A Kronecker-factored approximate Fisher matrix for convolution layers

Roger Grosse RGROSSE@CS.TORONTO.EDU
James Martens JMARTENS@CS.TORONTO.EDU
Department of Computer Science, University of Toronto

Abstract

Second-order optimization methods such as natural gradient descent have the potential to speed up training of neural networks by correcting for the curvature of the loss function. Unfortunately, the exact natural gradient is impractical to compute for large models, and most approximations either require an expensive iterative procedure or make crude approximations to the curvature. We present Kronecker Factors for Convolution (KFC), a tractable approximation to the Fisher matrix for convolutional networks based on a structured probabilistic model for the distribution over backpropagated derivatives. Similarly to the recently proposed Kronecker-Factored Approximate Curvature (K-FAC), each block of the approximate Fisher matrix decomposes as the Kronecker product of small matrices, allowing for efficient inversion. KFC captures important curvature information while still yielding comparably efficient updates to stochastic gradient descent (SGD). We show that the updates are invariant to commonly used reparameterizations, such as centering of the activations. In our experiments, approximate natural gradient descent with KFC was able to train convolutional networks several times faster than carefully tuned SGD. Furthermore, it was able to train the networks in 10-20 times fewer *iterations* than SGD, suggesting its potential applicability in a distributed setting.

1. Introduction

Despite advances in optimization, most neural networks are still trained using variants of stochastic gradient descent (SGD) with momentum. It has been suggested that natural gradient descent (Amari, 1998) could greatly speed up optimization because it accounts for the geometry of the optimization landscape and has desirable invariance properties. (See Martens (2014) for a review.) Unfortunately, computing the exact natural gradient is intractable for large networks, as it requires solving a large linear system involving the Fisher matrix, whose dimension is the number of parameters (potentially tens of millions for modern architectures). Approximations to the natural gradient typically either impose very restrictive structure on the Fisher matrix (e.g. LeCun et al., 1998; Le Roux et al., 2008) or require expensive iterative procedures to compute each update, analogously to approximate Newton methods (e.g. Martens, 2010). An ongoing challenge has been to develop a curvature matrix approximation which reflects enough structure to yield high-quality updates, while introducing minimal computational overhead beyond the standard gradient computations.

Much progress in machine learning has been driven by the development of structured probabilistic models whose independence structure allows for efficient computations, yet which still capture important dependencies between the variables of interest. In our case, since the Fisher matrix is the covariance of the backpropagated log-likelihood derivatives, we are interested in modeling the distribution over these derivatives. The model must support efficient computation of the inverse covariance, as this is what's required to compute the natural gradient. Recently, the Factorized Natural Gradient (FANG) (Grosse & Salakhutdinov, 2015) and Kronecker-Factored Approximate Curvature (K-FAC) (Martens & Grosse, 2015) methods exploited probabilistic models of the derivatives to efficiently compute approximate natural gradient updates. In its simplest version, K-FAC approximates each layer-wise block of the Fisher matrix as the Kronecker product of two much smaller matrices. These (very large) blocks can then be can be tractably inverted by inverting each of the two factors. K-FAC was shown to greatly speed up the training of deep autoencoders. However, its underlying probabilistic model assumed fully connected networks with no weight sharing, rendering the method inapplicable to two architectures which have recently revolutionized many applications of machine learning — convolutional networks (LeCun et al., 1989; Krizhevsky et al., 2012) and recurrent neural networks (Hochreiter & Schmidhuber, 1997; Sutskever et al., 2014).

We introduce Kronecker Factors for Convolution (KFC), an approximation to the Fisher matrix for convolutional net-

Proceedings of the 33^{rd} International Conference on Machine Learning, New York, NY, USA, 2016. JMLR: W&CP volume 48. Copyright 2016 by the author(s).

works. Most modern convolutional networks have trainable parameters only in convolutional and fully connected layers. Standard K-FAC can be applied to the latter; our contribution is a factorization of the Fisher blocks corresponding to convolution layers. KFC is based on a structured probabilistic model of the backpropagated derivatives where the activations are independent of the derivatives, the activations and derivatives are spatially homogeneous, and the derivatives are spatially uncorrelated. Under these approximations, we show that the Fisher blocks for convolution layers decompose as a Kronecker product of smaller matrices (analogously to K-FAC), yielding tractable updates.

KFC yields a tractable approximation to the Fisher matrix of a conv net. It can be used directly to compute approximate natural gradient descent updates, as we do in our experiments. One could further combine it with the adaptive step size, momentum, and damping methods from the full K-FAC algorithm (Martens & Grosse, 2015). It could also potentially be used as a pre-conditioner for iterative second-order methods (Martens, 2010; Vinyals & Povey, 2012; Sohl-Dickstein et al., 2014). We show that the approximate natural gradient updates are invariant to widely used reparameterizations of a network, such as whitening or centering of the activations.

We have evaluated our method on training conv nets on object recognition benchmarks. In our experiments, KFC was able to optimize conv nets several times faster than carefully tuned SGD with momentum, in terms of both training and test error. Furthermore, it required 10-20 times fewer *iterations*, suggesting its usefulness in the context of highly distributed training algorithms.

2. Background

In this section, we outline the K-FAC method as previously formulated for standard fully-connected feed-forward networks without weight sharing (Martens & Grosse, 2015). Each layer of a fully connected network computes activations as:

$$\mathbf{s}_\ell = \mathbf{W}_\ell \bar{\mathbf{a}}_{\ell-1} \tag{1}$$

$$\mathbf{a}_\ell = \phi_\ell(\mathbf{s}_\ell), \tag{2}$$

where $\ell \in \{1, \ldots, L\}$ indexes the layer, $\mathbf{s}_\ell$ denotes the inputs to the layer, $\mathbf{a}_\ell$ denotes the activations, $\bar{\mathbf{W}}_\ell = (\mathbf{b}_\ell \ \mathbf{W}_\ell)$ denotes the matrix of biases and weights, $\bar{\mathbf{a}}_\ell = (1 \ \mathbf{a}_\ell^\top)^\top$ denotes the activations with a homogeneous dimension appended, and ϕ_ℓ denotes a nonlinear activation function (usually applied coordinate-wise). (Throughout this paper, we will use the index 0 for all homogeneous coordinates.) We will refer to the values $\mathbf{s}_\ell$ as *pre-activations*. By convention, $\mathbf{a}_0$ corresponds to the inputs $\mathbf{x}$ and $\mathbf{a}_L$ corresponds to the prediction $\mathbf{z}$ made by the network. For convenience, we concatenate all of the parameters of the network into a vector $\boldsymbol{\theta} = (\mathrm{vec}(\mathbf{W}_1)^\top, \ldots, \mathrm{vec}(\mathbf{W}_L)^\top)^\top$, where vec denotes the Kronecker vector operator which stacks the columns of a matrix into a vector. We denote the function computed by the network as $f(\mathbf{x}, \boldsymbol{\theta}) = \mathbf{a}_L$.

Typically, a network is trained to minimize an objective $h(\boldsymbol{\theta})$ given by $\mathcal{L}(\mathbf{y}, f(\mathbf{x}, \boldsymbol{\theta}))$ as averaged over the training set, where $\mathcal{L}(\mathbf{y}, \mathbf{z})$ is a loss function. The gradient ∇h of $h(\boldsymbol{\theta})$, which is required by most optimization methods, is estimated stochastically using mini-batches of training examples. (We will often drop the explicit $\boldsymbol{\theta}$ subscript when the meaning is unambiguous.)

For the remainder of this paper, we will assume the network's prediction $f(\mathbf{x}, \boldsymbol{\theta})$ determines the value of the parameter $\mathbf{z}$ of a distribution $R_{\mathbf{y}|\mathbf{z}}$ over $\mathbf{y}$, and the loss function is the corresponding negative log-likelihood $\mathcal{L}(\mathbf{y}, \mathbf{z}) = -\log r(\mathbf{y}|\mathbf{z})$.

2.1. Second-order optimization of neural networks

Second-order optimization methods work by computing a parameter update $\mathbf{v}$ that minimizes (or approximately minimizes) a local quadratic approximation to the objective, given by $h(\boldsymbol{\theta}) = \nabla_{\boldsymbol{\theta}} h^\top \mathbf{v} + \frac{1}{2}\mathbf{v}^\top \mathbf{C} \mathbf{v}$, where $\mathbf{C}$ is a matrix which quantifies the curvature of the cost function h at $\boldsymbol{\theta}$. The exact solution to this minimization problem can be obtained by solving the linear system $\mathbf{C}\mathbf{v} = -\nabla_{\boldsymbol{\theta}} h$. The original and most well-known example is Newton's method, where $\mathbf{C}$ is chosen to be the Hessian matrix; this isn't appropriate in the non-convex setting because of the well-known problem that it searches for critical points rather than local optima (e.g. Pascanu et al., 2014). Therefore, it is more common to use natural gradient (Amari, 1998) or updates based on the generalized Gauss-Newton matrix (Schraudolph, 2002), which are guaranteed to produce descent directions because the curvature matrix $\mathbf{C}$ is positive semidefinite.

Natural gradient descent can be usefully interpreted as a second-order method (Martens, 2014) where $\mathbf{C}$ is the Fisher information matrix $\mathbf{F}$, as given by

$$\mathbf{F} = \mathbb{E}_{\substack{\mathbf{x} \sim p_{\mathrm{data}} \\ \mathbf{y} \sim R_{\mathbf{y}|f(\mathbf{x}, \boldsymbol{\theta})}}} \left[\mathcal{D}\boldsymbol{\theta} (\mathcal{D}\boldsymbol{\theta})^\top \right], \tag{3}$$

where p_{data} denotes the training distribution, $R_{\mathbf{y}|f(\mathbf{x},\boldsymbol{\theta})}$ denotes the model's predictive distribution, and $\mathcal{D}\boldsymbol{\theta} = \nabla_{\boldsymbol{\theta}} \mathcal{L}(\mathbf{y}, f(\mathbf{x}, \boldsymbol{\theta}))$ is the log-likelihood gradient. For the remainder of this paper, all expectations are with respect to this distribution (which we term the *model's distribution*), so we will leave off the subscripts. (In this paper, we will use the $\mathcal{D}$ notation for log-likelihood derivatives; derivatives of other functions will be written out explicitly.) In the case where $R_{\mathbf{y}|\mathbf{z}}$ corresponds to an exponential family model with "natural" parameters given by $\mathbf{z}$, $\mathbf{F}$ is equivalent to the generalized Gauss-Newton matrix (Martens, 2014), which is an approximation of the Hessian which has also seen extensive use in various neural-network optimization methods (e.g. Martens, 2010; Vinyals & Povey, 2012).

$\mathbf{F}$ is an $n \times n$ matrix, where n is the number of parameters and can be in the tens of millions for modern deep architectures. Therefore, it is impractical to represent $\mathbf{F}$

explicitly in memory, let alone solve the linear system exactly. There are two general strategies one typically takes to find a good search direction: either impose a structure on $\mathbf{F}$ enabling fast inversion (e.g. LeCun et al., 1998; Le Roux et al., 2008; Grosse & Salakhutdinov, 2015), or use an iterative procedure to approximately solve the linear system (e.g. Martens, 2010). These two strategies are not mutually exclusive: tractable curvature approximations can be used as preconditioners in second order optimization, and this has been observed to make a large difference (Martens, 2010; Chapelle & Erhan, 2011; Vinyals & Povey, 2012).

2.2. Kronecker-factored approximate curvature

Kronecker-factored approximate curvature (K-FAC; Martens & Grosse, 2015) is a recently proposed optimization method for neural networks which can be seen as a hybrid of the two approximation strategies: it uses a tractable approximation to the Fisher matrix $\mathbf{F}$, but also uses an optimization strategy which behaves locally like conjugate gradient. This section gives a conceptual summary of the aspects of K-FAC relevant to the contributions of this paper; a precise description of the full algorithm is given in Appendix B.2.

The block-diagonal version of K-FAC (which is the simpler of the two versions, and is what we will present here) is based on two approximations to $\mathbf{F}$ which together make it tractable to invert. First, weight derivatives in different layers are assumed to be uncorrelated, which corresponds to $\mathbf{F}$ being block diagonal, with one block per layer. Each block is given by $\mathbb{E}[\mathrm{vec}(\mathcal{D}\bar{\mathbf{W}}_\ell)\,\mathrm{vec}(\mathcal{D}\bar{\mathbf{W}}_\ell)^\top]$. This approximation by itself is insufficient, because each of the blocks may still be very large. (E.g., if a network has 1,000 units in each layer, each block would be of size $10^6 \times 10^6$.) For the second approximation, observe that

$$\mathbb{E}\left[\mathcal{D}[\bar{\mathbf{W}}_\ell]_{ij}\mathcal{D}[\bar{\mathbf{W}}_\ell]_{i'j'}\right] = \mathbb{E}\left[\mathcal{D}[\mathbf{s}_\ell]_i[\bar{\mathbf{a}}_{\ell-1}]_j\mathcal{D}[\mathbf{s}_\ell]_{i'}[\bar{\mathbf{a}}_{\ell-1}]_{j'}\right].$$

If we approximate the activations and pre-activation derivatives as independent, this can be decomposed as $\mathbb{E}\left[\mathcal{D}[\bar{\mathbf{W}}_\ell]_{ij}\mathcal{D}[\bar{\mathbf{W}}_\ell]_{i'j'}\right] \approx \mathbb{E}\left[\mathcal{D}[\mathbf{s}_\ell]_i\mathcal{D}[\mathbf{s}_\ell]_{i'}\right]\mathbb{E}\left[[\bar{\mathbf{a}}_{\ell-1}]_j[\bar{\mathbf{a}}_{\ell-1}]_{j'}\right]$. This can be written algebraically as a decomposition into a Kronecker product of two smaller matrices:

$$\mathbb{E}[\mathrm{vec}(\bar{\mathbf{W}}_\ell)\,\mathrm{vec}(\bar{\mathbf{W}}_\ell)^\top] \approx \boldsymbol{\Psi}_{\ell-1}\otimes\boldsymbol{\Gamma}_\ell \triangleq \hat{\mathbf{F}}_\ell, \tag{4}$$

where $\boldsymbol{\Psi}_{\ell-1} = \mathbb{E}[\bar{\mathbf{a}}_{\ell-1}\bar{\mathbf{a}}_{\ell-1}^\top]$ and $\boldsymbol{\Gamma}_\ell = \mathbb{E}[\mathbf{s}_\ell\mathbf{s}_\ell^\top]$ denote the second moment matrices of the activations and pre-activation derivatives, respectively. Call the block diagonal approximate Fisher matrix, with blocks given by Eqn. 4, $\hat{\mathbf{F}}$. The two factors are estimated online from the empirical moments of the model's distribution using exponential moving averages.

To invert $\hat{\mathbf{F}}$, we use the facts that (1) we can invert a block diagonal matrix by inverting each of the blocks, and (2) the Kronecker product satisfies the identity $(\mathbf{A}\otimes\mathbf{B})^{-1} = \mathbf{A}^{-1}\otimes\mathbf{B}^{-1}$:

$$\hat{\mathbf{F}}^{-1} = \begin{pmatrix} \boldsymbol{\Psi}_0^{-1}\otimes\boldsymbol{\Gamma}_1^{-1} & & \mathbf{0} \\ & \ddots & \\ \mathbf{0} & & \boldsymbol{\Psi}_{L-1}^{-1}\otimes\boldsymbol{\Gamma}_L^{-1} \end{pmatrix} \tag{5}$$

We do not represent $\hat{\mathbf{F}}^{-1}$ explicitly, as each of the blocks is quite large. Instead, we keep track of each of the Kronecker factors.

