Appendices for the ICML paper "Optimizing Neural Networks with Kronecker-factored Approximate Curvature"

A Backpropagation Algorithm

Algorithm 1 An algorithm for computing the gradient of the loss $L(y, f(x, \theta))$ for a given (x, y). Note that we are assuming here for simplicity that the ϕ_i are defined as coordinate-wise functions.

input: $a_0 = x$; θ mapped to $(W_1, W_2, \ldots, W_\ell)$.

/* Forward pass */
for all i **from** 1 **to** ℓ **do**
  $s_i \leftarrow W_i \bar{a}_{i-1}$
  $a_i \leftarrow \phi_i(s_i)$
end for

/* Loss derivative computation */
$$\mathcal{D}a_\ell \leftarrow \left.\frac{\partial L(y, z)}{\partial z}\right|_{z=a_\ell}$$

/* Backwards pass */
for all i **from** ℓ **downto** 1 **do**
  $g_i \leftarrow \mathcal{D}a_i \odot \phi_i'(s_i)$
  $\mathcal{D}W_i \leftarrow g_i \bar{a}_{i-1}^\top$
  $\mathcal{D}a_{i-1} \leftarrow W_i^\top g_i$
end for

output: $\mathcal{D}\theta = [\mathrm{vec}(\mathcal{D}W_1)^\top \mathrm{vec}(\mathcal{D}W_2)^\top \ldots \mathrm{vec}(\mathcal{D}W_\ell)^\top]^\top$

B Derivation of the expression for the approximation from Section 2.1

In this section we will show that

$$\begin{aligned}\mathrm{E}\left[\bar{a}^{(1)}\bar{a}^{(2)}\, g^{(1)}g^{(2)}\right] - \mathrm{E}\left[\bar{a}^{(1)}\bar{a}^{(2)}\right]\mathrm{E}\left[g^{(1)}g^{(2)}\right] \\ = \kappa(\bar{a}^{(1)}, \bar{a}^{(2)}, g^{(1)}, g^{(2)}) + \kappa(\bar{a}^{(1)})\kappa(\bar{a}^{(2)}, g^{(1)}, g^{(2)}) + \kappa(\bar{a}^{(2)})\kappa(\bar{a}^{(1)}, g^{(1)}, g^{(2)})\end{aligned}$$

The only specific property of the distribution over $\bar{a}^{(1)}$, $\bar{a}^{(2)}$, $g^{(1)}$, and $g^{(2)}$ which we will require to do this is captured by the following lemma.

Lemma 4. *Suppose u is a scalar variable which is independent of y when conditioned on the network's output $f(x, \theta)$, and v is some intermediate quantity computed during the evaluation of $f(x, \theta)$ (such as the activities of the units in some layer). Then we have*

$$\mathrm{E}\left[u\, \mathcal{D}v\right] = 0$$

Our proof of this lemma (which is at the end of this section) makes use of the fact that the expectations are taken with respect to the network's predictive distribution $P_{y|x}$ as opposed to the training distribution $\hat{Q}_{y|x}$.

Intuitively, this lemma says that the intermediate quantities computed in the forward pass of Algorithm 1 (or various functions of these) are statistically uncorrelated with various derivative quantities computed in the backwards pass, provided that the targets y are sampled according to the network's predictive distribution $P_{y|x}$ (instead of coming from the training set). Valid choices for u include $\bar{a}^{(k)}$, $\bar{a}^{(k)} - \mathrm{E}\left[\bar{a}^{(k)}\right]$ for $k \in \{1, 2\}$, and products of these. Examples of invalid choices for u include expressions involving $g^{(k)}$, since these will depend on the derivative of the loss, which is not independent of y given $f(x, \theta)$.

According to a well-known general formula relating moments to cumulants we may write $\mathrm{E}\left[\bar{a}^{(1)}\bar{a}^{(2)}\, g^{(1)}g^{(2)}\right]$ as a sum of 15 terms, each of which is a product of various cumulants corresponding to one of the 15 possible ways to partition the elements of $\{\bar{a}^{(1)}, \bar{a}^{(2)}, g^{(1)}, g^{(2)}\}$ into non-overlapping sets. For example, the term corresponding to the partition $\{\{\bar{a}^{(1)}\}, \{\bar{a}^{(2)}, g^{(1)}, g^{(2)}\}\}$ is $\kappa(\bar{a}^{(1)})\kappa(\bar{a}^{(2)}, g^{(1)}, g^{(2)})$.

Observing that 1st-order cumulants correspond to means and 2nd-order cumulants correspond to covariances, for $k \in \{1, 2\}$ Lemma 4 gives

$$\kappa(g^{(k)}) = \mathrm{E}\left[g^{(k)}\right] = \mathrm{E}\left[\mathcal{D}x^{(k)}\right] = 0$$

where $x^{(1)} = [x_i]_{k_2}$, and $x^{(2)} = [x_j]_{k_4}$ (so that $g^{(k)} = \mathcal{D}x^{(k)}$). And similarly for $k, m \in \{1, 2\}$ it gives

$$\kappa(\bar{a}^{(k)}, g^{(m)}) = \mathrm{E}\left[\left(\bar{a}^{(m)} - \mathrm{E}\left[\bar{a}^{(m)}\right]\right)\left(g^{(k)} - \mathrm{E}\left[g^{(k)}\right]\right)\right] = \mathrm{E}\left[\left(\bar{a}^{(m)} - \mathrm{E}\left[\bar{a}^{(m)}\right]\right) g^{(k)}\right] = 0$$

Using these identities we can eliminate 10 of the terms.

