PROPERTIES

In our final experiment we explored how well distributed K-FAC scales as additional parallel computing resources become available. To do this we trained GoogLeNet with varying mini-batch sizes of  $\{256, 1024, 2048\}$ , and measured per-training-case progress. Ideally, if extra gradient data is being used efficiently, one should expect the per-training-case progress to remain relatively constant with respect to mini-batch size. The results from this experiment are plotted in Fig. 6, and show that distributed K-FAC exhibits something close to this ideal behavior, while SGD+BN rapidly loses data efficiency when moving beyond a mini-batch size of 256. These results suggest that distributed K-FAC, more so than the SGD+BN baseline, is capable of speeding up training in proportion to the amount of parallel computational resources available.

## 7 DISCUSSION

We have introduced distributed K-FAC, an asynchronous distributed second-order optimization algorithm which computes Kronecker-factored Fisher approximations stochastic gradients over larger mini-batches asynchronously and in parallel.

Our experiments show that the extra overhead introduced by distributed K-FAC is mostly mitigated by the use of parallel asynchronous computation, resulting in updates that can be computed in a similar amount of time to those of distributed SGD, while making much more progress on the objective function per iteration. We showed that in practice this can lead to speedups of roughly 4x compared to standard SGD + Batch Normalization (BN), and 2x compared to SGD + an improved version of BN on large-scale convolutional network training tasks.

We also proposed a doubly-factored Kronecker approximation that allows distributed K-FAC to scale up to large models with hundreds of millions of parameters, and demonstrated the effectiveness of this approach in experiments.

Finally, we showed that distributed K-FAC enjoys a favorable scaling property with mini-batch size that is seemingly not shared by SGD+BN. In particular, we showed that the amount of per-iteration progress tends to be proportional to the mini-batch size up to a much larger threshold than for SGD+BN. This implies that it will yield even further reductions in total wall-clock training time when implemented in larger distributed system than the one we used in our experiments.

## 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, 2010a.

- 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.
- James Martens. New insights and perspectives on the natural gradient method. *arXiv preprint arXiv:1412.1193*, 2014.
- Shun-Ichi Amari. Natural gradient works efficiently in learning. *Neural computation*, 10(2):251–276, 1998.
- James Martens and Ilya 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.
- Martin 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.
- Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. , University of Toronto, 2009.
- 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.
- Nicol N. Schraudolph. Centering neural network gradient factors. In Genevieve B. Orr and Klaus-Robert Müller, editors, *Neural Networks: Tricks of the Trade*, volume 1524 of *Lecture Notes in Computer Science*, pages 207–226. Springer Verlag, Berlin, 1998.
- James Martens. Deep learning via Hessian-free optimization. In *Proceedings of the 27th International Conference on Machine Learning (ICML)*, 2010b.
- 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.

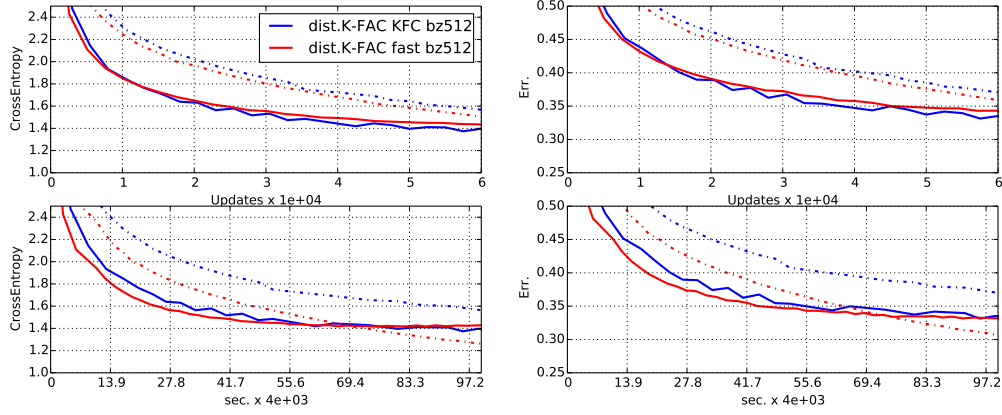


Figure 7: Empirical evaluation of the proposed cheaper Kronecker approximation on GoogLeNet. *bz* indicates the size of the mini-batches. Dashed lines denote training curves and solid lines denote validation curves. Top row: cross entropy loss and classification error vs the number of updates. Bottom row: cross entropy loss and classification error vs wallclock time.