The approximate natural gradient $\hat{\mathbf{F}}^{-1}\nabla h$ can then be computed as follows:

$$\hat{\mathbf{F}}^{-1}\nabla h = \begin{pmatrix} \mathrm{vec}\left(\boldsymbol{\Gamma}_1^{-1}(\nabla_{\bar{\mathbf{W}}_1}h)\boldsymbol{\Psi}_0^{-1}\right) \\ \vdots \\ \mathrm{vec}\left(\boldsymbol{\Gamma}_L^{-1}(\nabla_{\bar{\mathbf{W}}_L}h)\boldsymbol{\Psi}_{L-1}^{-1}\right) \end{pmatrix} \tag{6}$$

We would often like to add a multiple of the identity matrix to $\mathbf{F}$ for two reasons. First, many networks are regularized with weight decay, which corresponds to a penalty of $\frac{1}{2}\lambda\boldsymbol{\theta}^\top\boldsymbol{\theta}$, for some parameter λ. Following the interpretation of $\mathbf{F}$ as a quadratic approximation to the curvature, it would be appropriate to use $\mathbf{F}+\lambda\mathbf{I}$ to approximate the curvature of the regularized objective. The second reason is that the local quadratic approximation of h implicitly used when computing the natural gradient may be inaccurate over the region of interest, owing to the approximation of $\mathbf{F}$ by $\hat{\mathbf{F}}$, to the approximation of the Hessian by $\mathbf{F}$, and finally to the error associated with approximating h as locally quadratic in the first place. A common way to address this issue is to damp the updates by adding $\gamma\mathbf{I}$ to the approximate curvature matrix, for some small value γ, before minimizing the local quadratic model. Therefore, we would ideally like to compute $\left[\hat{\mathbf{F}}+(\lambda+\gamma)\mathbf{I}\right]^{-1}\nabla h$.

Unfortunately, adding $(\lambda+\gamma)\mathbf{I}$ breaks the Kronecker factorization structure. While it is possible to exactly solve the damped system (see Appendix B.2), it is often preferable to approximate $\hat{\mathbf{F}}+(\lambda+\gamma)\mathbf{I}$ in a way that maintains the factorizaton structure. Martens & Grosse (2015) pointed out that

$$\hat{\mathbf{F}}_\ell+(\lambda+\gamma)\mathbf{I} \approx \left(\boldsymbol{\Psi}_{\ell-1}+\pi_\ell\sqrt{\lambda+\gamma}\,\mathbf{I}\right)\otimes\left(\boldsymbol{\Gamma}_\ell+\frac{1}{\pi_\ell}\sqrt{\lambda+\gamma}\,\mathbf{I}\right). \tag{7}$$

We will denote this damped approximation as $\hat{\mathbf{F}}_\ell^{(\gamma)} = \boldsymbol{\Psi}_{\ell-1}^{(\gamma)}\otimes\boldsymbol{\Gamma}_\ell^{(\gamma)}$. Mathematically, π_ℓ can be any positive scalar, but Martens & Grosse (2015) suggest the formula

$$\pi_\ell = \sqrt{\frac{\|\boldsymbol{\Psi}_{\ell-1}\otimes\mathbf{I}\|}{\|\mathbf{I}\otimes\boldsymbol{\Gamma}_\ell\|}}, \tag{8}$$

where $\|\cdot\|$ denotes some matrix norm, as this value minimizes the norm of the residual in Eqn. 7. In this work, we use the trace norm $\|\mathbf{B}\| = \mathrm{tr}\,\mathbf{B}$. The approximate natural gradient $\hat{\nabla} h$ is then computed as:

$$\hat{\nabla} h \triangleq [\hat{\mathbf{F}}^{(\gamma)}]^{-1} \nabla h = \begin{pmatrix} \mathrm{vec}\left([\mathbf{\Gamma}_1^{(\gamma)}]^{-1} (\nabla_{\bar{\mathbf{W}}_1} h) [\mathbf{\Psi}_0^{(\gamma)}]^{-1} \right) \\ \vdots \\ \mathrm{vec}\left([\mathbf{\Gamma}_L^{(\gamma)}]^{-1} (\nabla_{\bar{\mathbf{W}}_L} h) [\mathbf{\Psi}_{L-1}^{(\gamma)}]^{-1} \right) \end{pmatrix} \tag{9}$$

The algorithm as presented by Martens & Grosse (2015) has many additional elements which are orthogonal to the contributions of this paper. For concision, a full description of the algorithm is relegated to Appendix B.2.

2.3. Convolutional networks

Convolutional networks can require somewhat crufty notation when the computations are written out in full. In our case, we are interested in computing correlations of derivatives, which compounds the notational difficulties. In this section, we summarize the notation we use. (Table 1 lists all convolutional network notation used in this paper.) In sections which focus on a single layer of the network, we drop the explicit layer indices.

A convolution layer takes as input a layer of activations $\{a_{j,t}\}$, where $j \in \{1, \ldots, J\}$ indexes the input map and $t \in \mathcal{T}$ indexes the spatial location. (Here, $\mathcal{T}$ is the set of spatial locations, which is typically a 2-D grid. For simplicity, we assume convolution is performed with a stride of 1 and padding equal to R, so that the set of spatial locations is shared between the input and output feature maps.) This layer is parameterized by a set of weights $w_{i,j,\delta}$ and biases b_i, where $i \in \{1, \ldots, I\}$ indexes the output map, j indexes the input map, and $\delta \in \Delta$ indexes the spatial offset (from the center of the filter). If the filters are of size $(2R+1) \times (2R+1)$, then we would have $\Delta = \{-R, \ldots, R\} \times \{-R, \ldots, R\}$. We denote the numbers of spatial locations and spatial offsets as $|\mathcal{T}|$ and $|\Delta|$, respectively. The convolution layer computes a set of pre-activations $\{s_{i,t}\}$ as follows:

$$s_{i,t} = \sum_{\delta \in \Delta} w_{i,j,\delta} a_{j,t+\delta} + b_i, \tag{10}$$

where b_i denotes the bias parameter. The activations are defined to take the value 0 outside of $\mathcal{T}$. The pre-activations are passed through a nonlinearity such as ReLU to compute the output layer activations, but we have no need to refer to this explicitly when analyzing a single layer. (For simplicity, we assume operations such as pooling and response normalization are implemented as separate layers.)

Pre-activation derivatives $\mathcal{D}s_{i,t}$ are computed during backpropagation. One then computes weight derivatives as:

$$\mathcal{D}w_{i,j,\delta} = \sum_{t \in \mathcal{T}} a_{j,t+\delta} \mathcal{D}s_{i,t}. \tag{11}$$

In some cases, it is useful to introduce vectorized notation for conv nets. We will represent the activations for a layer ℓ as a $|\mathcal{T}| \times J$ matrix $\mathbf{A}_\ell$ and the preactivations as a $|\mathcal{T}| \times I$ matrix $\mathbf{S}_\ell$. The weights are represented as a $I \times |\Delta| J$ matrix $\mathbf{W}_\ell$.

2.3.1. Efficient implementation and vectorized notation

For modern large-scale vision applications, it's necessary to implement conv nets efficiently for a GPU (or some other massively parallel computing architecture). Since one contribution of our own work was to exploit the same underlying implementation to efficiently compute the statistics needed by our algorithm, we outline a typical GPU implementation of a conv net. As a bonus, discussing the implementation gives us a convenient high-level notation for analyzing conv nets mathematically. Due to space constrants, we relegate this material to Appendix A. This appendix also contains a table of all conv net notation used in this paper.

3. Kronecker factorization for convolution layers

We begin by assuming a block-diagonal approximation to the Fisher matrix like that of K-FAC, where each block contains all the parameters relevant to one layer (see Section 2.2). (Recall that these blocks are typically too large to invert exactly, or even represent explicitly, which is why the further Kronecker approximation is required.) The Kronecker factorization from K-FAC applies only to fully connected layers. Convolutional networks introduce several kinds of layers not found in fully connected feed-forward networks: convolution, pooling, and response normalization. Since pooling and response normalization layers don't have trainable weights, they are not included in the Fisher matrix. However, we must deal with convolution layers. In this section, we present our main contribution, an approximate Kronecker factorization for the blocks of $\hat{\mathbf{F}}$ corresponding to convolution layers. In the tradition of fast food puns (Ranzato & Hinton, 2010; Yang et al., 2014), we call our method Kronecker Factors for Convolution (KFC).

For this section, we focus on the Fisher block for a single layer, so we drop the layer indices. All conv net notation is summarized in Appendix A.

Recall that the Fisher matrix $\mathbf{F} = \mathbb{E}\left[\mathcal{D}\boldsymbol{\theta}(\mathcal{D}\boldsymbol{\theta})^\top\right]$ is the covariance of the log-likelihood gradient under the model's distribution. (In this paper, all expectations are with respect to the model's distribution unless otherwise specified.) By plugging in Eqn. 11, the entries corresponding to weight derivatives are given by:

$$\mathbb{E}[\mathcal{D}w_{i,j,\delta}\mathcal{D}w_{i',j',\delta'}] = \mathbb{E}\left[\left(\sum_{t \in \mathcal{T}} a_{j,t+\delta} \mathcal{D}s_{i,t}\right) \left(\sum_{t' \in \mathcal{T}} a_{j',t'+\delta'} \mathcal{D}s_{i',t'}\right)\right] \tag{12}$$

To think about the computational complexity of computing the entries directly, consider the second convolution layer of AlexNet (Krizhevsky et al., 2012), which has 48 input feature maps, 128 output feature maps, $27 \times 27 = 729$ spatial locations, and 5×5 filters. Since there are $128 \times 48 \times 5 \times 5 = 245760$ weights and 128 biases, the full block would require $245888^2 \approx 60.5$ billion entries to represent explicitly, and inversion is clearly impractical.

Recall that K-FAC approximation for classical fully connected networks can be derived by approximating activations and pre-activation derivatives as being statistically independent (this is the **IAD** approximation below). Deriving an analogous Fisher approximation for convolution layers will require some additional approximations.

Here are the approximations we will make in deriving our Fisher approximation:

- **Independent activations and derivatives (IAD).** The activations are independent of the pre-activation derivatives, *i.e.* $\{a_{j,t}\} \perp\!\!\!\perp \{\mathcal{D}s_{i,t'}\}$.

- **Spatial homogeneity (SH).** The first-order statistics of the activations are independent of spatial location. The second-order statistics of the activations and pre-activation derivatives at any two spatial locations t and t' depend only on $t' - t$. This implies there are functions M, Ω and Γ such that:

$$\mathbb{E}\left[a_{j,t}\right] = M(j) \tag{13}$$
$$\mathbb{E}\left[a_{j,t} a_{j',t'}\right] = \Omega(j, j', t' - t) \tag{14}$$
$$\mathbb{E}\left[\mathcal{D}s_{i,t} \mathcal{D}s_{i',t'}\right] = \Gamma(i, i', t' - t). \tag{15}$$

  Note that $\mathbb{E}[\mathcal{D}s_{i,t}] = 0$ under the model's distribution, so $\mathrm{Cov}\left(\mathcal{D}s_{i,t}, \mathcal{D}s_{i',t'}\right) = \mathbb{E}\left[\mathcal{D}s_{i,t}\mathcal{D}s_{i',t'}\right]$.

- **Spatially uncorrelated derivatives (SUD).** The pre-activation derivatives at any two distinct spatial locations are uncorrelated, *i.e.* $\Gamma(i, i', \delta) = 0$ for $\delta \neq 0$.

We believe **SH** is fairly innocuous, as one is implicitly making a spatial homogeneity assumption when choosing to use convolution in the first place. **SUD** perhaps sounds like a more severe approximation, but in fact appeared to describe the model's distribution quite well in the networks we investigated; this is analyzed empirically in Section 5.1.

We now show that combining the above three approximations yields a Kronecker factorization of the Fisher blocks. For simplicity of notation, assume the data are two-dimensional, so that the offsets can be parameterized with indices $\delta = (\delta_1, \delta_2)$ and $\delta' = (\delta'_1, \delta'_2)$, and denote the dimensions of the activations map as (T_1, T_2). The formulas can be generalized to data dimensions higher than 2 in the obvious way. For clarity, we leave out the bias parameters in this section, but these are discussed in Appendix E.

Theorem 1. *Combining approximations* **IAD**, **SH**, *and* **SUD** *yields the following factorization:*

$$\mathbb{E}\left[\mathcal{D}w_{i,j,\delta}\mathcal{D}w_{i',j',\delta'}\right] = \beta(\delta, \delta')\, \Omega(j, j', \delta' - \delta)\, \Gamma(i, i', 0), \tag{16}$$

where

$$\beta(\delta, \delta') \triangleq (T_1 - \max(\delta_1, \delta'_1, 0) + \min(\delta_1, \delta'_1, 0)) \cdot \\ \cdot (T_2 - \max(\delta_2, \delta'_2, 0) + \min(\delta_2, \delta'_2, 0)) \tag{17}$$

Proof. See Appendix E. □

To talk about how this fits in to the block diagonal approximation to the Fisher matrix $\mathbf{F}$, we now restore the explicit layer indices and use the vectorized notation from Section 2.3.1. The above factorization yields a Kronecker factorization of each block, which will be useful for computing their inverses (and ultimately our approximate natural gradient). In particular, if $\hat{\mathbf{F}}_\ell \approx \mathbb{E}[\mathrm{vec}(\mathcal{D}\bar{\mathbf{W}}_\ell)\, \mathrm{vec}(\mathcal{D}\bar{\mathbf{W}}_\ell)^\top]$ denotes the block of the approximate Fisher for layer ℓ, Eqn. 16 yields our KFC factorization of $\hat{\mathbf{F}}_\ell$ into a Kronecker product of smaller factors:

$$\hat{\mathbf{F}}_\ell = \mathbf{\Omega}_{\ell-1} \otimes \mathbf{\Gamma}_\ell, \tag{18}$$

where

$$[\mathbf{\Omega}_{\ell-1}]_{j|\Delta|+\delta,\, j'|\Delta|+\delta'} \triangleq \beta(\delta, \delta')\, \Omega(j, j', \delta' - \delta) \\ [\mathbf{\Gamma}_\ell]_{i,i'} \triangleq \Gamma(i, i', 0). \tag{19}$$

(We will derive much simpler formulas for $\mathbf{\Omega}_{\ell-1}$ and $\mathbf{\Gamma}_\ell$ in the next section.) Using this factorization, the rest of the K-FAC algorithm can be carried out without modification. For instance, we can compute the approximate natural gradient using a damped version of $\hat{\mathbf{F}}$ analogously to Eqns. 7 and 9 of Section 2.2:

$$\hat{\mathbf{F}}_\ell^{(\gamma)} = \mathbf{\Omega}_{\ell-1}^{(\gamma)} \otimes \mathbf{\Gamma}_\ell^{(\gamma)} \tag{20}$$

$$\triangleq \left(\mathbf{\Omega}_{\ell-1} + \pi_\ell \sqrt{\lambda + \gamma}\, \mathbf{I}\right) \otimes \\ \otimes \left(\mathbf{\Gamma}_\ell + \frac{1}{\pi_\ell}\sqrt{\lambda + \gamma}\, \mathbf{I}\right). \tag{21}$$

$$\hat{\nabla} h = [\hat{\mathbf{F}}^{(\gamma)}]^{-1} \nabla h = \begin{pmatrix} \mathrm{vec}\left([\mathbf{\Gamma}_1^{(\gamma)}]^{-1}(\nabla_{\bar{\mathbf{W}}_1} h)[\mathbf{\Omega}_0^{(\gamma)}]^{-1}\right) \\ \vdots \\ \mathrm{vec}\left([\mathbf{\Gamma}_L^{(\gamma)}]^{-1}(\nabla_{\bar{\mathbf{W}}_L} h)[\mathbf{\Omega}_{L-1}^{(\gamma)}]^{-1}\right) \end{pmatrix} \tag{22}$$

Returning to our running example of AlexNet, $\bar{\mathbf{W}}_\ell$ is a $I \times (J|\Delta| + 1) = 128 \times 1201$ matrix. Therefore the factors $\mathbf{\Omega}_{\ell-1}$ and $\mathbf{\Gamma}_\ell$ are 1201×1201 and 128×128, respectively. These matrices are small enough that they can be represented exactly and inverted in a reasonable amount of time, allowing us to efficiently compute the approximate natural gradient direction using Eqn. 22.