The remaining expression for $\mathrm{E}\left[\bar{a}^{(1)}\bar{a}^{(2)}\, g^{(1)}g^{(2)}\right]$ is thus

$$\kappa(\bar{a}^{(1)},\bar{a}^{(2)},g^{(1)},g^{(2)}) + \kappa(\bar{a}^{(1)})\kappa(\bar{a}^{(2)},g^{(1)},g^{(2)}) + \kappa(\bar{a}^{(2)})\kappa(\bar{a}^{(1)},g^{(1)},g^{(2)}) \\ + \kappa(\bar{a}^{(1)},\bar{a}^{(2)})\kappa(g^{(1)},g^{(2)}) + \kappa(\bar{a}^{(1)})\kappa(\bar{a}^{(2)})\kappa(g^{(1)},g^{(2)})$$

Noting that

$$\kappa(\bar{a}^{(1)},\bar{a}^{(2)})\kappa(g^{(1)},g^{(2)}) + \kappa(\bar{a}^{(1)})\kappa(\bar{a}^{(2)})\kappa(g^{(1)},g^{(2)}) \\ = \mathrm{Cov}(\bar{a}^{(1)},\bar{a}^{(2)})\,\mathrm{E}\left[g^{(1)}g^{(2)}\right] + \mathrm{E}\left[\bar{a}^{(1)}\right]\mathrm{E}\left[\bar{a}^{(2)}\right]\mathrm{E}\left[g^{(1)}g^{(2)}\right] = \mathrm{E}\left[\bar{a}^{(1)}\bar{a}^{(2)}\right]\mathrm{E}\left[g^{(1)}g^{(2)}\right]$$

it thus follows that

$$\mathrm{E}\left[\bar{a}^{(1)}\bar{a}^{(2)}\, g^{(1)}g^{(2)}\right] - \mathrm{E}\left[\bar{a}^{(1)}\bar{a}^{(2)}\right]\mathrm{E}\left[g^{(1)}g^{(2)}\right] \\ = \kappa(\bar{a}^{(1)},\bar{a}^{(2)},g^{(1)},g^{(2)}) + \kappa(\bar{a}^{(1)})\kappa(\bar{a}^{(2)},g^{(1)},g^{(2)}) + \kappa(\bar{a}^{(2)})\kappa(\bar{a}^{(1)},g^{(1)},g^{(2)})$$

as required.

It remains to prove Lemma 4.

Proof of Lemma 4. The chain rule gives

$$\mathcal{D}v = -\frac{\mathrm{d}\log p(y|x,\theta)}{\mathrm{d}v} = -\frac{\mathrm{d}\log r(y|z)}{\mathrm{d}z}\bigg|_{z=f(x,\theta)}^{\top}\frac{\mathrm{d}f(x,\theta)}{\mathrm{d}v}$$

From which it follows that

$$\mathrm{E}\left[u\,\mathcal{D}v\right] = \mathrm{E}_{\hat{Q}_x}\left[\mathrm{E}_{P_{y|x}}\left[u\,\mathcal{D}v\right]\right] = \mathrm{E}_{\hat{Q}_x}\left[\mathrm{E}_{R_{y|f(x,\theta)}}\left[u\,\mathcal{D}v\right]\right] \\ = \mathrm{E}_{\hat{Q}_x}\left[\mathrm{E}_{R_{y|f(x,\theta)}}\left[-u\,\frac{\mathrm{d}\log r(y|z)}{\mathrm{d}z}\bigg|_{z=f(x,\theta)}^{\top}\frac{\mathrm{d}f(x,\theta)}{\mathrm{d}v}\right]\right] \\ = \mathrm{E}_{\hat{Q}_x}\left[-u\,\mathrm{E}_{R_{y|f(x,\theta)}}\left[\frac{\mathrm{d}\log r(y|z)}{\mathrm{d}z}\bigg|_{z=f(x,\theta)}\right]^{\top}\frac{\mathrm{d}f(x,\theta)}{\mathrm{d}v}\right] = \mathrm{E}_{\hat{Q}_x}\left[-u\,\vec{0}^{\top}\,\frac{\mathrm{d}f(x,\theta)}{\mathrm{d}v}\right] = 0$$

That the inner expectation above is $\vec{0}$ follows from the fact that the expected score of a distribution, when taken with respect to that distribution, is $\vec{0}$.

□

C Additional figures for Section 3

Figures 5 and 6 examine the quality of the approximations $\breve{F}$ and $\hat{F}$ of $\tilde{F}$, which are derived by approximating $\tilde{F}^{-1}$ as block-diagonal and block-tridiagonal (resp.), for an example network.

From Figure 5, which compares $\breve{F}$ and $\hat{F}$ directly to $\tilde{F}$, we can see that while $\breve{F}$ and $\hat{F}$ exactly capture the diagonal and tridiagonal blocks (resp.) of $\tilde{F}$, as they must by definition, $\hat{F}$ ends up approximating the off-tridiagonal blocks of $\tilde{F}$ very well too. This is likely owed to the fact that the approximating assumption used to derive $\hat{F}$, that $\tilde{F}^{-1}$ is block-tridiagonal, is a very reasonable one in practice (judging by Figure 2).