## 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  $\mathbb{E}_t[\mathcal{D}s_t]$ , multiplied by the number of spatial locations  $|\mathcal{T}|$ . In other words:

$$\mathbb{E}[\text{vec}\{DW\} \text{vec}\{DW\}^\top] = \mathbb{E}\left[\text{vec}\left\{\sum_{t \in \mathcal{T}} \mathcal{A}_t \mathcal{D}s_t^\top\right\} \text{vec}\left\{\sum_{t \in \mathcal{T}} \mathcal{A}_t \mathcal{D}s_t^\top\right\}^\top\right] \quad (15)$$

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

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

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:

$$|\mathcal{T}|^2 \mathbb{E}\left[\mathbb{E}_t[\mathcal{A}_t] \mathbb{E}_t[\mathcal{A}_t]^\top\right] \otimes \mathbb{E}\left[\mathbb{E}_t[\mathcal{D}s_t] \mathbb{E}_t[\mathcal{D}s_t]^\top\right] \quad (18)$$

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

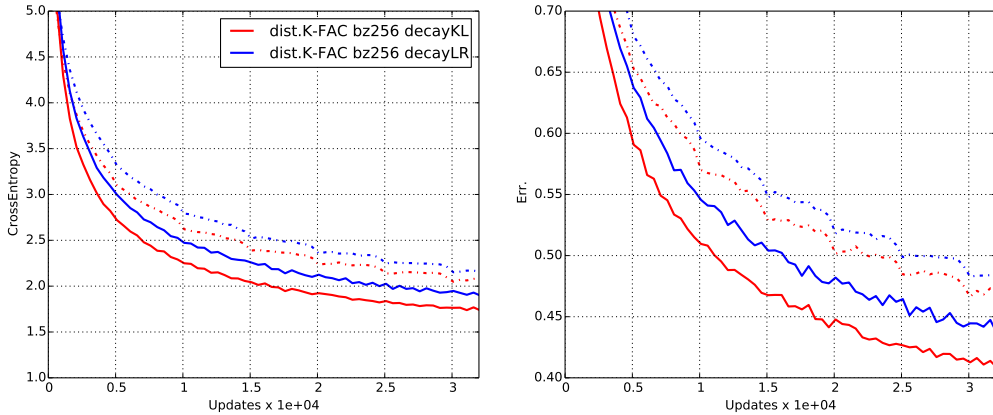


Figure 8: Results from the experiment described in Appendix B. *decayKL* indicates the proposed step-size selection method and *decayLR* indicates standard exponential learning rate decay.

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 EXPERIMENTAL EVALUATION OF THE STEP-SIZE SELECTION METHOD OF SECTION 5

To compare our proposed step size selection from Sec. 5 with the commonly-used exponential learning rate decay, we performed a simple experiment training GoogLeNet. Both the learning rate and threshold  $c$  on the square Fisher norm, is decayed by a factor of 0.96 after every 3200 iterations. The results of this experiment are plotted in Fig. 8, and indicate that our method outperforms the standard baseline.

## C AUTOMATIC CONSTRUCTION OF THE K-FAC COMPUTATION GRAPH

In recent years, deep learning libraries have moved towards the computational graph abstraction (Bergstra et al., 2010; Abadi et al., 2016) to represent neural network computations. In this section we give a high level description of an algorithm that scans a computational graph for parameters for which one of the various Kronecker-factored approximations can be applied, locates nodes containing the required information to compute the second-order statistics required by the approximations, and then constructs a new graph that computes the approximations and uses them to update the parameters.

For the sake of discussion, we will assume the computation graph is 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. We also assume that the trainable parameters are grouped layer-wise as a set of weights and biases. Finally, we assume the gradient computation for the trainable parameters is performed by a computation graph (which is usually is generated via automatic differentiation).

In analogy 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 a new computation graph, a “K-FAC computation graph”, that computes the Kronecker-factored approximate Fisher blocks associated with each group of parameters (typically layers in a neural net), and then uses them to update the parameters.

To compute the Fisher block for a given layer, we want to find all the nodes holding the gradients of the trainable parameters in a computation graph. One simple strategy is to traverse the computation graph from the gradient nodes to their immediate parent nodes.

A set of parameters has a Kronecker-factored 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 Kronecker factor matrices are the second-order statistics of the inputs to the parent operator node of their gradient nodes (typically the activities  $\mathcal{A}$  and back-propagated derivatives  $\mathcal{D}_s$ ). For other sets of parameters an exact Fisher block can be computed instead (assuming they have low enough dimension).

In a typical neural network, most of the parameters are concentrated in weight matrices, that are used for matrix product or convolution operations, for which one of the existing Kronecker-factored approximations applies. Homogeneous coordinates can be used if the weights and biases of the same layer are annotated in the computation graph. The rest of the parameters are often gain and bias vectors for each hidden unit, and it is feasible to compute and invert exact Fisher blocks for these.

Kronecker factors can sometimes be shared by approximate Fisher blocks for two or more parameters. This is the case, for example, when a vector of units serves as inputs to two different weight-matrix multiplication operations. In such cases, the computation of the second-order statistics can be reused, which is what we do in our implementation.

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 a summation whose inputs are computed by the same type of operators. Without additional knowledge about the computation graph, one approximation is to treat the individual gradient contributions in the summation as statistically independent of each other (similarly to how gradient contributions from multiple spatial locations are treated as independent in the KFC approximation (Grosse and Martens, 2016)). Under this approximation, the Kronecker factors associated with the gradient can be computed by lumping the statistics associated with each of the gradient contributions together.

Our implementation of Distributed K-FAC in TensorFlow applies the above the strategy to automatically generate K-FAC computation graphs without requiring the user to modify their existing model-definition code.