3.1. Estimating the factors

Since the true covariance statistics are unknown, we estimate them empirically by sampling from the model's distribution, similarly to Martens & Grosse (2015). To sample derivatives from the model's distribution, we select a mini-batch, sample the outputs from the model's predictive distribution, and backpropagate the derivatives.

We need to estimate the Kronecker factors $\{\mathbf{\Omega}_\ell\}_{\ell=0}^{L-1}$ and $\{\mathbf{\Gamma}_\ell\}_{\ell=1}^{L}$. Since these matrices are defined in terms of the autocovariance functions Ω and Γ, it would appear natural to estimate these functions empirically. Unfortunately, if the empirical autocovariances are plugged into Eqn. 19, the resulting $\mathbf{\Omega}_\ell$ may not be positive semidefinite. This is a problem, since negative eigenvalues in the approximate Fisher could cause the optimization to diverge (a phenomenon we have observed in practice).

Instead, we estimate each $\mathbf{\Omega}_\ell$ directly using the following fact:

Theorem 2. *Under assumption* **SH***,*

$$\begin{aligned}\mathbf{\Omega}_\ell &= \mathbb{E}\left[[\![\mathbf{A}_\ell]\!]_H^\top [\![\mathbf{A}_\ell]\!]_H\right] \\ \mathbf{\Gamma}_\ell &= \frac{1}{|\mathcal{T}|}\mathbb{E}\left[\mathcal{D}\mathbf{S}_\ell^\top \mathcal{D}\mathbf{S}_\ell\right].\end{aligned} \tag{23}$$

(The $[\![\cdot]\!]$ notation is defined in Appendix A.)

Proof. See Appendix E. □

We maintain exponential moving averages of the covariance statistics, where the empirical statistics are computed on each mini-batch using these formulas.

3.2. Using KFC in optimization

So far, we have defined an approximation $\hat{\mathbf{F}}^{(\gamma)}$ to the Fisher matrix $\mathbf{F}$ which can be tractably inverted. This can be used in any number of ways in the context of optimization, most simply by using $\hat{\nabla} h = [\hat{\mathbf{F}}^{(\gamma)}]^{-1}\nabla h$ as an approximation to the natural gradient $\mathbf{F}^{-1}\nabla h$. Alternatively, we could use it in the context of the full K-FAC algorithm, or as a preconditioner for iterative second-order methods (Martens, 2010; Vinyals & Povey, 2012; Sohl-Dickstein et al., 2014).

In our experiments, we explored two particular instantiations of KFC in optimization algorithms. First, in order to provide as direct a comparison as possible to standard SGD-based optimization, we used $\hat{\nabla} h$ in the context of a generic approximate natural gradient descent procedure; this procedure is like SGD, except that $\hat{\nabla} h$ is substituted for the Euclidean gradient. Additionally, we used momentum, update clipping, and parameter averaging — all standard techniques in the context of stochastic optimization.[1] One can also view this as a preconditioned SGD method, where $\hat{\mathbf{F}}^{(\gamma)}$ is used as the preconditioner. Therefore, we refer to this method in our experiments as KFC-pre (to distinguish it from the KFC approximation itself). This method is spelled out in detail in Appendix B.1.

We also explored the use of $\hat{\mathbf{F}}^{(\gamma)}$ in the context of the full K-FAC training procedure (see Appendix B.2). Since this performed about the same as KFC-pre, we report results only for KFC-pre.

With the exception of inverting the Kronecker factors, all of the heavy computation for our methods was performed on the GPU. We based our implementation on CUDAMat (Mnih, 2009) and the convolution kernels provided by the Toronto Deep Learning ConvNet (TDLCN) package (Srivastava, 2015). Full details on our GPU implementation and other techniques for minimizing computational overhead are given in Appendix B.3.

4. Theoretical analysis

4.1. Invariance

Natural gradient descent is motivated partly by way of its invariance to reparameterization: regardless of how the model is parameterized, the updates are equivalent to the first order. Approximations to natural gradient don't satisfy full invariance to parameterization, but certain approximations have been shown to be invariant to more limited, but still fairly broad, classes of transformations (Ollivier, 2015; Martens & Grosse, 2015). For instance, K-FAC was shown to be invariant to affine transformations of the activations (Martens & Grosse, 2015).

For convolutional layers, we cannot expect an algorithm to be invariant to arbitrary affine transformations of a given layer's activations, as such transformations can change the set of functions which are representable. (Consider for instance, a transformation which permutes the spatial locations.) However, we show that the KFC updates are invariant to homogeneous, *pointwise* affine transformations of the activations, both before and after the nonlinearity. This is perhaps an overly limited statement, as it doesn't use the fact that the algorithm accounts for spatial correlations. However, it still accounts for a broad set of transformations, such as normalizing activations to be zero mean and unit variance either before or after the nonlinearity.

To formalize this, recall that a layer's activations are represented as a $|\mathcal{T}| \times J$ matrix and are computed from that layer's pre-activations by way of an elementwise nonlinearity, i.e. $\mathbf{A}_\ell = \phi_\ell(\mathbf{S}_\ell)$. We replace this with an activation function $\phi_\ell^\dagger$ which additionally computes affine transformations before and after the nonlinearity. Such transformations can be represented in matrix form:

$$\mathbf{A}_\ell^\dagger = \phi_\ell^\dagger(\mathbf{S}_\ell^\dagger) = \phi_\ell(\mathbf{S}_\ell^\dagger \mathbf{U}_\ell + \mathbf{1}\mathbf{c}_\ell^\top)\mathbf{V}_\ell + \mathbf{1}\mathbf{d}_\ell^\top, \tag{24}$$

where $\mathbf{U}_\ell$ and $\mathbf{V}_\ell$ are invertible matrices, and $\mathbf{c}_\ell$ and $\mathbf{d}_\ell$

[1] Our SGD baseline used momentum and parameter averaging as well. Clipping was not needed for SGD, for reasons explained in Appendix B.1.

are vectors. For convenience, the inputs to the network can be treated as an activation function ϕ_0 which takes no arguments. We also assume the final layer outputs are not transformed, i.e. $\mathbf{V}_L = \mathbf{I}$ and $\mathbf{d}_L = \mathbf{0}$. KFC is invariant to this class of transformations:

Theorem 3. *Let $\mathcal{N}$ be a network with parameter vector $\boldsymbol{\theta}$ and activation functions $\{\phi_\ell\}_{\ell=0}^L$. Given activation functions $\{\phi_\ell^\dagger\}_{\ell=0}^L$ defined as in Eqn. 24, there exists a parameter vector $\boldsymbol{\theta}^\dagger$ such that a network $\mathcal{N}^\dagger$ with parameters $\boldsymbol{\theta}^\dagger$ and activation functions $\{\phi_\ell^\dagger\}_{\ell=0}^L$ computes the same function as $\mathcal{N}$. The KFC updates on $\mathcal{N}$ and $\mathcal{N}^\dagger$ are equivalent, in that the resulting networks compute the same function.*

Proof. See Appendix E. □

Invariance to affine transformations also implies approximate invariance to smooth nonlinear transformations; see Martens (2014) for further discussion.

4.2. Relationship with other algorithms

It is possible to interpret many other neural net optimization methods as structured probabilistic approximations to natural gradient. This includes coordinatewise rescaling methods (e.g. LeCun et al., 1998; Duchi et al., 2011; Tieleman & Hinton, 2012; Zeiler, 2013; Kingma & Ba, 2015), centering of activations (Cho et al., 2013; Vatanen et al., 2013; Ioffe & Szegedy, 2015, e.g.), and the recently proposed Projected Natural Gradient (Desjardins et al., 2015). This allows us to compare the modeling assumptions implicitly made by different methods. See Appendix C for a full discussion.

5. Experiments

We have evaluated our method on two standard image recognition benchmark datasets: CIFAR-10 (Krizhevsky, 2009), and Street View Housing Numbers (SVHN; Netzer et al., 2011). Our aim is not to achieve state-of-the-art performance, but to evaluate KFC's ability to optimize previously published architectures. We first examine the probabilistic assumptions, and then present optimization results.

For CIFAR-10, we used the architecture from `cuda-convnet`[2] which achieved 18% error in 20 minutes. This network consists of three convolution layers and a fully connected layer. (While `cuda-convnet` provides some better-performing architectures, we could not use these, since these included locally connected layers, which KFC can't handle.) For SVHN, we used the architecture of Srivastava (2013). This architecture consists of three convolutional layers followed by three fully connected layers, and uses dropout for regularization. Both of these architectures were carefully tuned for their respective tasks. Furthermore, the TDLCN CUDA kernels we used were carefully tuned at a low level to implement SGD updates efficiently for both of these architectures. Therefore, we believe our SGD baseline is quite strong.

5.1. Evaluating the probabilistic modeling assumptions

One of the benefits of using a structured probabilistic model to approximate the Fisher matrix is that we can analyze whether the modeling assumptions are satisfied. As discussed above, **IAD** is the standard approximation made by standard K-FAC, and was discussed in detail both theoretically and empirically by Martens & Grosse (2015). One implicitly assumes **SH** when choosing to use a convolutional architecture. However, **SUD** is perhaps less intuitive. Why should we suppose the derivatives are spatially uncorrelated? Conversely, why not go a step further and assume the *activations* are spatially uncorrelated (as do some methods; see Appendix C) or even drop all of the correlations (thereby obtaining a much simpler diagonal approximation to the Fisher matrix)?

Appendix D.1 analyzes empirically the validity of assumption **SUD** on conv nets trained to CIFAR-10 and SVHN. We conclude that **SUD** appears to describe the model distributions quite well for both networks. By contrast, the networks' activations have very strong spatial correlations, so it is significant that KFC does not assume spatially uncorrelated activations.

5.2. Optimization performance

We evaluated KFC-pre in the context of optimizing deep convolutional networks. We compared against stochastic gradient descent (SGD) with momentum, which is widely considered a strong baseline for training conv nets. All architectural choices (e.g. sizes of layers) were kept consistent with the previously published configurations. Since the focus of this work is optimization rather than generalization, metaparameters were tuned with respect to *training* error. This protocol was favorable to the SGD baseline, as the learning rates which performed the best on training error also performed the best on test error.[3] We tuned the learning rates from the set $\{0.3, 0.1, 0.03, \ldots, 0.0003\}$ separately for each experiment. For KFC-pre, we also chose several algorithmic parameters using the method of Appendix B.3, which considers only per-epoch running time and not final optimization performance.[4]

[2] `https://code.google.com/p/cuda-convnet/`

[3] For KFC-pre, we encountered a more significant tradeoff between training and test error, most notably in the choice of mini-batch size, so the presented results do not reflect our best runs on the test set. For instance, as reported in Figure 1, the test error on CIFAR-10 leveled off at 18.5% after 5 minutes, after which the network started overfitting. When we reduced the mini-batch size from 512 to 128, the test error reached 17.5% after 5 minutes and 16% after 35 minutes. However, this run performed far worse on the training set. On the flip side, very large mini-batch sizes hurt generalization for both methods, as discussed in Section 5.3.

[4] For SGD, we used a momentum parameter of 0.9 and mini-batches of size 128, which match the previously published config-

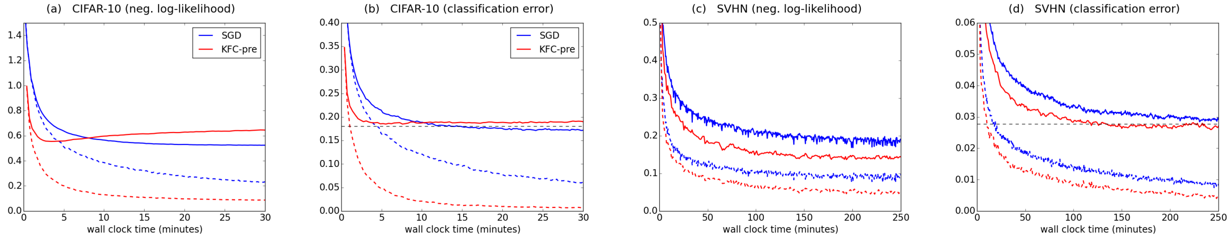


Figure 1. Optimization performance of KFC-pre and SGD. **(a)** CIFAR-10, negative log-likelihood. **(b)** CIFAR-10, classification error. **(c)** SVHN, negative log-likelihood. **(d)** SVHN, classification error. **Solid lines** represent test error and **dashed lines** represent training error. The **horizontal dashed line** represents the previously reported test error for the same architecture.

For both SGD and KFC-pre, we used an exponential moving average of the iterates (see Appendix B.1) with a timescale of 50,000 training examples (which corresponds to one epoch on CIFAR-10). This helped both SGD and KFC-pre substantially. All experiments for which wall clock time is reported were run on a single Nvidia GeForce GTX Titan Z GPU board.

As baselines, we also tried Adagrad (Duchi et al., 2011), RMSProp (Tieleman & Hinton, 2012), and Adam (Kingma & Ba, 2015), but none of these approaches outperformed carefully tuned SGD with momentum. This is consistent with the observations of Kingma & Ba (2015).

Figure 1(a,b) shows the optimization performance on the CIFAR-10 dataset, in terms of wall clock time. Both KFC-pre and SGD reached approximately the previously published test error of 18% before they started overfitting. However, KFC-pre reached 19% test error in 3 minutes, compared with 9 minutes for SGD. The difference in training error was more significant: KFC-pre reaches a training error of 6% in 4 minutes, compared with 30 minutes for SGD. On SVHN, KFC-pre reached the previously published test error of 2.78% in 120 minutes, while SGD did not reach it within 250 minutes. (As discussed above, test error comparisons should be taken with a grain of salt.)

Appendix D.2 analyzes the performance of KFC-pre in relation to the recently proposed batch normalization method (Ioffe & Szegedy, 2015).

5.3. Potential for distributed implementation

Much work has been devoted recently to highly parallel or distributed implementations of neural network optimization (e.g. Dean et al. (2012)). Synchronous SGD effectively allows one to use very large mini-batches efficiently, which helps optimization by reducing the variance in the stochastic gradient estimates. However, the per-update performace levels off to that of batch SGD once the variance is no longer significant and curvature effects come to dominate. Asynchronous SGD partially alleviates this issue by using new network parameters as soon as they become available, but needing to compute gradients with stale parameters limits the benefits of this approach.

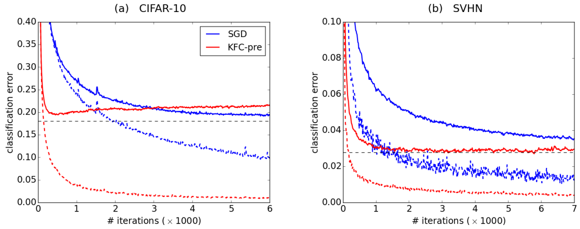


Figure 2. Classification error as a function of the number of iterations (weight updates). Heuristically, this is a rough measure of how the algorithms might perform in a highly distributed setting. **(a)** CIFAR-10. **(b)** SVHN. See Figure 1 caption for details.

As a proxy for how the algorithms are likely to perform in a highly distributed setting[5], we measured the classification error as a function of the *number of iterations* (weight updates) for each algorithm. Both algorithms were run with large mini-batches of size 4096 (in place of 128 for SGD and 512 for KFC-pre). Figure 2 shows training curves for both algorithms on CIFAR-10 and SVHN, using the same architectures as above.[6] KFC-pre required far fewer weight updates to achieve good training and test error compared with SGD. For instance, on CIFAR-10, KFC-pre obtained a training error of 10% after 300 updates, compared with 6000 updates for SGD, a 20-fold improvement. Similar speedups were obtained on test error and on the SVHN dataset. These results suggest that a distributed implementation of KFC-pre has the potential to obtain large speedups over distributed SGD-based algorithms.

urations. For KFC-pre, we used a momentum parameter of 0.9, mini-batches of size 512, and a damping parameter $\gamma = 10^{-3}$. In both cases, our informal explorations did not find other values which performed substantially better in terms of training error.