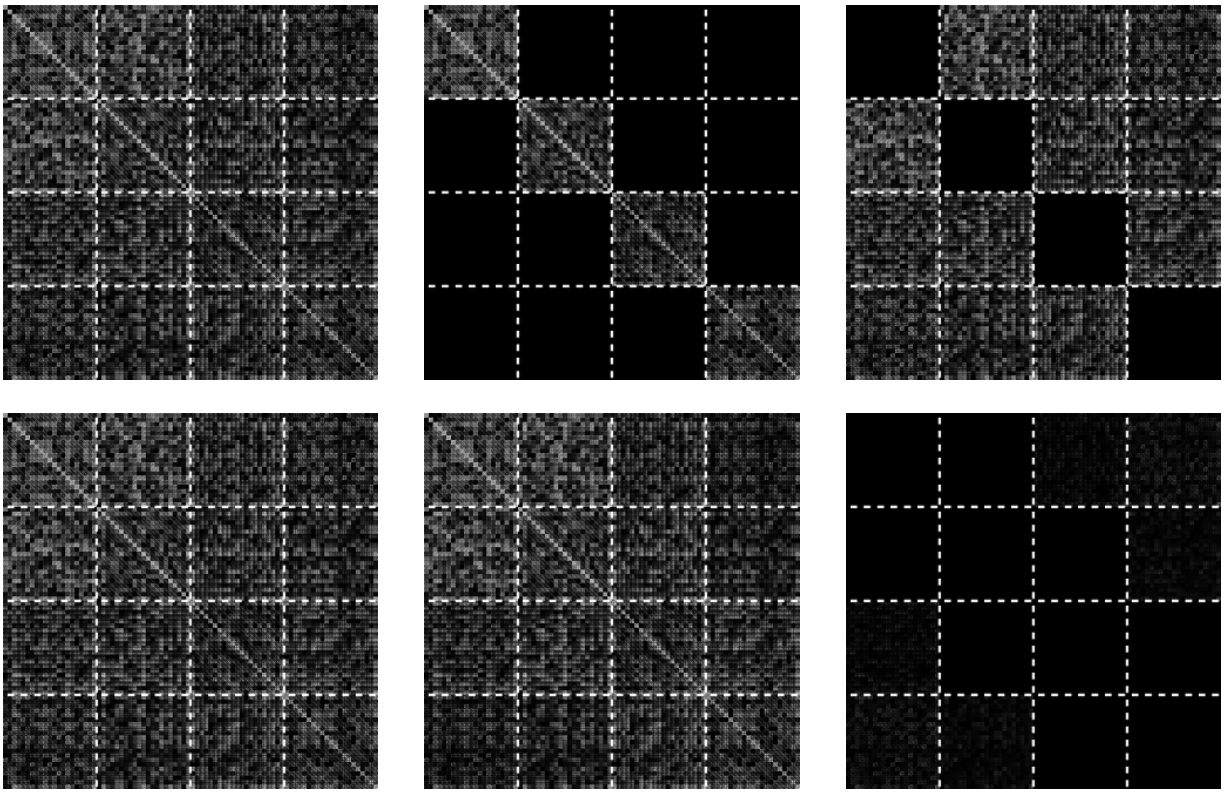

Figure 5: A comparison of our block-wise Kronecker-factored approximation $\tilde{F}$, and its approximations $\breve{F}$ and $\hat{F}$ (which are based on approximating the inverse $\tilde{F}^{-1}$ as either block-diagonal or block-tridiagonal, respectively), using the example neural network from Figure 1. On the **left** is $\tilde{F}$, in the **middle** its approximation, and on the **right** is the absolute difference of these. The **top row** compares to $\breve{F}$ and the **bottom row** compares to $\hat{F}$. While the diagonal blocks of the top right matrix, and the tridiagonal blocks of the bottom right matrix are exactly zero due to how $\breve{F}$ and $\hat{F}$ (resp.) are constructed, the off-tridiagonal blocks of the bottom right matrix, while being very close to zero, are actually non-zero (which is hard to see from the plot). Note that for the purposes of visibility we plot the absolute values of the entries, with the white level corresponding linearly to the size of these values (up to some maximum, which is the same in each image).

Figure 6, which compares $\breve{F}^{-1}$ and $\hat{F}^{-1}$ to $\tilde{F}^{-1}$, paints an arguably more interesting and relevant picture, as the quality of the approximation of the natural gradient will be roughly proportional[1] to the quality of approximation of the *inverse* Fisher. We can see from this figure that due to the approximate block-diagonal structure of $\tilde{F}^{-1}$, $\breve{F}^{-1}$ is actually a reasonably good approximation of $\tilde{F}^{-1}$, despite $\breve{F}$ being a rather poor approximation of $\tilde{F}$ (based on Figure 5). Meanwhile, we can see that by accounting for the tri-diagonal blocks, $\hat{F}^{-1}$ is indeed a significantly better approximation of $\tilde{F}^{-1}$ than $\breve{F}^{-1}$ is, even on the *diagonal* blocks.

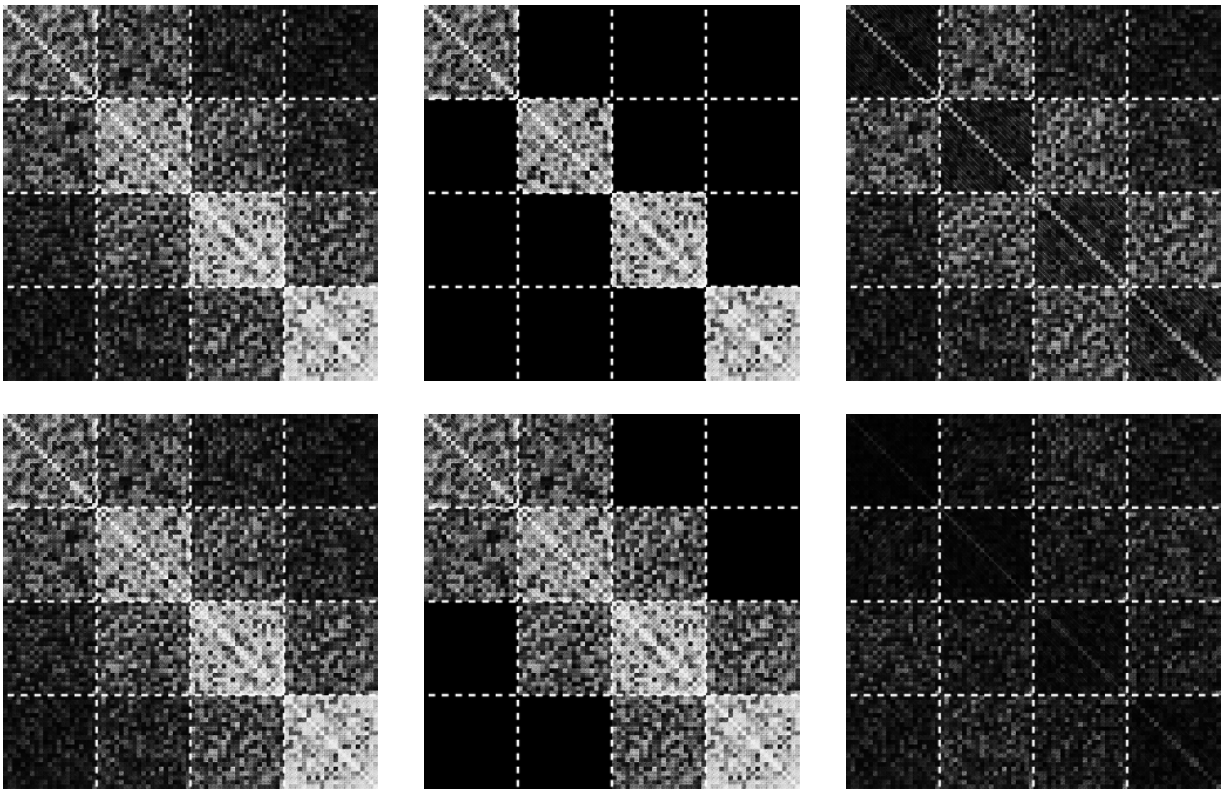

Figure 6: A comparison of the exact inverse $\tilde{F}^{-1}$ of our block-wise Kronecker-factored approximation $\tilde{F}$, and its block-diagonal and block-tridiagonal approximations $\breve{F}^{-1}$ and $\hat{F}^{-1}$ (resp.), using the example neural network from Figure 1. On the **left** is $\tilde{F}^{-1}$, in the **middle** its approximation, and on the **right** is the absolute difference of these. The **top row** compares to $\breve{F}^{-1}$ and the **bottom row** compares to $\hat{F}^{-1}$. The inverse was computed subject to the factored Tikhonov damping technique described in Appendices E.3 and E.6, using the same value of γ that was used by K-FAC at the iteration from which this example was taken (see Figure 1). Note that for the purposes of visibility we plot the absolute values of the entries, with the white level corresponding linearly to the size of these values (up to some maximum, which is the same in each image).