[5] The gradient computations can be farmed out to worker nodes, exactly as with SGD, and we expect the computations of Kronecker factors and their inverses can be performed asynchronously. Therefore, we would not expect additional sequential bottlenecks or communication overhead.

[6] Both SGD and KFC-pre reached a slightly worse test error before they started overfitting, compared with the small-minibatch experiments of the previous section. This is because large mini-batches lose the regularization benefit of stochastic gradients. One would need to adjust the regularizer in order to get good generalization performance in this setting.

Acknowledgments

We gratefully acknowledge support from Google, NSERC, and the Banting Research Foundation. We thank Jimmy Ba, Guillaume Desjardins, Geoffrey Hinton, and Nitish Srivastava for helpful discussions.

References

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

Chapelle, O. and Erhan, D. Improved preconditioner for Hessian-free optimization. In *NIPS Workshop on Deep Learning and Unsupervised Feature Learning*, 2011.

Chellapilla, K., Puri, S., and Simard, P. High performance convolutional neural networks for document processing. In *International Workshop on Frontiers in Handwriting Recognition*, 2006.

Cho, K., Raiko, T., and Ilin, A. Enhanced gradient for training restricted Boltzmann machines. *Neural Computation*, 25:805–813, 2013.

Dean, J., Corrado, G. S., Monga, R., Chen, K., Devin, M., Le, Q. V., Mao, M. Z., Ranzato, M., Senior, A., Tucker, P., Yang, K., and Ng, A. Y. Large scale distributed deep networks. In *Neural Information Processing Systems*, 2012.

Demmel, J. W. *Applied Numerical Linear Algebra*. SIAM, 1997.

Desjardins, G., Simonyan, K., Pascanu, R., and Kavukcuoglu, K. Natural neural networks. arXiv:1507.00210, 2015.

Duchi, J., Hazan, E., and Singer, Y. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12:2121–2159, 2011.

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

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

Hochreiter, S. and Schmidhuber, J. Long short-term memory. *Neural Computation*, 9:1735–1780, 1997.

Ioffe, S. and Szegedy, C. Batch normalization: accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning*, 2015.

Kingma, D. P. and Ba, J. L. Adam: a method for stochastic optimization. In *International Conference on Learning Representations*, 2015.

Krizhevsky, A. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. ImageNet classification with deep convolutional neural networks. In *Neural Information Processing Systems*, 2012.

Le Roux, Nicolas, Manzagol, Pierre-antoine, and Bengio, Yoshua. Topmoumoute online natural gradient algorithm. In *Advances in Neural Information Processing Systems 20*, pp. 849–856. MIT Press, 2008.

LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackel, L. D. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1:541–551, 1989.

LeCun, Y., Bottou, L., Orr, G., and Müller, K. Efficient backprop. *Neural networks: Tricks of the trade*, pp. 546–546, 1998.

Martens, J. Deep learning via Hessian-free optimization. In *Proceedings of the 27th International Conference on Machine Learning (ICML)*, 2010.

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

Martens, J. and Grosse, R. Optimizing neural networks with Kronecker-factored approximate curvature. In *International Conference on Machine Learning*, 2015.

Mnih, V. CUDAMat: A CUDA-based matrix class for Python. Technical Report 004, University of Toronto, 2009.

Moré, J.J. The Levenberg-Marquardt algorithm: implementation and theory. *Numerical analysis*, pp. 105–116, 1978.

Netzer, Y., Wang, T., Coates, A., Bissacco, A., Wu, B., and Ng, A. Y. Reading digits in natural images with unsupervised feature learning. In *Neural Information Processing Systems Deep Learning and Unsupervised Feature Learning Workshop*, 2011.

Nocedal, Jorge and Wright, Stephen J. *Numerical optimization*. Springer, 2. ed. edition, 2006.

Ollivier, Y. Riemannian metrics for neural networks I: feedforward networks. *Information and Inference*, 4(2):108–153, 2015.

Pascanu, R., Mikolov, T., and Bengio, Y. On the difficulty of training recurrent neural networks. In *International Conference on Machine Learning*, 2013.

Pascanu, R., Dauphin, Y. N., Ganguli, S., and Bengio, Y. On the saddle point problem for non-convex optimization. arXiv:1405.4604, 2014.

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

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

Ranzato, M. and Hinton, G. E. Modeling pixel means and covariances using factorized third-order Boltzmann machines. In *Computer Vision and Pattern Recognition*, 2010.

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

Simoncelli, E. P. and Olshausen, B. A. Natural image statistics and neural representation. *Annual Review of Neuroscience*, 24: 1193–1216, 2001.

Sohl-Dickstein, J., Poole, B., and Ganguli, S. Fast large-scale optimization by unifying stochastic gradient and quasi-Newton methods. In *International Conference on Machine Learning*, 2014.

Srivastava, N. Improving neural networks with dropout. Master's thesis, University of Toronto, 2013.

Srivastava, N. Toronto Deep Learning ConvNet. https://github.com/TorontoDeepLearning/convnet/, 2015.

Sutskever, I., Vinyals, O., and Le, Q. V. V. Sequence to sequence learning with neural networks. In *Neural Information Processing Systems*, 2014.

Swersky, K., Chen, Bo, Marlin, B., and de Freitas, N. A tutorial on stochastic approximation algorithms for training restricted Boltzmann machines and deep belief nets. In *Information Theory and Applications Workshop (ITA), 2010*, pp. 1–10, Jan 2010.

Tieleman, T. and Hinton, G. Lecture 6.5, RMSProp. In Coursera course Neural Networks for Machine Learning, 2012.

Vatanen, Tommi, Raiko, Tapani, Valpola, Harri, and LeCun, Yann. Pushing stochastic gradient towards second-order methods – backpropagation learning with transformations in nonlinearities. 2013.

Vinyals, O. and Povey, D. Krylov subspace descent for deep learning. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2012.

Yang, Z., Moczulski, M., Denil, M., de Freitas, N., Smola, A., Song, L., and Wang, Z. Deep fried convnets. arXiv:1412.7149, 2014.

Zeiler, Matthew D. ADADELTA: An adaptive learning rate method. 2013.

.

A. Conv net notation and GPU implementation

For modern large-scale vision applications, it's necessary to implement conv nets efficiently for a GPU (or some other massively parallel computing architecture). We provide a very brief overview of the low-level efficiency issues which are relevant to K-FAC. We base our discussion on the Toronto Deep Learning ConvNet (TDLCN) package (Srivastava, 2015), whose convolution kernels we use in our experiments. Like many modern implementations, this implementation follows the approach of Chellapilla et al. (2006), which reduces the convolution operations to large matrix-vector products in order to exploit memory locality and efficient parallel BLAS operators. We describe the implementation explicitly, as it is important that our proposed algorithm be efficient using the same memory layout (shuffling operations are extremely expensive). As a bonus, these vectorized operations provide a convenient high-level notation which we will use throughout the paper.

The ordering of arrays in memory is significant, as it determines which operations can be performed efficiently without requiring (very expensive) transpose operations. The activations are stored as a $M \times |\mathcal{T}| \times J$ array $\tilde{\mathbf{A}}_{\ell-1}$, where M is the mini-batch size, $|\mathcal{T}|$ is the number of spatial locations, and J is the number of feature maps.[7] This can be interpreted as an $M|\mathcal{T}| \times J$ matrix. (We must assign orderings to $\mathcal{T}$ and Δ, but this choice is arbitrary.) Similarly, the weights are stored as an $I \times |\Delta| \times J$ array $\mathbf{W}_\ell$, which can be interpreted either as an $I \times |\Delta| J$ matrix or a $I|\Delta| \times J$ matrix without reshuffling elements in memory. We will almost always use the former interpretation, which we denote $\mathbf{W}_\ell$; the $I|\Delta| \times J$ matrix will be denoted $\breve{\mathbf{W}}_\ell$.

The naive implementation of convolution, while highly parallel in principle, suffers from poor memory locality. Instead, efficient implementations typically use what we will term the *expansion operator* and denote $[\![\cdot]\!]$. This operator extracts the patches surrounding each spatial location and flattens them into vectors. These vectors become the rows of a matrix. For instance, $[\![\tilde{\mathbf{A}}_{\ell-1}]\!]$ is a $M|\mathcal{T}| \times J|\Delta|$ matrix, defined as

$$[\![\tilde{\mathbf{A}}_{\ell-1}]\!]_{tM+m,\, j|\Delta|+\delta} = [\tilde{\mathbf{A}}_{\ell-1}]_{(t+\delta)M+m,\, j} = a^{(m)}_{j,t+\delta}, \tag{25}$$

for all entries such that $t + \delta \in \mathcal{T}$. All other entries are defined to be 0. Here, m indexes the data instance within the mini-batch.

In TDLCN, the forward pass is computed as

$$\tilde{\mathbf{A}}_\ell = \phi(\tilde{\mathbf{S}}_\ell) = \phi\left([\![\tilde{\mathbf{A}}_{\ell-1}]\!]\mathbf{W}_\ell^\top + \mathbf{1}\mathbf{b}_\ell^\top\right), \tag{26}$$

where ϕ is the nonlinearity, applied elementwise, $\mathbf{1}$ is a vector of ones, and $\mathbf{b}$ is the vector of biases. In backpropagation, the activation derivatives are computed as:

$$\mathcal{D}\tilde{\mathbf{A}}_{\ell-1} = [\![\mathcal{D}\tilde{\mathbf{S}}_\ell]\!]\breve{\mathbf{W}}_\ell. \tag{27}$$

Finally, the gradient for the weights is computed as

$$\mathcal{D}\mathbf{W}_\ell = \mathcal{D}\tilde{\mathbf{S}}_\ell^\top [\![\tilde{\mathbf{A}}_{\ell-1}]\!] \tag{28}$$

The matrix products are computed using the cuBLAS function `cublasSgemm`. In practice, the expanded matrix $[\![\tilde{\mathbf{A}}_{\ell-1}]\!]$ may be too large to store in memory. In this case, a subset of the rows of $[\![\tilde{\mathbf{A}}_{\ell-1}]\!]$ are computed and processed at a time.

We will also use the $|\mathcal{T}| \times J$ matrix $\mathbf{A}_{\ell-1}$ and the $|\mathcal{T}| \times I$ matrix $\mathbf{S}_\ell$ to denote the activations and pre-activations for a single training case. $\mathbf{A}_{\ell-1}$ and $\mathbf{S}_\ell$ can be substituted for $\tilde{\mathbf{A}}_{\ell-1}$ and $\tilde{\mathbf{S}}_\ell$ in Eqns. 26-28.

For fully connected networks, it is often convenient to append a homogeneous coordinate to the activations so that the biases can be folded into the weights (see Section 2.2). For convolutional layers, there is no obvious way to add extra activations such that the convolution operation simulates the effect of biases. However, we can achieve an analogous effect by adding a homogeneous coordinate (i.e. a column of all 1's) to the *expanded* activations. We will denote this $[\![\tilde{\mathbf{A}}_{\ell-1}]\!]_H$. Similarly, we can prepend the bias vector to the weights matrix: $\bar{\mathbf{W}}_\ell = (\mathbf{b}_\ell\ \mathbf{W}_\ell)$. The homogeneous coordinate is not typically used in conv net implementations, but it will be convenient for us notationally. For instance, the forward pass can be written as:

$$\tilde{\mathbf{A}}_\ell = \phi\left([\![\tilde{\mathbf{A}}_{\ell-1}]\!]_H \bar{\mathbf{W}}_\ell^\top\right) \tag{29}$$

Table 1 summarizes all of the conv net notation used in this paper.

B. Optimization methods

B.1. KFC as a preconditioner for SGD

The first optimization procedure we used in our experiments was a generic natural gradient descent approximation, where $\hat{\mathbf{F}}^{(\gamma)}$ was used to approximate $\mathbf{F}$. This procedure is like SGD with momentum, except that $\hat{\nabla} h$ is substituted for the Euclidean gradient. One can also view this as a preconditioned SGD method, where $\hat{\mathbf{F}}^{(\gamma)}$ is used as the preconditioner. To distinguish this optimization procedure from the KFC approximation itself, we refer to it as KFC-pre. Our procedure is perhaps more closely analogous to earlier Kronecker product-based natural gradient approximations (Heskes, 2000; Povey et al., 2015) than to K-FAC itself.

In addition, we used a variant of gradient clipping (Pascanu et al., 2013) to avoid instability. In particular, we clipped the approximate natural gradient update $\mathbf{v}$ so that

[7] The first index of the array is the least significant in memory.

j	input map index
J	number of input maps
i	output map index
I	number of output maps
$T_1 \times T_2$	feature map dimension
t	spatial location index
$\mathcal{T}$	set of spatial locations $= \{1, \ldots, T_1\} \times \{1, \ldots, T_2\}$
R	radius of filters
δ	spatial offset
Δ	set of spatial offsets (in a filter) $= \{-R, \ldots, R\} \times \{-R, \ldots, R\}$
$\delta = (\delta_1, \delta_2)$	explicit 2-D parameterization (δ_1 and δ_2 run from $-R$ to R)
$a_{j,t}$	input layer activations
$s_{i,t}$	output layer pre-activations
$\mathcal{D}s_{i,t}$	the loss derivative $\partial \mathcal{L} / \partial s_{i,t}$
ϕ	activation function (nonlinearity)
$w_{i,j,\delta}$	weights
b_i	biases
$M(j)$	mean activation
$\Omega(j, j', \delta)$	uncentered autocovariance of activations
$\Gamma(i, i', \delta)$	autocovariance of pre-activation derivatives
$\beta(\delta, \delta')$	function defined in Theorem 1

$\otimes$	Kronecker product
vec	Kronecker vector operator
ℓ	layer index
L	number of layers
M	size of a mini-batch
$\mathbf{A}_\ell$	activations for a data instance
$\tilde{\mathbf{A}}_\ell$	activations for a mini-batch
$[\![\mathbf{A}_\ell]\!]$	expanded activations
$[\![\mathbf{A}_\ell]\!]_H$	expanded activations with homogeneous coordinate
$\mathbf{S}_\ell$	pre-activations for a data instance
$\tilde{\mathbf{S}}_\ell$	pre-activations for a mini-batch
$\mathcal{D}\mathbf{S}_\ell$	the loss gradient $\nabla_{\mathbf{S}_\ell} \mathcal{L}$
$\boldsymbol{\theta}$	vector of trainable parameters
$\mathbf{W}_\ell$	weight matrix
$\mathbf{b}_\ell$	bias vector
$\bar{\mathbf{W}}_\ell$	combined parameters = $(\mathbf{b}_\ell \; \mathbf{W}_\ell)$
$\mathbf{F}$	exact Fisher matrix
$\hat{\mathbf{F}}$	approximate Fisher matrix
$\hat{\mathbf{F}}_\ell$	diagonal block of $\hat{\mathbf{F}}$ for layer ℓ
$\boldsymbol{\Omega}_\ell$	Kronecker factor for activations
$\boldsymbol{\Gamma}_\ell$	Kronecker factor for derivatives
λ	weight decay parameter
γ	damping parameter
$\hat{\mathbf{F}}^{(\gamma)}$	damped approximate Fisher matrix
$\boldsymbol{\Omega}_\ell^{(\gamma)}, \boldsymbol{\Gamma}_\ell^{(\gamma)}$	damped Kronecker factors

Table 1. Summary of convolutional network notation used in this paper. The left column focuses on a single convolution layer, which convolves its "input layer" activations with a set of filters to produce the pre-activations for the "output layer." Layer indices are omitted for clarity. The right column considers the network as a whole, and therefore includes explicit layer indices.

$\nu \triangleq \mathbf{v}^\top \mathbf{F} \mathbf{v} < 0.3$, where $\mathbf{F}$ is estimated using 1/4 of the training examples from the current mini-batch. One motivation for this heuristic is that ν approximates the KL divergence of the predictive distributions before and after the update, and one wouldn't like the predictive distributions to change too rapidly. The value ν can be computed using curvature-vector products (Schraudolph, 2002). The clipping was only triggered near the beginning of optimization, where the parameters (and hence also the curvature) tended to change rapidly.[8] Therefore, one can likely eliminate this step by initializing from a model partially trained using SGD.