[1]The error in any approximation $F_0^{-1}\nabla h$ of the natural gradient $F^{-1}\nabla h$ will be roughly proportional to the error in the approximation F_0^{-1} of the associated *inverse* Fisher F^{-1}, since $\|F^{-1}\nabla h - F_0^{-1}\nabla h\| \leq \|\nabla h\|\|F^{-1} - F_0^{-1}\|$.

D Estimating the required statistics

Recall that $\bar{A}_{i,j} = \mathrm{E}\left[\bar{a}_i \bar{a}_j^\top\right]$ and $G_{i,j} = \mathrm{E}\left[g_i g_j^\top\right]$. Both approximate Fisher inverses discussed in Section 3 require some subset of these. In particular, the block-diagonal approximation requires them for $i = j$, while the block-tridiagonal approximation requires them for $j \in \{i, i+1\}$ (noting that $\bar{A}_{i,j}^\top = \bar{A}_{j,i}$ and $G_{i,j}^\top = G_{j,i}$).

Since the $\bar{a}_i$'s don't depend on y, we can take the expectation $\mathrm{E}\left[\bar{a}_i \bar{a}_j^\top\right]$ with respect to just the training distribution $\hat{Q}_x$ over the inputs x. On the other hand, the g_i's do depend on y, and so the expectation[2] $\mathrm{E}\left[g_i g_j^\top\right]$ must be taken with respect to *both* $\hat{Q}_x$ and the network's predictive distribution $P_{y|x}$.

While computing matrix-vector products with the $G_{i,j}$ could be done exactly and efficiently for a given input x (or small mini-batch of x's) by adapting the methods of Schraudolph (2002), there doesn't seem to be a sufficiently efficient method for computing the entire matrix itself. Indeed, the hardness results of Martens et al. (2012) suggest that this would require, for each example x in the mini-batch, work that is asymptotically equivalent to matrix-matrix multiplication involving matrices the same size as $G_{i,j}$. While a small constant number of such multiplications is arguably an acceptable cost (see Appendix G), a number which grows with the size of the mini-batch would not be.

Instead, we will approximate the expectation over y by a standard Monte-Carlo estimate obtained by sampling y's from the network's predictive distribution and then rerunning the backwards phase of backpropagation (see Algorithm 1) as if these were the training targets.

Note that computing/estimating the required $\bar{A}_{i,j}$/$G_{i,j}$'s involves computing averages over outer products of various $\bar{a}_i$'s from network's usual forward pass, and g_i's from the modified backwards pass (with targets sampled as above). Thus we can compute/estimate these quantities on the same input data used to compute the gradient ∇h, at the cost of one or more additional backwards passes, and a few additional outer-product averages. Fortunately, this turns out to be quite inexpensive, as we have found that just one modified backwards pass is sufficient to obtain a good quality estimate in practice, and the required outer-product averages are similar to those already used to compute the gradient in the usual backpropagation algorithm.

In the case of online/stochastic optimization we have found that the best strategy is to maintain running estimates of the required $\bar{A}_{i,j}$'s and $G_{i,j}$'s using a simple exponentially decaying averaging scheme. In particular, we take the new running estimate to be the old one weighted by ϵ, plus the

[2] It is important to note this expectation should *not* be taken with respect to the training/data distribution over y (i.e. $\hat{Q}_{y|x}$ or $Q_{y|x}$). Using the training/data distribution for y would perhaps give an approximation to a quantity known as the "empirical Fisher information matrix", which lacks the previously discussed equivalence to the Generalized Gauss-Newton matrix, and would not be compatible with the theoretical analysis performed in Section 2.1 (in particular, Lemma 4 would break down). Moreover, such a choice would not give rise to what is usually thought of as the natural gradient, and based on the findings of Martens (2010), would likely perform worse in practice as part of an optimization algorithm. See Martens (2014) for a more detailed discussion of the empirical Fisher and reasons why it may be a poor choice for a curvature matrix compared to the standard Fisher.

estimate on the new mini-batch weighted by $1 - \epsilon$, for some $0 \leq \epsilon < 1$. In our experiments we used $\epsilon = \min\{1 - 1/k, 0.95\}$, where k is the iteration number.

Note that the more naive averaging scheme where the estimates from each iteration are given equal weight would be inappropriate here. This is because the $\bar{A}_{i,j}$'s and $G_{i,j}$'s depend on the network's parameters θ, and these will slowly change over time as optimization proceeds, so that estimates computed many iterations ago will become stale.

This kind of exponentially decaying averaging scheme is commonly used in methods involving diagonal or block-diagonal approximations (with much smaller blocks than ours) to the curvature matrix (e.g. LeCun et al., 1998; Park et al., 2000; Schaul et al., 2013). Such schemes have the desirable property that they allow the curvature estimate to depend on much more data than can be reasonably processed in a single mini-batch.

Notably, for methods like HF which deal with the exact Fisher indirectly via matrix-vector products, such a scheme would be impossible to implement efficiently, as the exact Fisher matrix (or GGN) seemingly cannot be summarized using a compact data structure whose size is independent of the amount of data used to estimate it. Indeed, it seems that the only representation of the exact Fisher which would be independent of the amount of data used to estimate it would be an explicit $n \times n$ matrix (which is far too big to be practical). Because of this, HF and related methods must base their curvature estimates only on subsets of data that can be reasonably processed all at once, which limits their effectiveness in the stochastic optimization regime.

E Update damping

E.1 Background and motivation

The idealized natural gradient approach is to follow the smooth path[3] in the Riemannian manifold (implied by the Fisher information matrix viewed as a metric tensor) that is generated by taking a series of infinitesimally small steps (in the original parameter space) in the direction of the natural gradient (which gets recomputed at each point). While this is clearly impractical as a real optimization method, one can take larger steps and still follow these paths approximately. But in our experience, to obtain an update which satisfies the minimal requirement of not worsening the objective function value, it is often the case that one must make the step size so small that the resulting optimization algorithm performs poorly in practice.

The reason that the natural gradient can only be reliably followed a short distance is that it is defined merely as an optimal *direction* (which trades off improvement in the objective versus change in the predictive distribution), and not a discrete *update*.