Taking inspiration from Polyak averaging (Polyak & Juditsky, 1992; Swersky et al., 2010), we used an exponential moving average of the iterates. This helps to smooth out the variability caused by the mini-batch selection. The full optimization procedure is given in Algorithm 1.

B.2. Kronecker-factored approximate curvature

The central idea of K-FAC is the combination of approximations to the Fisher matrix described in Section 2.2. While one could potentially perform standard natural gradient descent using the approximate natural gradient $\hat{\nabla} h$, perhaps with a fixed learning rate and with fixed Tikhonov-style damping/reglarization, Martens & Grosse (2015) found that the most effective way to use $\hat{\nabla} h$ was within a robust 2nd-order optimization framework based on adaptively damped quadratic models, similar to the one employed in HF (Martens, 2010). In this section, we describe the K-FAC method in detail, while omitting certain aspects of the method which we do not use, such as the block tri-diagonal inverse approximation.

K-FAC uses a quadratic model of the objective to dynamically choose the step size α and momentum decay parameter μ at each step. This is done by taking $\mathbf{v} = \alpha \hat{\nabla} h + \mu \mathbf{v}_{prev}$ where $\mathbf{v}_{prev}$ is the update computed at the previous iteration, and minimizing the following quadratic model of the objective (over the current mini-batch):

$$M(\boldsymbol{\theta} + \mathbf{v}) = h(\boldsymbol{\theta}) + \nabla h^\top \mathbf{v} + \frac{1}{2} \mathbf{v}^\top (\mathbf{F} + r\mathbf{I}) \mathbf{v}. \quad (30)$$

where we assume the h is the expected loss plus an ℓ_2-regularization term of the form $\frac{r}{2} \|\boldsymbol{\theta}\|^2$. Since $\mathbf{F}$ behaves like a curvature matrix, this quadratic function is similar to the second-order Taylor approximation to h. Note that here we use the *exact* $\mathbf{F}$ for the mini-batch, rather than the approximation $\hat{\mathbf{F}}$. Intuitively, one can think of $\mathbf{v}$ as being itself iteratively optimized at each step in order to better minimize M, or in other words, to more closely match the true

[8] This may be counterintuitive, since SGD applied to neural nets tends to take small steps early in training, at least for commonly used initializations. For SGD, this happens because the initial parameters, and hence also the initial curvature, are relatively small in magnitude. Our method, which corrects for the curvature, takes larger steps early in training, when the error signal is the largest.

Algorithm 1 Using KFC as a preconditioner for SGD

```
Require: initial network parameters $\boldsymbol{\theta}^{(0)}$
         weight decay penalty $\lambda$
         learning rate $\alpha$
         momentum parameter $\mu$ (suggested value: 0.9)
         parameter averaging timescale $\tau$ (suggested value: number of mini-batches in the dataset)
         damping parameter $\gamma$ (suggested value: $10^{-3}$, but this may require tuning)
         statistics update period $T_s$ (see Appendix B.3)
         inverse update period $T_f$ (see Appendix B.3)
         clipping parameter $C$ (suggested value: 0.3)
  $k \leftarrow 0$
  $\mathbf{p} \leftarrow \mathbf{0}$
  $\xi \leftarrow e^{-1/\tau}$
  $\bar{\boldsymbol{\theta}}^{(0)} \leftarrow \boldsymbol{\theta}^{(0)}$
  Estimate the factors $\{\boldsymbol{\Omega}_\ell\}_{\ell=0}^{L-1}$ and $\{\boldsymbol{\Gamma}_\ell\}_{\ell=1}^{L}$ on the full dataset using Eqn. 23
  Compute the inverses $\{[\boldsymbol{\Omega}_\ell^{(\gamma)}]^{-1}\}_{\ell=0}^{L-1}$ and $\{[\boldsymbol{\Gamma}_\ell^{(\gamma)}]^{-1}\}_{\ell=1}^{L}$ using Eqn. 21
  while stopping criterion not met do
    $k \leftarrow k + 1$
    Select a new mini-batch

    if $k \equiv 0 \pmod{T_s}$ then
      Update the factors $\{\boldsymbol{\Omega}_\ell\}_{\ell=0}^{L-1}$ and $\{\boldsymbol{\Gamma}_\ell\}_{\ell=1}^{L}$ using Eqn. 23
    end if
    if $k \equiv 0 \pmod{T_f}$ then
      Compute the inverses $\{[\boldsymbol{\Omega}_\ell^{(\gamma)}]^{-1}\}_{\ell=0}^{L-1}$ and $\{[\boldsymbol{\Gamma}_\ell^{(\gamma)}]^{-1}\}_{\ell=1}^{L}$ using Eqn. 21
    end if

    Compute $\nabla h$ using backpropagation
    Compute $\hat{\nabla} h = [\hat{\mathbf{F}}^{(\gamma)}]^{-1} \nabla h$ using Eqn. 22
    $\mathbf{v} \leftarrow -\alpha \hat{\nabla} h$

    $\{$Clip the update if necessary$\}$
    Estimate $\nu = \mathbf{v}^\top \mathbf{F} \mathbf{v} + \lambda \mathbf{v}^\top \mathbf{v}$ using a subset of the current mini-batch
    if $\nu > C$ then
      $\mathbf{v} \leftarrow \mathbf{v} / \sqrt{\nu / C}$
    end if

    $\mathbf{p}^{(k)} \leftarrow \mu \mathbf{p}^{(k-1)} + \mathbf{v}$ $\{$Update momentum$\}$
    $\boldsymbol{\theta}^{(k)} \leftarrow \boldsymbol{\theta}^{(k-1)} + \mathbf{p}^{(k)}$ $\{$Update parameters$\}$
    $\bar{\boldsymbol{\theta}}^{(k)} \leftarrow \xi \bar{\boldsymbol{\theta}}^{(k-1)} + (1 - \xi) \boldsymbol{\theta}^{(k)}$ $\{$Parameter averaging$\}$
  end while
  return Averaged parameter vector $\bar{\boldsymbol{\theta}}^{(k)}$
```

natural gradient (which is the exact minimum of M). Interestingly, in full batch mode, this method is equivalent to performing preconditioned conjugate gradient in the vicinity of a local optimum (where $\mathbf{F}$ remains approximately constant).

To see how this minimization over α and μ can be done efficiently, without computing the entire matrix $\mathbf{F}$, consider the general problem of minimizing M on the subspace spanned by arbitrary vectors $\{\mathbf{v}_1, \ldots, \mathbf{v}_R\}$. (In our case, $R = 2$, $\mathbf{v}_1 = \hat{\nabla} h$ and $\mathbf{v}_2 = \mathbf{v}_{prev}$.) The coefficients $\boldsymbol{\alpha}$ can be found by solving the linear system $\mathbf{C}\boldsymbol{\alpha} = -\mathbf{d}$, where $\mathbf{C}_{ij} = \mathbf{v}_i^\top \mathbf{F} \mathbf{v}_j$ and $\mathbf{d}_i = \nabla h^\top \mathbf{v}_i$. To compute the matrix $\mathbf{C}$, we compute each of the matrix-vector products $\mathbf{F}\mathbf{v}_j$ using automatic differentiation (Schraudolph, 2002).

Both the approximate natural gradient $\hat{\nabla} h$ and the update $\mathbf{v}$ (generated as described above) arise as the minimum, or approximate minimum, of a corresponding quadratic model. In the case of $\mathbf{v}$, this model is given by M and is designed to be a good local approximation to the objective h. Meanwhile, the quadratic model which is implicitly minimized when computing $\hat{\nabla} h$ is designed to approximate M (by approximating $\mathbf{F}$ with $\hat{\mathbf{F}}$).

Because these quadratic models are approximations, naively minimizing them over $\mathbb{R}^n$ can lead to poor results in both theory and practice. To help deal with this problem K-FAC employs an adaptive Tikhonov-style damping scheme applied to each of them (the details of which differ in either case).

To compensate for the inaccuracy of M as a model of h, K-FAC adds a Tikhonov regularization term $\frac{\lambda}{2}\|\mathbf{v}\|^2$ to M which encourages the update to remain small in magnitude, and thus more likely to remain in the region where M is a reasonable approximation to h. This amounts to replacing r with $r + \lambda$ in Eqn. 30. Note that this technique is formally equivalent to performing constrained minimization of M within some spherical region around $\mathbf{v} = 0$ (a "trust-region"). See for example Nocedal & Wright (2006).

K-FAC uses the well-known Levenberg-Marquardt technique (Moré, 1978) to automatically adapt the damping parameter λ so that the damping is loosened or tightened depending on how accurately $M(\boldsymbol{\theta} + \mathbf{v})$ predicts the true decrease in the objective function after each step. This accuracy is measured by the so-called "reduction ratio", which is given by

$$\rho = \frac{h(\boldsymbol{\theta}) - h(\boldsymbol{\theta} + \mathbf{v})}{M(\boldsymbol{\theta}) - M(\boldsymbol{\theta} + \mathbf{v})}, \tag{31}$$

and should be close to 1 when the quadratic approximation is reasonably accurate around the given value of $\boldsymbol{\theta}$. The update rule for λ is as follows:

$$\lambda \leftarrow \begin{cases} \lambda \cdot \lambda_- & \text{if } \rho > 3/4 \\ \lambda & \text{if } 1/4 \leq \rho \leq 3/4 \\ \lambda \cdot \lambda_+ & \text{if } \rho < 1/4 \end{cases} \tag{32}$$

where λ_+ and λ_- are constants such that $\lambda_- < 1 < \lambda_+$.

To compensate for the inaccuracy of $\hat{\mathbf{F}}$, and encourage $\hat{\nabla} h$ to be smaller and more conservative, K-FAC similarly adds $\gamma \mathbf{I}$ to $\hat{\mathbf{F}}$ before inverting it. As discussed in Section 2.2, this can be done approximately by adding multiples of $\mathbf{I}$ to each of the Kronecker factors $\boldsymbol{\Psi}_\ell$ and $\boldsymbol{\Gamma}_\ell$ of $\hat{\mathbf{F}}_\ell$ before inverting them. Alternatively, an exact solution can be obtained by expanding out the eigendecomposition of each block $\hat{\mathbf{F}}_\ell$ of $\hat{\mathbf{F}}$, and using the following identity:

$$\left[\hat{\mathbf{F}}_\ell + \gamma \mathbf{I}\right]^{-1} = \left[(\mathbf{Q}_{\boldsymbol{\Psi}} \otimes \mathbf{Q}_{\boldsymbol{\Gamma}})(\mathbf{D}_{\boldsymbol{\Psi}} \otimes \mathbf{D}_{\boldsymbol{\Gamma}})\left(\mathbf{Q}_{\boldsymbol{\Psi}}^\top \otimes \mathbf{Q}_{\boldsymbol{\Gamma}}^\top\right) + \gamma \mathbf{I}\right]^{-1} \tag{33}$$

$$= \left[(\mathbf{Q}_{\boldsymbol{\Psi}} \otimes \mathbf{Q}_{\boldsymbol{\Gamma}})(\mathbf{D}_{\boldsymbol{\Psi}} \otimes \mathbf{D}_{\boldsymbol{\Gamma}} + \gamma \mathbf{I})\left(\mathbf{Q}_{\boldsymbol{\Psi}}^\top \otimes \mathbf{Q}_{\boldsymbol{\Gamma}}^\top\right)\right]^{-1} \tag{34}$$

$$= (\mathbf{Q}_{\boldsymbol{\Psi}} \otimes \mathbf{Q}_{\boldsymbol{\Gamma}})(\mathbf{D}_{\boldsymbol{\Psi}} \otimes \mathbf{D}_{\boldsymbol{\Gamma}} + \gamma \mathbf{I})^{-1}\left(\mathbf{Q}_{\boldsymbol{\Psi}}^\top \otimes \mathbf{Q}_{\boldsymbol{\Gamma}}^\top\right), \tag{35}$$

where $\boldsymbol{\Psi}_\ell = \mathbf{Q}_{\boldsymbol{\Psi}} \mathbf{D}_{\boldsymbol{\Psi}} \mathbf{Q}_{\boldsymbol{\Psi}}^\top$ and $\boldsymbol{\Gamma}_\ell = \mathbf{Q}_{\boldsymbol{\Gamma}} \mathbf{D}_{\boldsymbol{\Gamma}} \mathbf{Q}_{\boldsymbol{\Gamma}}^\top$ are the orthogonal eigendecompositions of $\boldsymbol{\Psi}_\ell$ and $\boldsymbol{\Gamma}_\ell$ (which are symmetric PSD). These manipulations are based on well-known properties of the Kronecker product which can be found in, e.g., Demmel (1997, sec. 6.3.3). Matrix-vector products $(\hat{\mathbf{F}} + \gamma \mathbf{I})^{-1} \nabla h$ can then be computed from the above identity using the following block-wise formulas:

$$\mathbf{V}_1 = \mathbf{Q}_{\boldsymbol{\Gamma}}^\top (\nabla_{\bar{\mathbf{W}}_\ell} h) \mathbf{Q}_{\boldsymbol{\Psi}} \tag{36}$$

$$\mathbf{V}_2 = \mathbf{V}_1 / (\mathbf{d}_{\boldsymbol{\Gamma}} \mathbf{d}_{\boldsymbol{\Psi}}^\top + \gamma) \tag{37}$$

$$(\hat{\mathbf{F}}_\ell + \gamma \mathbf{I})^{-1} \operatorname{vec}(\nabla_{\bar{\mathbf{W}}_\ell} h) = \operatorname{vec}\left(\mathbf{Q}_{\boldsymbol{\Gamma}} \mathbf{V}_2 \mathbf{Q}_{\boldsymbol{\Psi}}^\top\right), \tag{38}$$

where $\mathbf{d}_{\boldsymbol{\Gamma}}$ and $\mathbf{d}_{\boldsymbol{\Psi}}$ are the diagonals of $\mathbf{D}_{\boldsymbol{\Gamma}}$ and $\mathbf{D}_{\boldsymbol{\Psi}}$ and the division and addition in Eqn. 37 are both elementwise.

One benefit of this damping strategy is that it automatically accounts for the curvature contributed by both the quadratic damping term $\frac{\lambda}{2}\|\mathbf{v}\|^2$ and the weight decay penalty $\frac{r}{2}\|\boldsymbol{\theta}\|^2$ if these are used. Heuristically, one could even set $\gamma = \sqrt{\lambda + r}$, which can sometimes perform well. One should always choose γ at least this large. However, it may sometimes be advantageous to choose γ significantly larger, as $\hat{\mathbf{F}}$ might not be a good approximation to $\mathbf{F}$, and the damping may help reduce the impact of directions erroneously estimated to have low curvature. For consistency with Martens & Grosse (2015), we adopt their method of automatically adapting γ. In particular, each time we adapt γ, we compute $\hat{\nabla} h$ for three different values $\gamma_- < \gamma < \gamma_+$. We choose whichever of the three values results in the lowest value of $M(\boldsymbol{\theta} + \mathbf{v})$.

B.3. Efficient implementation

We based our implementation on the Toronto Deep Learning ConvNet (TDLCN) package (Srivastava, 2015), which is a Python wrapper around CUDA kernels. We needed to write a handful of additional kernels:

- a kernel for computing $\hat{\boldsymbol{\Omega}}_\ell$ (Eqn. 23)

- kernels which performed forward mode automatic differentiation for the max-pooling and response normalization layers

Most of the other operations for KFC could be performed on the GPU using kernels provided by TDLCN. The only exception is computing the inverses $\{[\mathbf{\Omega}_\ell^{(\gamma)}]^{-1}\}_{\ell=0}^{L-1}$ and $\{[\mathbf{\Gamma}_\ell^{(\gamma)}]^{-1}\}_{\ell=1}^{L}$, which was done on the CPU. (The forward mode kernels are only used in update clipping; as mentioned above, one can likely eliminate this step in practice by initializing from a partially trained model.)

KFC introduces several sources of overhead per iteration compared with SGD:

- Updating the factors $\{\mathbf{\Omega}_\ell\}_{\ell=0}^{L-1}$ and $\{\mathbf{\Gamma}_\ell\}_{\ell=1}^{L}$
- Computing the inverses $\{[\mathbf{\Omega}_\ell^{(\gamma)}]^{-1}\}_{\ell=0}^{L-1}$ and $\{[\mathbf{\Gamma}_\ell^{(\gamma)}]^{-1}\}_{\ell=1}^{L}$
- Computing the approximate natural gradient $\hat{\nabla} h = [\hat{\mathbf{F}}^{(\gamma)}]^{-1}\nabla h$
- Estimating $\nu = \mathbf{v}^\top \mathbf{F} \mathbf{v} + \lambda \mathbf{v}^\top \mathbf{v}$ (which is used for gradient clipping)

The overhead from the first two could be reduced by only periodically recomputing the factors and inverses, rather than doing so after every mini-batch. The cost of estimating $\mathbf{v}^\top \mathbf{F} \mathbf{v}$ can be reduced by using only a subset of the mini-batch. These shortcuts did not seem to hurt the per-epoch progress very much, as one can get away with using quite stale curvature information, and ν is only used for clipping and therefore doesn't need to be very accurate. The cost of computing $\hat{\nabla} h$ is unavoidable, but because it doesn't grow with the size of the mini-batch, its per-epoch cost can be made smaller by using larger mini-batches. (As we discuss further in Section 5.3, KFC can work well with large mini-batches.) These shortcuts introduce several additional hyperparameters, but fortunately these are easy to tune: we simply chose them such that the per-epoch cost of KFC was less than twice that of SGD. This requires only running a profiler for a few epochs, rather than measuring overall optimization performance.

Observe that the inverses $\{[\mathbf{\Omega}_\ell^{(\gamma)}]^{-1}\}_{\ell=0}^{L-1}$ and $\{[\mathbf{\Gamma}_\ell^{(\gamma)}]^{-1}\}_{\ell=1}^{L}$ are computed on the CPU, while all of the other heavy computation is GPU-bound. In principle, since KFC works fine with stale curvature information, the inverses could be computed asychronously while the algorithm is running, thereby making their cost almost free. We did not exploit this in our experiments, however.

C. Relationship with other algorithms

Other neural net optimization methods have been proposed which attempt to correct for various statistics of the activations or gradients. Perhaps the most commonly used are algorithms which attempt to adapt learning rates for individual parameters based on the variance of the gradients (LeCun et al., 1998; Duchi et al., 2011; Tieleman & Hinton, 2012; Zeiler, 2013; Kingma & Ba, 2015). These can be thought of as diagonal approximations to the curvature.

Another class of approaches attempts to reparameterize a network such that its activations have zero mean and unit variance, with the goals of preventing covariate shift and improving the conditioning of the curvature (Cho et al., 2013; Vatanen et al., 2013; Ioffe & Szegedy, 2015). Centering can be viewed as an approximation to natural gradient where the Fisher matrix is approximated with a directed Gaussian graphical model (Grosse & Salakhutdinov, 2015). As discussed in Section 4.1, KFC is invariant to re-centering of activations, so it ought to automatically enjoy the optimization benefits of centering. However, batch normalization (Ioffe & Szegedy, 2015) includes some effects not automatically captured by KFC. First, the normalization is done separately for each mini-batch rather than averaged across mini-batches; this introduces stochasticity into the computations which may serve as a regularizer. Second, it discourages large covariate shifts in the pre-activations, which may help to avoid dead units. Since batch normalization is better regarded as a modification to the architecture than an optimization algorithm, it can be combined with KFC; we investigated this in our experiments.

Projected Natural Gradient (PRONG; Desjardins et al., 2015) goes a step further than centering methods by fully whitening the activations in each layer. In the case of fully connected layers, the activations are transformed to have zero mean and unit covariance. For convolutional layers, they apply a linear transformation that whitens the activations *across feature maps*. While PRONG includes clever heuristics for updating the statistics, it's instructive to consider an idealized version of the method which has access to the exact statistics. We can interpret this idealized PRONG in our own framework as arising from following two additional approximations:

- **Spatially uncorrelated activations (SUA).** The activations at any two distinct spatial locations are uncorrelated, *i.e.* $\mathrm{Cov}(a_{j,t}, a_{j',t'}) = 0$ for $t \neq t'$. Also assuming **SH**, the correlations can then be written as $\mathrm{Cov}(a_{j,t}, a_{j',t}) = \Sigma(j, j')$.
- **White derivatives (WD).** Pre-activation derivatives are uncorrelated and have spherical covariance, i.e. $\Gamma(i, i', \delta) \propto \mathbb{1}_{i=i'} \mathbb{1}_{\delta=0}$. We can assume WLOG that the proportionality constant is 1, since any scalar factor can be absorbed into the learning rate.

Theorem 4. *Combining approximations* **IAD**, **SH**, **SUA**, *and* **WD** *results in the following approximation to the entries of the Fisher matrix:*

$$\mathbb{E}\left[\mathcal{D}w_{i,j,\delta}\mathcal{D}w_{i',j',\delta'}\right] = \beta(\delta,\delta')\,\tilde{\Omega}(j,j',\delta'-\delta)\,\mathbb{1}_{i=i'}, \tag{39}$$

where $\mathbb{1}$ is the indicator function and $\tilde{\Omega}(j, j', \delta) \triangleq \Sigma(j, j')\mathbb{1}_{\delta=0} + M(j)M(j')$ is the uncentered autocovariance function. (β is defined in Theorem 1. Formulas for the remaining entries are given in Appendix E.) If the $\beta(\delta, \delta')$ term is dropped, the resulting approximate natural gradient descent update rule is equivalent to idealized PRONG, up to rescaling.

As we later discuss in Section 5.1, assumption **WD** appears to hold up well empirically, while **SUA** appears to lose a lot of information. Observe, for instance, that the input images are themselves treated as a layer of activations. Assumption **SUA** amounts to modeling each channel of an image as white noise, corresponding to a flat power spectrum. Images have a well-characterized $1/f^p$ power spectrum with $p \approx 2$ (Simoncelli & Olshausen, 2001), which implies that the curvature may be much larger in directions corresponding to low-frequency Fourier components than in directions corresponding to high-frequency components.

D. Experiments

D.1. Evaluating the probabilistic modeling assumptions

In order to analyze the reasonableness of our spatially uncorrelated derivatives (**SUD**) assumption, we investigated the autocorrelation functions for networks trained on CIFAR-10 and SVHN, each with 50 epochs of SGD. (These models were trained long enough to achieve good test error, but not long enough to overfit.) Derivatives were sampled from the model's distribution as described in Section 2.2. Figure 3(a) shows the autocorrelation functions of the pre-activation gradients for three (arbitrary) feature maps in all of the convolution layers of both networks. Figure 3(b) shows the correlations between derivatives for different feature maps in the same spatial position. Evidently, the derivatives are very weakly correlated, both spatially and cross-map, although there are some modest cross-map correlations in the first layers of both models, as well as modest spatial correlations in the top convolution layer of the CIFAR-10 network. This suggests that **SUD** is a good approximation for these networks.

Interestingly, the lack of correlations between derivatives appears to be a result of max-pooling. Max-pooling has a well-known sparsifying effect on the derivatives, as any derivative is zero unless the corresponding activation achieves the maximum within its pooling group. Since neighboring locations are unlikely to simultaneously achieve the maximum, max-pooling weakens the spatial correlations. To test this hypothesis, we trained networks equivalent to those described above, except that the max-pooling layers were replaced with average pooling. The spatial autocorrelations and cross-map correlations are shown in Figure 3(c, d). Replacing max-pooling with average pooling dramatically strengthens both sets of correlations.

In contrast with the derivatives, the activations have very strong correlations, both spatially and cross-map, as shown in Figure 4. This suggests the spatially uncorrelated activations **(SUA)** assumption implicitly made by some algorithms could be problematic, despite appearing superficially analogous to **SUD**.

D.2. Comparison with batch normalization

Batch normalization (BN Ioffe & Szegedy, 2015) has recently had much success at training a variety of neural network architectures. It has been motivated both in terms of optimization benefits (because it reduces covariate shift) and regularization benefits (because it adds stochasticity to the updates). However, BN is best regarded not as an optimization algorithm, but as a modification to the network architecture, and it can be used in conjunction with algorithms other than SGD. We modified the original CIFAR-10 architecture to use batch normalization in each layer. Since the parameters of a batch normalized network would have a different scale from those of an ordinary network, we disabled the ℓ_2 regularization term so that both networks would be optimized to the same objective function. While our own (inefficient) implementation of batch normalization incurred substantial computational overhead, we believe an efficient implementation ought to have very little overhead; therefore, we simulated an efficient implementation by reusing the timing data from the non-batch-normalized networks. Learning rates were tuned separately for all four conditions (similarly to the rest of our experiments).

Training curves are shown in Figure 5. All of the methods achieved worse test error than the original network as a result of ℓ_2 regularization being eliminated. However, the BN networks reached a lower test error than the non-BN networks before they started overfitting, consistent with the stochastic regularization interpretation of BN.[9] For both the BN and non-BN architectures, KFC-pre optimized both the training and test error and NLL considerably faster than SGD. Furthermore, it appeared not to lose the regularization benefit of BN. This suggests that KFC-pre and BN can be combined synergistically.

[9] Interestingly, the BN networks were slower to optimize the training error than their non-BN counterparts. We speculate that this is because (1) the SGD baseline, being carefully tuned, didn't exhibit the pathologies that BN is meant to correct for (i.e. dead units and extreme covariate shift), and (2) the regularization effects of BN made it harder to overfit.

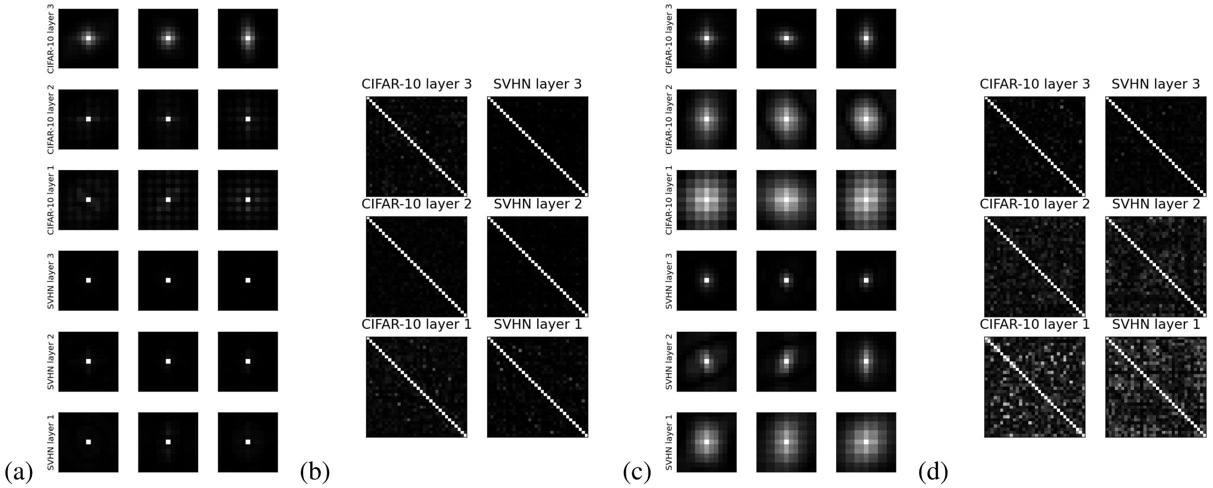


Figure 3. Visualization of the absolute values of the correlations between the pre-activation derivatives for all of the convolution layers of CIFAR-10 and SVHN networks trained with SGD. **(a)** Autocorrelation functions of the derivatives of three feature maps from each layer. **(b)** Cross-map correlations for a single spatial position. **(c, d)** Same as (a) and (b), except that the networks use average pooling rather than max-pooling.

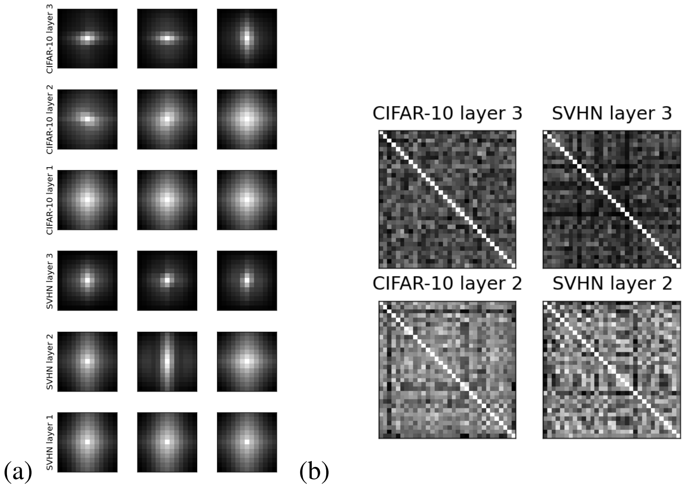
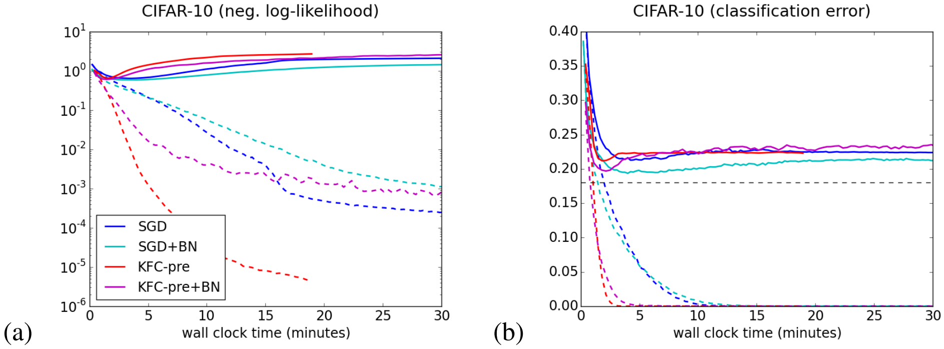


Figure 4. Visualization of the uncentered correlations Ω between activations in all of the convolution layers of the CIFAR-10 and SVHN networks. **(a)** Spatial autocorrelation functions of three feature maps in each layer. **(b)** Correlations of the activations at a given spatial location. The activations have much stronger correlations than the backpropagated derivatives.



(a) (b)

Figure 5. Optimization performance of KFC-pre and SGD on a CIFAR-10 network, with and without batch normalization (BN). **(a)** Negative log-likelihood, on a log scale. **(b)** Classification error. **Solid lines** represent test error and **dashed lines** represent training error. The **horizontal dashed line** represents the previously reported test error for the same architecture. The KFC-pre training curve is cut off because the algorithm became unstable when the training NLL reached 4×10^{-6}.