[3]Which has the interpretation of being a geodesic in the Riemannian manifold from the current predictive distribution towards the training distribution when using a likelihood or KL-divergence based objective function (see Martens (2014)).

Fortunately, as observed by Martens (2014), the natural gradient can be understood using a more traditional optimization-theoretic perspective which implies how it can be used to generate updates that will be useful over larger distances. In particular, when $R_{y|z}$ is an exponential family model with z as its *natural* parameters (as it will be in our experiments), Martens (2014) showed that the Fisher becomes equivalent to the Generalized Gauss-Newton matrix (GGN), which is a positive semi-definite approximation of the Hessian of h. Additionally, there is the well-known fact that when $L(x, f(x, \theta))$ is the negative log-likelihood function associated with a given (x, y) pair (as we are assuming in this work), the Hessian H of h and the Fisher F are closely related in the sense H is the expected Hessian of L under the *training* distribution $\hat{Q}_{x,y}$, while F is the expected Hessian of L under the *model's* distribution $P_{x,y}$ (defined by the density $p(x, y) = p(y|x)q(x)$).

From these observations it follows that

$$M(\delta) = \frac{1}{2}\delta^\top F\delta + \nabla h(\theta)^\top \delta + h(\theta) \tag{5}$$

can be viewed as a convex approximation of the 2nd-order Taylor series of expansion of $h(\delta + \theta)$, whose minimizer δ^* is the (negative) natural gradient $-F^{-1}\nabla h(\theta)$. Note that if we add an ℓ_2 or "weight-decay" regularization term to h of the form $\frac{\eta}{2}\|\theta\|_2^2$, then similarly $F+\eta I$ can be viewed as an approximation of the Hessian of h, and replacing F with $F + \eta I$ in $M(\delta)$ yields an approximation of the 2nd-order Taylor series, whose minimizer is a kind of "regularized" (negative) natural gradient $-(F + \eta I)^{-1}\nabla h(\theta)$, which is what we end up using in practice.

From the interpretation of the natural gradient as the minimizer of $M(\delta)$, we can see that it fails to be useful as a local update only insofar as $M(\delta)$ fails to be a good local approximation to $h(\delta + \theta)$. And so as argued by Martens (2014), it is natural to make use of the various "damping" techniques that have been developed in the optimization literature for dealing with the breakdowns in local quadratic approximations that inevitably occur during optimization. Notably, this breakdown usually won't occur in the final "local convergence" stage of optimization where the function becomes well approximated as a convex quadratic within a sufficiently large neighborhood of the local optimum. This is the phase traditionally analyzed in most theoretical results, and while it is important that an optimizer be able to converge well in this final phase, it is arguably much more important from a practical standpoint that it behaves sensibly before this phase.

This initial "exploration phase" (Darken and Moody, 1990) is where damping techniques help in ways that are not apparent from the asymptotic convergence theorems alone, which is not to say there are not strong mathematical arguments that support their use (see Nocedal and Wright, 2006). In particular, in the exploration phase it will often still be true that $h(\theta + \delta)$ is accurately approximated by a convex quadratic *locally within some region* around $\delta = 0$, and that therefor optimization can be most efficiently performed by minimizing a sequence of such convex quadratic approximations within adaptively sized local regions.

Note that well designed damping techniques, such as the ones we will employ, automatically adapt to the local properties of the function, and effectively "turn themselves off" when the quadratic model becomes a sufficiently accurate local approximation of h, allowing the optimizer to achieve the desired asymptotic convergence behavior (Moré, 1978).

Successful and theoretically well-founded damping techniques include Tikhonov damping (aka Tikhonov regularization, which is closely connected to the trust-region method) with Levenberg-Marquardt style adaptation (Moré, 1978), line-searches, and trust regions, truncation, etc., all of which tend to be much more effective in practice than merely applying a learning rate to the update, or adding a *fixed* multiple of the identity to the curvature matrix. Indeed, a subset of these techniques was exploited in the work of Martens (2010), and primitive versions of them have appeared implicitly in older works such as Becker and LeCun (1989), and also in many recent diagonal methods like that of Zeiler (2013), although often without a good understanding of what they are doing and why they help.

Crucially, more powerful 2nd-order optimizers like HF and K-FAC, which have the capability of taking *much larger steps* than 1st-order methods (or methods which use diagonal curvature matrices), *require* more sophisticated damping solutions to work well, and will usually *completely fail* without them, which is consistent with predictions made in various theoretical analyses (e.g. Nocedal and Wright, 2006). As an analogy one can think of such powerful 2nd-order optimizers as extremely fast racing cars that need more sophisticated control systems than standard cars to prevent them from flying off the road. Arguably one of the reasons why high-powered 2nd-order optimization methods have historically tended to under-perform in machine learning applications, and in neural network training in particular, is that their designers did not understand or take seriously the issue of quadratic model approximation quality, and did not employ the more sophisticated and effective damping techniques that are available to deal with this issue.

For a detailed review and discussion of various damping techniques and their crucial role in practical 2nd-order optimization methods, we refer the reader to Martens and Sutskever (2012).

E.2 A highly effective damping scheme for K-FAC

Methods like HF which use the exact Fisher seem to work reasonably well with an adaptive Tikhonov regularization technique where λI is added to $F + \eta I$, and where λ is adapted according to Levenberg-Marquardt style adjustment rule. This common and well-studied method can be shown to be equivalent to imposing an adaptive spherical region (known as a "trust region") which constrains the optimization of the quadratic model (e.g Nocedal and Wright, 2006). However, we found that this simple technique is insufficient when used with our approximate natural gradient update proposals. In particular, we have found that there never seems to be a "good" choice for λ that gives rise to updates which are of a quality comparable to those produced by methods that use the exact Fisher, such as HF.

One possible explanation for this finding is that, unlike quadratic models based on the exact Fisher (or equivalently, the GGN), the one underlying K-FAC has no guarantee of being accurate up to 2nd-order. Thus, λ must remain large in order to compensate for this intrinsic 2nd-order inaccuracy of the model, which has the side effect of "washing out" the small eigenvalues (which represent important low-curvature directions).

Fortunately, through trial and error, we were able to find a relatively simple and highly effective damping scheme, which combines several different techniques, and which works well within K-FAC. Our scheme works by computing an initial update proposal using a version of the above described adaptive Tikhonov damping/regularization method, and then re-scaling this according to quadratic model computed using the exact Fisher. This second step is made practical by the fact that it only requires a single matrix-vector product with the exact Fisher, and this can be computed efficiently using standard methods. We discuss the details of this scheme in the following subsections.