E. Proofs of theorems

E.1. Proofs for Section 3

Lemma 1. *Under approximation* **IAD***,*

$$\begin{aligned} \mathbb{E}\left[\mathcal{D}w_{i,j,\delta}\mathcal{D}w_{i',j',\delta'}\right] &= \sum_{t\in\mathcal{T}}\sum_{t'\in\mathcal{T}} \mathbb{E}\left[a_{j,t+\delta}a_{j',t'+\delta'}\right]\mathbb{E}\left[\mathcal{D}s_{i,t}\mathcal{D}s_{i',t'}\right] && (40)\\ \mathbb{E}\left[\mathcal{D}w_{i,j,\delta}\mathcal{D}b_{i'}\right] &= \sum_{t\in\mathcal{T}}\sum_{t'\in\mathcal{T}} \mathbb{E}\left[a_{j,t+\delta}\right]\mathbb{E}\left[\mathcal{D}s_{i,t}\mathcal{D}s_{i',t'}\right] && (41)\\ \mathbb{E}\left[\mathcal{D}b_i\mathcal{D}b_{i'}\right] &= |\mathcal{T}|\,\mathbb{E}\left[\mathcal{D}s_{i,t}\mathcal{D}s_{i',t'}\right] && (42)\end{aligned}$$

Proof. We prove the first equality; the rest are analogous.

$$\begin{aligned} \mathbb{E}[\mathcal{D}w_{i,j,\delta}\mathcal{D}w_{i',j',\delta'}] &= \mathbb{E}\left[\left(\sum_{t\in\mathcal{T}} a_{j,t+\delta}\mathcal{D}s_{i,t}\right)\left(\sum_{t'\in\mathcal{T}} a_{j',t'+\delta'}\mathcal{D}s_{i',t'}\right)\right] && (43)\\ &= \mathbb{E}\left[\sum_{t\in\mathcal{T}}\sum_{t'\in\mathcal{T}} a_{j,t+\delta}\mathcal{D}s_{i,t}a_{j',t'+\delta'}\mathcal{D}s_{i',t'}\right] && (44)\\ &= \sum_{t\in\mathcal{T}}\sum_{t'\in\mathcal{T}} \mathbb{E}\left[a_{j,t+\delta}\mathcal{D}s_{i,t}a_{j',t'+\delta'}\mathcal{D}s_{i',t'}\right] && (45)\\ &= \sum_{t\in\mathcal{T}}\sum_{t'\in\mathcal{T}} \mathbb{E}\left[a_{j,t+\delta}a_{j',t'+\delta'}\right]\mathbb{E}\left[\mathcal{D}s_{i,t}\mathcal{D}s_{i',t'}\right] && (46)\end{aligned}$$

Assumption **IAD** is used in the final line. □

Theorem 1. *Combining approximations* **IAD***,* **SH***, and* **SUD** *yields the following factorization:*

$$\begin{aligned} \mathbb{E}\left[\mathcal{D}w_{i,j,\delta}\mathcal{D}w_{i',j',\delta'}\right] &= \beta(\delta,\delta')\,\Omega(j,j',\delta'-\delta)\,\Gamma(i,i',0),\\ \mathbb{E}\left[\mathcal{D}w_{i,j,\delta}\mathcal{D}b_{i'}\right] &= \beta(\delta)\,M(j)\,\Gamma(i,i',0)\\ \mathbb{E}\left[\mathcal{D}b_i\mathcal{D}b_{i'}\right] &= |\mathcal{T}|\,\Gamma(i,i',0) && (47)\end{aligned}$$

where

$$\begin{aligned} \beta(\delta) &\triangleq (T_1-|\delta_1|)\;(T_2-|\delta_2|)\\ \beta(\delta,\delta') &\triangleq (T_1-\max(\delta_1,\delta_1',0)+\min(\delta_1,\delta_1',0))\cdot(T_2-\max(\delta_2,\delta_2',0)+\min(\delta_2,\delta_2',0)) && (48)\end{aligned}$$

Proof.

$$\begin{aligned}
\mathbb{E}[\mathcal{D}w_{i,j,\delta}\mathcal{D}w_{i',j',\delta'}] &= \sum_{t\in\mathcal{T}}\sum_{t'\in\mathcal{T}} \mathbb{E}\left[a_{j,t+\delta}a_{j',t'+\delta'}\right]\mathbb{E}\left[\mathcal{D}s_{i,t}\mathcal{D}s_{i',t'}\right] && (49)\\
&= \sum_{t\in\mathcal{T}}\sum_{t'\in\mathcal{T}} \Omega(j,j',t'+\delta'-t-\delta)\,\mathbb{1}_{\substack{t+\delta\in\mathcal{T}\\ t'+\delta'\in\mathcal{T}}}\,\Gamma(i,i',t'-t) && (50)\\
&= \sum_{t\in\mathcal{T}} \Omega(j,j',\delta'-\delta)\,\mathbb{1}_{\substack{t+\delta\in\mathcal{T}\\ t+\delta'\in\mathcal{T}}}\,\Gamma(i,i',0) && (51)\\
&= |\{t\in\mathcal{T} : t+\delta\in\mathcal{T}, t+\delta'\in\mathcal{T}\}|\;\Omega(j,j',\delta'-\delta)\,\Gamma(i,i',0) && (52)\\
&= \beta(\delta,\delta')\,\Omega(j,j',\delta'-\delta)\,\Gamma(i,i',0) && (53)
\end{aligned}$$

Lines 49, 50, and 51 use Lemma 1 and assumptions **SH**, and **SUD**, respectively. In Line 50, the indicator function (denoted $\mathbb{1}$) arises because the activations are defined to be zero outside the set of spatial locations. The remaining formulas can be derived analogously. □

Theorem 2. *Under assumption* **SH**,

$$\begin{aligned}
\mathbf{\Omega}_\ell &= \mathbb{E}\left[[\![\mathbf{A}_\ell]\!]_H^\top [\![\mathbf{A}_\ell]\!]_H\right] && (54)\\
\mathbf{\Gamma}_\ell &= \frac{1}{|\mathcal{T}|}\mathbb{E}\left[\mathcal{D}\mathbf{S}_\ell^\top \mathcal{D}\mathbf{S}_\ell\right] && (55)
\end{aligned}$$

Proof. In this proof, all activations and pre-activations are taken to be in layer ℓ. The expected entries are given by:

$$\begin{aligned}
\mathbb{E}\left[[\![\mathbf{A}_\ell]\!]_H^\top [\![\mathbf{A}_\ell]\!]_H\right]_{j|\Delta|+\delta,\, j'|\Delta|+\delta} &= \mathbb{E}\left[\sum_{t\in\mathcal{T}} a_{j,t+\delta}a_{j',t+\delta'}\right] && (56)\\
&= \sum_{t\in\mathcal{T}} \mathbb{E}\left[a_{j,t+\delta}a_{j',t+\delta'}\right] && (57)\\
&= \sum_{t\in\mathcal{T}} \Omega(j,j',\delta'-\delta)\,\mathbb{1}_{\substack{t+\delta\in\mathcal{T}\\ t+\delta'\in\mathcal{T}}} && (58)\\
&= |\{t\in\mathcal{T} : t+\delta\in\mathcal{T}, t+\delta'\in\mathcal{T}\}|\;\Omega(j,j',\delta'-\delta) && (59)\\
&= \beta(\delta,\delta')\,\Omega(j,j',\delta'-\delta) && (60)\\
&= [\mathbf{\Omega}_\ell]_{j|\Delta|+\delta,\, j'|\Delta|+\delta'} && (61)
\end{aligned}$$

SH is used in Line 58. Similarly,

$$\begin{aligned}
\mathbb{E}\left[[\![\mathbf{A}_\ell]\!]_H^\top [\![\mathbf{A}_\ell]\!]_H\right]_{0,\, j|\Delta|+\delta} &= \mathbb{E}\left[\sum_{t\in\mathcal{T}} a_{j,t+\delta}\right] && (62)\\
&= \beta(\delta)\,M(j) && (63)\\
&= [\mathbf{\Omega}_\ell]_{0,\, j|\Delta|+\delta} && (64)\\
\left[[\![\mathbf{A}_\ell]\!]_H^\top [\![\mathbf{A}_\ell]\!]_H\right]_{0,\,0} &= |\mathcal{T}| && (65)\\
&= [\mathbf{\Omega}_\ell]_{0,\,0} && (66)\\
\mathbb{E}\left[\mathcal{D}\mathbf{S}_\ell^\top \mathcal{D}\mathbf{S}_\ell\right]_{i,i'} &= \mathbb{E}\left[\sum_{t\in\mathcal{T}} \mathcal{D}s_{i,t}\mathcal{D}s_{i',t}\right] && (67)\\
&= |\mathcal{T}|\,\Gamma(i,i',0) && (68)\\
&= |\mathcal{T}|\;[\mathbf{\Gamma}_\ell]_{i,\,i'} && (69)
\end{aligned}$$

□

E.2. Proofs for Section 4

Preliminaries and notation. In discussing invariances, it will be convenient to add homogeneous coordinates to various matrices:

$$[\mathbf{A}_\ell]_H \triangleq \begin{pmatrix} \mathbf{1} & \mathbf{A}_\ell \end{pmatrix} \tag{70}$$

$$[\mathbf{S}_\ell]_H \triangleq \begin{pmatrix} \mathbf{1} & \mathbf{S}_\ell \end{pmatrix} \tag{71}$$

$$[\bar{\mathbf{W}}_\ell]_H \triangleq \begin{pmatrix} 1 & \\ \mathbf{b}_\ell & \mathbf{W}_\ell \end{pmatrix} \tag{72}$$

We also define the activation function ϕ to ignore the homogeneous column, so that

$$[\mathbf{A}_\ell]_H = \phi([\mathbf{S}_\ell]_H) = \phi(\llbracket \mathbf{A}_{\ell-1} \rrbracket [\bar{\mathbf{W}}_\ell]_H). \tag{73}$$

Using the homogeneous coordinate notation, we can write the effect of the affine transformations on the pre-activations and activations:

$$\begin{aligned} [\mathbf{S}_\ell^\dagger \mathbf{U}_\ell + \mathbf{1}\mathbf{c}_\ell^\top]_H &= [\mathbf{S}_\ell^\dagger]_H [\mathbf{U}_\ell]_H \\ [\mathbf{A}_\ell \mathbf{V}_\ell + \mathbf{1}\mathbf{d}_\ell^\top]_H &= [\mathbf{A}_\ell]_H [\mathbf{V}_\ell]_H, \end{aligned} \tag{74}$$

where

$$[\mathbf{U}_\ell]_H \triangleq \begin{pmatrix} 1 & \mathbf{c}_\ell^\top \\ & \mathbf{U}_\ell \end{pmatrix} \tag{75}$$

$$[\mathbf{V}_\ell]_H \triangleq \begin{pmatrix} 1 & \mathbf{d}_\ell^\top \\ & \mathbf{V}_\ell \end{pmatrix}. \tag{76}$$

The inverse transformations are represented as

$$[\mathbf{U}_\ell]_H^{-1} \triangleq \begin{pmatrix} 1 & -\mathbf{c}_\ell^\top \mathbf{U}_\ell^{-1} \\ & \mathbf{U}_\ell^{-1} \end{pmatrix} \tag{77}$$

$$[\mathbf{V}_\ell]_H^{-1} \triangleq \begin{pmatrix} 1 & -\mathbf{d}_\ell^\top \mathbf{V}_\ell^{-1} \\ & \mathbf{V}_\ell^{-1} \end{pmatrix}. \tag{78}$$

We can also determine the effect of the affine transformation on the *expanded* activations:

$$\llbracket \mathbf{A}_\ell \mathbf{V}_\ell + \mathbf{1}\mathbf{d}_\ell^\top \rrbracket_H = \llbracket \mathbf{A}_\ell \rrbracket_H \llbracket \mathbf{V}_\ell \rrbracket_H, \tag{79}$$

where

$$\llbracket \mathbf{V}_\ell \rrbracket_H \triangleq \begin{pmatrix} 1 & \mathbf{d}_\ell^\top \otimes \mathbf{1}^\top \\ & \mathbf{V}_\ell \otimes \mathbf{I} \end{pmatrix}, \tag{80}$$

with inverse

$$\llbracket \mathbf{V}_\ell \rrbracket_H^{-1} = \begin{pmatrix} 1 & -\mathbf{d}_\ell^\top \mathbf{V}_\ell^{-1} \otimes \mathbf{1}^\top \\ & \mathbf{V}_\ell^{-1} \otimes \mathbf{I} \end{pmatrix}. \tag{81}$$

Note that $\llbracket \mathbf{V}_\ell \rrbracket_H$ is simply a suggestive notation, rather than an application of the expansion operator $\llbracket \cdot \rrbracket$.

Lemma 2. *Let $\mathcal{N}$, $\boldsymbol{\theta}$, $\{\phi_\ell\}_{\ell=0}^L$, and $\{\phi_\ell^\dagger\}_{\ell=0}^L$ be given as in Theorem 3. The network $\mathcal{N}^\dagger$ with activations functions $\{\phi_\ell^\dagger\}_{\ell=0}^L$ and parameters defined by*

$$[\bar{\mathbf{W}}_\ell^\dagger]_H \triangleq [\mathbf{U}_\ell]_H^{-\top} [\bar{\mathbf{W}}_\ell]_H \llbracket \mathbf{V}_{\ell-1} \rrbracket_H^{-\top}, \tag{82}$$

compute the same function as $\mathcal{N}$.

Remark. The definition of $\phi_\ell^\dagger$ (Eqn. 24) can be written in homogeneous coordinates as

$$[\mathbf{A}_\ell^\dagger]_H = \phi_\ell^\dagger([\mathbf{S}_\ell^\dagger]_H) = \phi_\ell([\mathbf{S}_\ell^\dagger]_H [\mathbf{U}_\ell]_H)[\mathbf{V}_\ell]_H. \tag{83}$$

Eqn. 82 can be expressed equivalently without homogeneous coordinates as

$$\bar{\mathbf{W}}_\ell^\dagger \triangleq \mathbf{U}_\ell^{-\top} \left(\bar{\mathbf{W}}_\ell - \mathbf{c}_\ell \mathbf{e}^\top \right) \llbracket \mathbf{V}_{\ell-1} \rrbracket_H^{-\top}, \tag{84}$$

where $\mathbf{e} = (1\ 0\ \cdots\ 0)^\top$.

Proof. We will show inductively the following relationship between the activations in each layer of the two networks:

$$[\mathbf{A}_\ell^\dagger]_H = [\mathbf{A}_\ell]_H [\mathbf{V}_\ell]_H. \tag{85}$$

(By our assumption that the top layer inputs are not transformed, i.e. $[\mathbf{V}_L]_H = \mathbf{I}$, this would imply that $[\mathbf{A}_L^\dagger]_H = [\mathbf{A}_L]_H$, and hence that the networks compute the same function.) For the first layer, Eqn. 85 is true by definition. For the inductive step, assume Eqn. 85 holds for layer $\ell - 1$. From Eqn 79, this is equivalent to

$$[\![\mathbf{A}_{\ell-1}^\dagger]\!]_H = [\![\mathbf{A}_{\ell-1}]\!]_H [\![\mathbf{V}_{\ell-1}]\!]_H. \tag{86}$$

We then derive the activations in the following layer:

$$\begin{aligned}
[\mathbf{A}_\ell^\dagger]_H &= \phi_\ell^\dagger\left([\mathbf{S}_\ell^\dagger]_H\right) && (87)\\
&= \phi_\ell\left([\mathbf{S}_\ell^\dagger]_H\,[\mathbf{U}_\ell]_H\right)\,[\mathbf{V}_\ell]_H && (88)\\
&= \phi_\ell\left([\![\mathbf{A}_{\ell-1}^\dagger]\!]_H\,[\bar{\mathbf{W}}_\ell^\dagger]_H^\top\,[\mathbf{U}_\ell]_H\right)\,[\mathbf{V}_\ell]_H && (89)\\
&= \phi_\ell\left([\![\mathbf{A}_{\ell-1}]\!]_H\,[\![\mathbf{V}_{\ell-1}]\!]_H\,[\bar{\mathbf{W}}_\ell^\dagger]_H^\top\,[\mathbf{U}_\ell]_H\right)\,[\mathbf{V}_\ell]_H && (90)\\
&= \phi_\ell\left([\![\mathbf{A}_{\ell-1}]\!]_H\,[\![\mathbf{V}_{\ell-1}]\!]_H\,[\![\mathbf{V}_{\ell-1}]\!]_H^{-1}\,[\bar{\mathbf{W}}_\ell]_H^\top\,[\mathbf{U}_\ell]_H^{-1}\,[\mathbf{U}_\ell]_H\right)\,[\mathbf{V}_\ell]_H && (91)\\
&= \phi_\ell\left([\![\mathbf{A}_{\ell-1}]\!]_H\,[\bar{\mathbf{W}}_\ell]_H^\top\right)\,[\mathbf{V}_\ell]_H && (92)\\
&= [\mathbf{A}_\ell]_H\,[\mathbf{V}_\ell]_H && (93)
\end{aligned}$$