E.3 A factored Tikhonov regularization technique

In the first stage of our damping scheme we generate a candidate update proposal Δ by applying a slightly modified form of Tikhononv damping to our approximate Fisher, before multiplying $-\nabla h$ by its inverse.

In the usual Tikhonov regularization/damping technique, one adds $(\lambda + \eta)I$ to the curvature matrix (where η accounts for the ℓ_2 regularization), which is equivalent to adding a term of the form $\frac{\lambda + \eta}{2}\|\delta\|_2^2$ to the corresponding quadratic model (given by $M(\delta)$ with F replaced by our approximation). For the block-diagonal approximation $\breve{F}$ of $\tilde{F}$ (from Section 3.2) this amounts to adding $(\lambda + \eta)I$ (for a lower dimensional I) to each of the individual diagonal blocks, which gives modified diagonal blocks of the form

$$\bar{A}_{i-1,i-1} \otimes G_{i,i} \ + \ (\lambda + \eta)I \ = \ \bar{A}_{i-1,i-1} \otimes G_{i,i} \ + \ (\lambda + \eta)I \otimes I \tag{6}$$

Because this is the sum of two Kronecker products we cannot use the simple identity $(A \otimes B)^{-1} = A^{-1} \otimes B^{-1}$ anymore. Fortunately however, there are efficient techniques for inverting such matrices, which we discuss in detail in Appendix I.

If we try to apply this same Tikhonov technique to our more sophisticated approximation $\hat{F}$ of $\tilde{F}$ (from Section 3.3) by adding $(\lambda + \eta)I$ to each of the diagonal blocks of $\hat{F}$, it is no longer clear how to efficiently invert $\hat{F}$. Instead, a solution which we have found works very well in practice (and which we also use for the block-diagonal approximation $\breve{F}$), is to add $\pi_i(\sqrt{\lambda + \eta})I$ and $\frac{1}{\pi_i}(\sqrt{\lambda + \eta})I$ for a scalar constant π_i to the individual Kronecker factors $\bar{A}_{i-1,i-1}$ and $G_{i,i}$ (resp.) of each diagonal block, giving

$$\left(\bar{A}_{i-1,i-1} + \pi_i(\sqrt{\lambda + \eta})I\right) \otimes \left(G_{i,i} + \frac{1}{\pi_i}(\sqrt{\lambda + \eta})I\right) \tag{7}$$

As this is a single Kronecker product, all of the computations described in Sections 3.2 and 3.3 can still be used here too, simply by replacing each $\bar{A}_{i-1,i-1}$ and $G_{i,i}$ with their modified versions $\bar{A}_{i-1,i-1} + \pi_i(\sqrt{\lambda + \eta})I$ and $G_{i,i} + \frac{1}{\pi_i}(\sqrt{\lambda + \eta})I$.

To see why the expression in eqn. 7 is a reasonable approximation to eqn. 6, note that expanding it gives

$$\bar{A}_{i-1,i-1} \otimes G_{i,i} + \pi_i(\sqrt{\lambda+\eta})I \otimes G_{i,i} + \frac{1}{\pi_i}(\sqrt{\lambda+\eta})\bar{A}_{i-1,i-1} \otimes I + (\lambda+\eta)I \otimes I$$

which differs from eqn. 6 by the residual error expression

$$\pi_i(\sqrt{\lambda+\eta})I \otimes G_{i,i} + \frac{1}{\pi_i}(\sqrt{\lambda+\eta})\bar{A}_{i-1,i-1} \otimes I$$

While the choice of $\pi_i = 1$ is simple and can sometimes work well in practice, a slightly more principled choice can be found by minimizing the obvious upper bound (following from the triangle inequality) on the matrix norm of this residual expression, for some matrix norm $\|\cdot\|_v$. This gives

$$\pi_i = \sqrt{\frac{\|\bar{A}_{i-1,i-1} \otimes I\|_v}{\|I \otimes G_{i,i}\|_v}}$$

Evaluating this expression can be done efficiently for various common choices of the matrix norm $\|\cdot\|_v$. For example, for a general B we have $\|I \otimes B\|_F = \|B \otimes I\|_F = \sqrt{d}\|B\|_F$ where d is the height/dimension of I, and also $\|I \otimes B\|_2 = \|B \otimes I\|_2 = \|B\|_2$.

In our experience, one of the best and must robust choices for the norm $\|\cdot\|_v$ is the trace-norm, which for PSD matrices is given by the trace. With this choice, the formula for π_i has the following simple form:

$$\pi_i = \sqrt{\frac{\mathrm{tr}(\bar{A}_{i-1,i-1})/(d_{i-1}+1)}{\mathrm{tr}(G_{i,i})/d_i}}$$

where d_i is the dimension (number of units) in layer i. Intuitively, the inner fraction is just the average eigenvalue of $\bar{A}_{i-1,i-1}$ divided by the average eigenvalue of $G_{i,i}$.

Interestingly, we have found that this factored approximate Tikhonov approach, which was originally motivated by computational concerns, often works better than the exact version (eqn. 6) in practice. The reasons for this are still somewhat mysterious to us, but it may have to do with the fact that the inverse of the product of two quantities is often most robustly estimated as the inverse of the product of their individually regularized estimates.

E.4 Re-scaling according to the exact F

Given an update proposal Δ produced by multiplying the negative gradient $-\nabla h$ by our approximate Fisher inverse (subject to the Tikhonov technique described in the previous subsection), the

second stage of our proposed damping scheme re-scales Δ according to the quadratic model M as computed with the exact F, to produce a final update $\delta = \alpha\Delta$.

More precisely, optimize α according to the value of the quadratic model $M(\delta) = M(\alpha\Delta)$ as computed using an estimate of the exact Fisher F (to which we add the ℓ_2 regularization + Tikhonov term $(\lambda + \eta)I$). In particular, we minimize the following function with respect to α:

$$M(\delta) = M(\alpha\Delta) = \frac{\alpha^2}{2}\Delta^\top(F + (\lambda + \eta)I)\Delta + \alpha\nabla h^\top\Delta + h(\theta)$$

Because this is a 1-dimensional quadratic minimization problem, the formula for the optimal α can be computed very efficiently as

$$\alpha^* = \frac{-\nabla h^\top\Delta}{\Delta^\top(F + (\lambda + \eta)I)\Delta} = \frac{-\nabla h^\top\Delta}{\Delta^\top F\Delta + (\lambda + \eta)\|\Delta\|_2^2}$$

To evaluate this formula we use the current stochastic gradient ∇h (i.e. the same one used to produce Δ), and compute matrix-vector products with F using the input data from the same mini-batch. While using a mini-batch to compute F gets away from the idea of basing our estimate of the curvature on a long history of data (as we do with our *approximate* Fisher), it is made slightly less objectionable by the fact that we are only using it to estimate a single scalar quantity ($\Delta^\top F\Delta$). This is to be contrasted with methods like HF which perform a long and careful optimization of $M(\delta)$ using such an estimate of F.