Lines 89 and 93 are from Eqn. 73. This proves the inductive hypothesis for layer ℓ, so we have shown that both networks compute the same function. □

Lemma 3. *Suppose the parameters are transformed according to Lemma 2, and the parameters are updated according to*

$$[\bar{\mathbf{W}}_\ell^\dagger]^{(k+1)} \leftarrow [\bar{\mathbf{W}}_\ell^\dagger]^{(k)} - \alpha \mathbf{P}_\ell^\dagger (\nabla_{\bar{\mathbf{W}}_\ell^\dagger} h) \mathbf{R}_\ell^\dagger, \tag{94}$$

for matrices $\mathbf{P}_\ell$ *and* $\mathbf{R}_\ell$*. This is equivalent to applying the following update to the original network:*

$$[\bar{\mathbf{W}}_\ell]^{(k+1)} \leftarrow [\bar{\mathbf{W}}_\ell]^{(k+1)} - \alpha \mathbf{P}_\ell (\nabla_{\bar{\mathbf{W}}_\ell} h) \mathbf{R}_\ell, \tag{95}$$

with

$$\begin{aligned}
\mathbf{P}_\ell &= \mathbf{U}_\ell^\top \mathbf{P}_\ell^\dagger \mathbf{U}_\ell && (96)\\
\mathbf{R}_\ell &= [\![\mathbf{V}_{\ell-1}]\!]_H \mathbf{R}_\ell^\dagger [\![\mathbf{V}_{\ell-1}]\!]_H^\top. && (97)
\end{aligned}$$

Proof. This is a special case of Lemma 5 from Martens & Grosse (2015). □

Theorem 3. *Let* $\mathcal{N}$ *be a network with parameter vector* $\boldsymbol{\theta}$ *and activation functions* $\{\phi_\ell\}_{\ell=0}^L$*. Given activation functions* $\{\phi_\ell^\dagger\}_{\ell=0}^L$ *defined as in Eqn. 24, there exists a parameter vector* $\boldsymbol{\theta}^\dagger$ *such that a network* $\mathcal{N}^\dagger$ *with parameters* $\boldsymbol{\theta}^\dagger$ *and activation functions* $\{\phi_\ell^\dagger\}_{\ell=0}^L$ *computes the same function as* $\mathcal{N}$*. The KFC updates on* $\mathcal{N}$ *and* $\mathcal{N}^\dagger$ *are equivalent, in that the resulting networks compute the same function.*

Proof. Lemma 2 gives the desired $\boldsymbol{\theta}^\dagger$. We now prove equivalence of the KFC updates. The Kronecker factors for $\mathcal{N}^\dagger$ are given by:

$$
\begin{aligned}
\mathbf{\Omega}_\ell^\dagger &= \mathbb{E}\left[[\![\mathbf{A}_\ell^\dagger]\!]_H^\top [\![\mathbf{A}_\ell^\dagger]\!]_H \right] && (98)\\
&= \mathbb{E}\left[[\![\mathbf{V}_\ell]\!]_H^\top [\![\mathbf{A}_\ell]\!]_H^\top [\![\mathbf{A}_\ell]\!]_H [\![\mathbf{V}_\ell]\!]_H \right] && (99)\\
&= [\![\mathbf{V}_\ell]\!]_H^\top \mathbb{E}\left[[\![\mathbf{A}_\ell]\!]_H^\top [\![\mathbf{A}_\ell]\!]_H \right] [\![\mathbf{V}_\ell]\!]_H && (100)\\
&= [\![\mathbf{V}_\ell]\!]_H^\top \mathbf{\Omega}_\ell [\![\mathbf{V}_\ell]\!]_H && (101)\\
\mathbf{\Gamma}_\ell^\dagger &= \frac{1}{|\mathcal{T}|}\mathbb{E}\left[(\mathcal{D}\mathbf{S}_\ell^\dagger)^\top \mathcal{D}\mathbf{S}_\ell^\dagger \right] && (102)\\
&= \frac{1}{|\mathcal{T}|}\mathbb{E}\left[\mathbf{U}_\ell (\mathcal{D}\mathbf{S}_\ell^\dagger)^\top \mathcal{D}\mathbf{S}_\ell^\dagger \mathbf{U}_\ell^\top \right] && (103)\\
&= \frac{1}{|\mathcal{T}|}\mathbf{U}_\ell \mathbb{E}\left[(\mathcal{D}\mathbf{S}_\ell^\dagger)^\top \mathcal{D}\mathbf{S}_\ell^\dagger \right] \mathbf{U}_\ell^\top && (104)\\
&= \mathbf{U}_\ell \mathbf{\Gamma}_\ell \mathbf{U}_\ell^\top && (105)
\end{aligned}
$$

The approximate natural gradient update, ignoring momentum, clipping, and damping, is given by $\boldsymbol{\theta}^{(k+1)} \leftarrow \boldsymbol{\theta}^{(k)} - \alpha \hat{\mathbf{F}}^{-1} \nabla_{\boldsymbol{\theta}} h$. For each layer of $\mathcal{N}^\dagger$,

$$
[\bar{\mathbf{W}}_\ell^\dagger]^{(k+1)} \leftarrow [\bar{\mathbf{W}}_\ell^\dagger]^{(k)} - \alpha (\mathbf{\Gamma}_\ell^\dagger)^{-1} (\nabla_{\bar{\mathbf{W}}_\ell^\dagger} h)(\mathbf{\Omega}_{\ell-1}^\dagger)^{-1} \tag{106}
$$

We apply Lemma 3 with $\mathbf{P}_\ell^\dagger = (\mathbf{\Gamma}_\ell^\dagger)^{-1}$ and $\mathbf{R}_\ell^\dagger = (\mathbf{\Omega}_{\ell-1}^\dagger)^{-1}$. This gives us

$$
\begin{aligned}
\mathbf{P}_\ell &= \mathbf{U}_\ell^\top (\mathbf{\Gamma}_\ell^\dagger)^{-1} \mathbf{U}_\ell && (107)\\
&= \mathbf{\Gamma}_\ell^{-1} && (108)\\
\mathbf{R}_\ell &= [\![\mathbf{V}_{\ell-1}]\!]_H (\mathbf{\Omega}_{\ell-1}^\dagger)^{-1} [\![\mathbf{V}_{\ell-1}]\!]_H^\top && (109)\\
&= \mathbf{\Omega}_{\ell-1}^{-1}, && (110)
\end{aligned}
$$

with the corresponding update

$$
[\bar{\mathbf{W}}_\ell]^{(k+1)} \leftarrow [\bar{\mathbf{W}}_\ell]^{(k)} - \alpha \mathbf{\Gamma}_\ell^{-1} (\nabla_{\bar{\mathbf{W}}_\ell} h) \mathbf{\Omega}_{\ell-1}^{-1}. \tag{111}
$$

But this is the same as the KFC update for the original network. Therefore, the two updates are equivalent, in that the resulting networks compute the same function. □

Theorem 4. *Combining approximations* **IAD**, **SH**, **SUA**, *and* **WD** *results in the following approximation to the entries of the Fisher matrix:*

$$
\begin{aligned}
\mathbb{E}\left[\mathcal{D}w_{i,j,\delta} \mathcal{D}w_{i',j',\delta'}\right] &= \beta(\delta,\delta')\, \tilde{\Omega}(j,j',\delta'-\delta)\, \mathbb{1}_{i=i'} && (112)\\
\mathbb{E}\left[\mathcal{D}w_{i,j,\delta} \mathcal{D}b_{i'}\right] &= \beta(\delta)\, M(j)\, \mathbb{1}_{i=i'} && (113)\\
\mathbb{E}\left[\mathcal{D}b_i \mathcal{D}b_{i'}\right] &= |\mathcal{T}|\, \mathbb{1}_{i=i'} && (114)
\end{aligned}
$$

where $\mathbb{1}$ is the indicator function and $\tilde{\Omega}(j,j',\delta) = \Sigma(j,j')\mathbb{1}_{\delta=0} + M(j)M(j')$ is the uncentered autocovariance function. (β is defined in Theorem 1.) If the β and $|\mathcal{T}|$ terms are dropped, the resulting approximate natural gradient descent update rule is equivalent to idealized PRONG, up to rescaling.

Proof. We first compute the second moments of the activations and derivatives, under assumptions **SH**, **SUA**, and **WD**:

$$
\begin{aligned}
\mathbb{E}\left[a_{j,t} a_{j',t'}\right] &= \mathrm{Cov}(a_{j,t}, a_{j',t'}) + \mathbb{E}[a_{j,t}]\mathbb{E}[a_{j',t'}] && (115)\\
&= \Sigma(j,j')\mathbb{1}_{\delta=0} + M(j)M(j') && (116)\\
&\triangleq \tilde{\Omega}(j,j',\delta) && (117)\\
\mathbb{E}\left[\mathcal{D}s_{i,t} \mathcal{D}s_{i',t'}\right] &= \mathbb{1}_{i=i'}\mathbb{1}_{\delta=\delta'}. && (118)
\end{aligned}
$$

for any $t, t' \in \mathcal{T}$. We now compute

$$\begin{aligned}
\mathbb{E}\left[\mathcal{D}w_{i,j,\delta}\mathcal{D}w_{i,j,\delta}\right] &= \sum_{t\in\mathcal{T}}\sum_{t'\in\mathcal{T}} \mathbb{E}\left[a_{j,t+\delta}a_{j',t'+\delta'}\right]\mathbb{E}\left[\mathcal{D}s_{i,t}\mathcal{D}s_{i',t'}\right] && (119)\\
&= \sum_{t\in\mathcal{T}}\sum_{t'\in\mathcal{T}} \tilde{\Omega}(j,j',t'+\delta'-t-\delta)\,\mathbb{1}_{\substack{t+\delta\in\mathcal{T}\\ t'+\delta'\in\mathcal{T}}}\mathbb{1}_{i=i'}\mathbb{1}_{t=t'} && (120)\\
&= \sum_{t\in\mathcal{T}} \tilde{\Omega}(j,j',\delta'-\delta)\,\mathbb{1}_{\substack{t+\delta\in\mathcal{T}\\ t+\delta'\in\mathcal{T}}}\mathbb{1}_{i=i'} && (121)\\
&= \left|\{t\in\mathcal{T} : t+\delta\in\mathcal{T}, t+\delta'\in\mathcal{T}\}\right|\,\tilde{\Omega}(j,j',\delta'-\delta)\,\mathbb{1}_{i=i'} && (122)\\
&= \beta(\delta,\delta')\,\tilde{\Omega}(j,j',\delta'-\delta)\,\mathbb{1}_{i=i'} && (123)
\end{aligned}$$

Line 119 is from Lemma 1. The other formulas are derived analogously.

This can be written in matrix form as

$$\begin{aligned}
\hat{\mathbf{F}} &= \tilde{\mathbf{\Omega}} \otimes \mathbf{I} && (124)\\
\tilde{\mathbf{\Omega}} &\triangleq \begin{pmatrix} 1 & \boldsymbol{\mu}^\top \otimes \mathbf{1}^\top \\ \boldsymbol{\mu}\otimes\mathbf{1} & \mathbf{\Sigma}\otimes\mathbf{I} + \boldsymbol{\mu}\boldsymbol{\mu}^\top \otimes \mathbf{1}\mathbf{1}^\top \end{pmatrix} && (125)
\end{aligned}$$

It is convenient to compute block Cholesky decompositions:

$$\begin{aligned}
\tilde{\mathbf{\Omega}} &= \begin{pmatrix} 1 & \\ \boldsymbol{\mu}\otimes\mathbf{1} & \mathbf{B}\otimes\mathbf{I} \end{pmatrix}\begin{pmatrix} 1 & \boldsymbol{\mu}^\top\otimes\mathbf{1}^\top \\ & \mathbf{B}^\top\otimes\mathbf{I} \end{pmatrix} && (126)\\
&\triangleq \mathbf{L}\mathbf{L}^\top && (127)\\
\tilde{\mathbf{\Omega}}^{-1} &= \mathbf{L}^{-\top}\mathbf{L}^{-1} && (128)\\
&= \begin{pmatrix} 1 & -\boldsymbol{\mu}^\top\mathbf{B}^{-\top}\otimes\mathbf{1}^\top \\ & \mathbf{B}^{-\top}\otimes\mathbf{I} \end{pmatrix}\begin{pmatrix} 1 & \\ -\mathbf{B}^{-1}\boldsymbol{\mu}\otimes\mathbf{1} & \mathbf{B}^{-1}\otimes\mathbf{I} \end{pmatrix}, && (129)
\end{aligned}$$

where $\mathbf{B}$ is some square root matrix, i.e. $\mathbf{B}\mathbf{B}^\top = \mathbf{\Sigma}$ (not necessarily lower triangular).

Now consider PRONG. In the original algorithm, the network is periodically reparameterized such that the activations are white. In our idealized version of the algorithm, we assume this is done after every update. For convenience, we assume that the network is converted to the white parameterizaton immediately before computing the SGD update, and then converted back to its original parameterization immediately afterward. In other words, we apply an affine transformation (Eqn. 24) which whitens the activations:

$$\begin{aligned}
\mathbf{A}_\ell^\dagger = \phi_\ell^\dagger(\mathbf{S}_\ell^\dagger) &= \left(\phi_\ell(\mathbf{S}_\ell^\dagger) - \mathbf{1}\boldsymbol{\mu}^\top\right)\mathbf{B}^{-1} && (130)\\
&= \phi_\ell(\mathbf{S}_\ell^\dagger)\mathbf{B}^{-1} - \mathbf{1}\boldsymbol{\mu}^\top\mathbf{B}^{-1}, && (131)
\end{aligned}$$

where $\mathbf{B}$ is a square root matrix of $\mathbf{\Sigma}$, as defined above. This is an instance of Eqn. 24 with $\mathbf{U}_\ell = \mathbf{I}$, $\mathbf{c}_\ell = \mathbf{0}$, $\mathbf{V}_\ell = \mathbf{B}^{-1}$, and $\mathbf{d}_\ell = -\mathbf{B}^{-1}\boldsymbol{\mu}$. The transformed weights which compute the same function as the original network according to Lemma 2 are $\bar{\mathbf{W}}_\ell^\dagger = \bar{\mathbf{W}}_\ell\,[\![\mathbf{B}^{-1}]\!]_H^{-\top}$, where

$$[\![\mathbf{B}^{-1}]\!]_H \triangleq \begin{pmatrix} 1 & -\boldsymbol{\mu}^\top\mathbf{B}^{-\top}\otimes\mathbf{1}^\top \\ & \mathbf{B}^{-1}\otimes\mathbf{I} \end{pmatrix}, \qquad (132)$$

is defined according to Eqn. 80. But observe that $[\![\mathbf{B}^{-1}]\!]_H = \mathbf{L}^{-\top}$, where $\mathbf{L}$ is the Cholesky factor of $\tilde{\mathbf{\Omega}}$ (Eqn. 129). Therefore, we have

$$\bar{\mathbf{W}}_\ell^\dagger = \bar{\mathbf{W}}_\ell\,\mathbf{L}. \qquad (133)$$

We apply Lemma 3 with $\mathbf{P}_\ell^\dagger = \mathbf{I}$ and $\mathbf{R}_\ell^\dagger = \mathbf{I}$. This gives us the update in the original coordinate system:

$$\begin{aligned}
\bar{\mathbf{W}}_\ell^{(k+1)} &\leftarrow \bar{\mathbf{W}}_\ell^{(k)} - \alpha(\nabla_{\bar{\mathbf{W}}_\ell} h)\,\mathbf{L}^{-\top}\mathbf{L}^{-1} && (134)\\
&= \bar{\mathbf{W}}_\ell^{(k)} - \alpha(\nabla_{\bar{\mathbf{W}}_\ell} h)\,\tilde{\mathbf{\Omega}}^{-1}. && (135)
\end{aligned}$$

This is equivalent to the approximate natural gradient update where the Fisher block is approximated as $\tilde{\mathbf{\Omega}} \otimes \mathbf{I}$. This is the same approximate Fisher block we derived given the assumptions of the theorem (Eqn. 124). □