Because the matrix-vector products with F are only used to compute scalar quantities in K-FAC, we can reduce their computational cost by roughly one half (versus standard matrix-vector products with F) using a simple trick which is discussed in Appendix J.

Intuitively, this second stage of our damping scheme effectively compensates for the intrinsic inaccuracy of the approximate quadratic model (based on our approximate Fisher) used to generate the initial update proposal Δ, by essentially falling back on a more accurate quadratic model based on the exact Fisher.

Interestingly, by re-scaling Δ according to $M(\delta)$, K-FAC can be viewed as a version of HF which uses our approximate Fisher as a preconditioning matrix (instead of the traditional diagonal preconditioner), and runs CG for only 1 step, initializing it from 0. This observation suggests running CG for longer, thus obtaining an algorithm which is even closer to HF (although using a much better preconditioner for CG). Indeed, this approach works reasonably well in our experience, but suffers from some of the same problems that HF has in the stochastic setting, due its much stronger use of the mini-batch–estimated exact F.

Figure 7 demonstrates the effectiveness of this re-scaling technique versus the simpler method of just using the raw Δ as an update proposal. We can see that Δ, without being re-scaled, is a very poor update to θ, and won't even give *any* improvement in the objective function unless the strength of the factored Tikhonov damping terms is made very large. On the other hand, when the update is re-scaled, we can afford to compute Δ using a much smaller strength for the factored Tikhonov damping terms, and overall this yields a much larger and more effective update to θ.

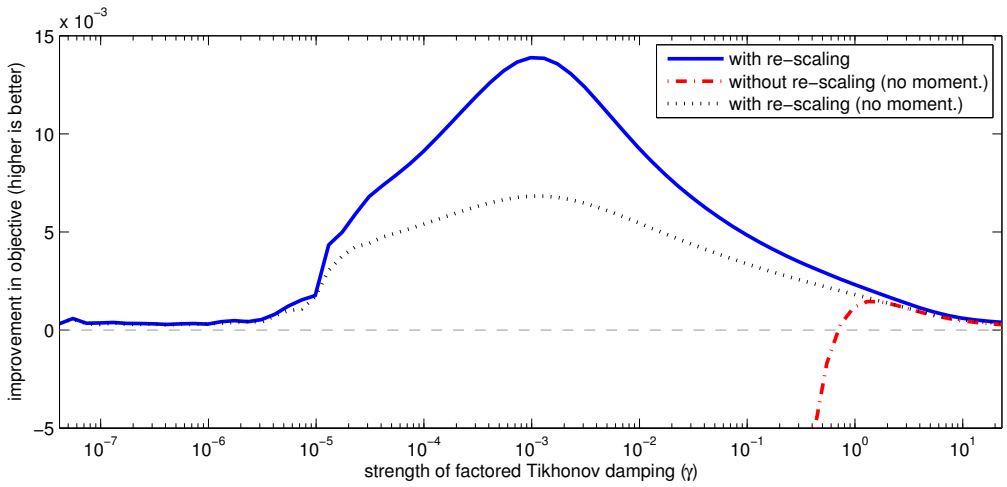

Figure 7: A comparison of the effectiveness of the proposed damping scheme, with and without the re-scaling techniques described in Appendix E.4. The network used for this comparison is the one produced at iteration 500 by K-FAC (with the block-tridiagonal inverse approximation) on the MNIST autoencoder problem described in Section 7. The y-axis is the improvement in the objective function h (i.e. $h(\theta) - h(\theta + \delta)$) produced by the update δ, while the x-axis is the strength constant used in the factored Tikhonov damping technique (which is denoted by "γ" as described in Appendix E.6). In the legend, "no moment." indicates that the momentum technique developed for K-FAC in Appendix F (which relies on the use of re-scaling) was not used.

E.5 Adapting λ

Tikhonov damping can be interpreted as implementing a trust-region constraint on the update δ, so that in particular the constraint $\|\delta\| \leq r$ is imposed for some r, where r depends on λ and the curvature matrix (e.g. Nocedal and Wright, 2006). While some approaches adjust r and then seek to find the matching λ, it is often simpler just to adjust λ directly, as the precise relationship between λ and r is complicated, and the curvature matrix is constantly evolving as optimization takes place.

The theoretically well-founded Levenberg-Marquardt style rule used by HF for doing this, which we will adopt for K-FAC, is given by

$$\text{if } \rho > 3/4 \text{ then } \lambda \leftarrow \omega_1 \lambda$$

$$\text{if } \rho < 1/4 \text{ then } \lambda \leftarrow \frac{1}{\omega_1} \lambda$$

where $\rho \equiv \dfrac{h(\theta + \delta) - h(\theta)}{M(\delta)}$ is the "reduction ratio" and $0 < \omega_1 < 1$ is some decay constant, and all quantities are computed on the current mini-batch (and M uses the exact F).

Intuitively, this rule tries to make λ as small as possible (and hence the implicit trust-region as large as possible) while maintaining the property that the quadratic model $M(\delta)$ remains a good

local approximation to h (in the sense that it accurately predicts the value of $h(\theta + \delta)$ for the δ which gets chosen at each iteration). It has the desirable property that as the optimization enters the final convergence stage where M becomes an almost exact approximation in a sufficiently large neighborhood of the local minimum, the value of λ will go rapidly enough towards 0 that it doesn't interfere with the asymptotic local convergence theory enjoyed by 2nd-order methods (Moré, 1978).

In our experiments we applied this rule every T_1 iterations of K-FAC, with $\omega_1 = (19/20)^{T_1}$ and $T_1 = 5$, from a starting value of $\lambda = 150$. Note that the optimal value of ω_1 and the starting value of λ may be application dependent, and setting them inappropriately could significantly slow down K-FAC in practice.

Computing ρ can be done quite efficiently. Note that for the optimal δ, $M(\delta) = \frac{1}{2}\nabla h^\top \delta$, and $h(\theta)$ is available from the usual forward pass. The only remaining quantity which is needed to evaluate ρ is thus $h(\theta + \delta)$, which will require an additional forward pass. But fortunately, we only need to perform this once every T_1 iterations.

E.6 Maintaining a separate damping strength for the approximate Fisher

While the scheme described in the previous sections works reasonably well in most situations, we have found that in order to avoid certain failure cases and to be truly robust in a large variety of situations, the Tikhonov damping strength parameter for the factored Tikhonov technique described in Appendix E.3 should be maintained and adjusted independently of λ. To this end we replace the expression $\sqrt{\lambda + \eta}$ in Appendix E.3 with a separate constant γ, which we initialize to $\sqrt{\lambda + \eta}$ but which is then adjusted using a different rule, which is described at the end of this section.

The reasoning behind this modification is as follows. The role of λ, according to the Levenberg Marquardt theory (Moré, 1978), is to be as small as possible while maintaining the property that the quadratic model M remains a trust-worthy approximation of the true objective. Meanwhile, γ's role is to ensure that the initial update proposal Δ is as good an approximation as possible to the true optimum of M (as computed using a mini-batch estimate of the exact F), so that in particular the re-scaling performed in Appendix E.4 is as benign as possible. While one might hope that adding the same multiple of the identity to our approximate Fisher as we do to the exact F (as it appears in M) would produce the best Δ in this regard, this isn't obviously the case. In particular, using a larger multiple may help compensate for the approximation we are making to the Fisher when computing Δ, and thus help produce a more "conservative" but ultimately more useful initial update proposal Δ, which is what we observe happens in practice.

A simple measure of the quality of our choice of γ is the (negative) value of the quadratic model $M(\delta) = M(\alpha\Delta)$ for the optimally chosen α. To adjust γ based on this measure (or others like it) we use a simple greedy adjustment rule. In particular, every T_2 iterations during the optimization we try 3 different values of γ (γ_0, $\omega_2\gamma_0$, and $(1/\omega_2)\gamma_0$, where γ_0 is the current value) and choose the new γ to be the best of these, as measured by our quality metric. In our experiments

we used $T_2 = 20$ (which must be a multiple of the constant T_3 as defined in Appendix G), and $\omega_2 = (\sqrt{19/20})^{T_2}$.

We have found that $M(\delta)$ works well in practice as a measure of the quality of γ, and has the added bonus that it can be computed at essentially no additional cost from the incidental quantities already computed when solving for the optimal α. In our initial experiments we found that using it gave similar results to those obtained by using other obvious measures for the quality of γ, such as $h(\theta + \delta)$.

F Momentum

Sutskever et al. (2013) found that momentum (Polyak, 1964; Plaut et al., 1986) was very helpful in the context of stochastic gradient descent optimization of deep neural networks. A version of momentum is also present in the original HF method, and it plays an arguably even more important role in more "stochastic" versions of HF (Martens and Sutskever, 2012; Kiros, 2013).

A natural way of adding momentum to K-FAC, and one which we have found works well in practice, is to take the update to be $\delta = \alpha\Delta + \mu\delta_0$, where δ_0 is the final update computed at the previous iteration, and where α and μ are chosen to minimize $M(\delta)$. This allows K-FAC to effectively build up a better solution to the local quadratic optimization problem $\min_\delta M(\delta)$ (where M uses the *exact* F) over many iterations, somewhat similarly to how Matrix Momentum (Scarpetta et al., 1999) and HF do this (see Sutskever et al., 2013).

The optimal solution for α and μ can be computed as

$$\begin{bmatrix} \alpha^* \\ \mu^* \end{bmatrix} = - \begin{bmatrix} \Delta^\top F \Delta + (\lambda + \eta)\|\Delta\|_2^2 & \Delta^\top F \delta_0 + (\lambda + \eta)\Delta^\top \delta_0 \\ \Delta^\top F \delta_0 + (\lambda + \eta)\Delta^\top \delta_0 & \delta_0^\top F \delta_0 + (\lambda + \eta)\|\delta_0\|_2^2 \end{bmatrix}^{-1} \begin{bmatrix} \nabla h^\top \Delta \\ \nabla h^\top \delta_0 \end{bmatrix}$$

The main cost in evaluating this formula is computing the two matrix-vector products $F\Delta$ and $F\delta_0$. Fortunately, the technique discussed in Appendix J can be applied here to compute the 4 required scalars at the cost of only two forwards passes (equivalent to the cost of only one matrix-vector product with F).

Empirically we have found that this type of momentum provides substantial acceleration in regimes where the gradient signal has a low noise to signal ratio, which is usually the case in the early to mid stages of stochastic optimization, but can also be the case in later stages if the mini-batch size is made sufficiently large. These findings are consistent with predictions made by convex optimization theory, and with older empirical work done on neural network optimization (LeCun et al., 1998).

Notably, because the implicit "momentum decay constant" μ in our method is being computed on the fly, one doesn't have to worry about setting schedules for it, or adjusting it via heuristics, as one often does in the context of SGD.

Interestingly, if h is a quadratic function (so the definition of $M(\delta)$ remains fixed at each iteration) and all quantities are computed deterministically (i.e. without noise), then using this type of momentum makes K-FAC equivalent to performing preconditioned linear CG on $M(\delta)$, with the preconditioner given by our approximate Fisher. This follows from the fact that linear CG can be interpreted as a momentum method where the learning rate α and momentum decay coefficient μ are chosen to jointly minimize $M(\delta)$ at the current iteration.

G Computational Costs and Efficiency Improvements

Let d be the typical number of units in each layer and m the mini-batch size. The significant computational tasks required to compute a single update/iteration of K-FAC, and rough estimates of their associated computational costs, are as follows:

1. standard forwards and backwards pass: $2C_1\ell d^2 m$
2. computation of the gradient ∇h on the current mini-batch using quantities computed in backwards pass: $C_2\ell d^2 m$
3. additional backwards pass with random targets (as described in Appendix D): $C_1\ell d^2 m$
4. updating the estimates of the required $\bar{A}_{i,j}$'s and $G_{i,j}$'s from quantities computed in the forwards pass and the additional randomized backwards pass: $2C_2\ell d^2 m$
5. matrix inverses (or SVDs for the block-tridiagonal inverse, as described in Appendix I) required to compute the inverse of the approximate Fisher: $C_3\ell d^3$ for the block-diagonal inverse, $C_4\ell d^3$ for the block-tridiagonal inverse
6. various matrix-matrix products required to compute the matrix-vector product of the approximate inverse with the stochastic gradient: $C_5\ell d^3$ for the block-diagonal inverse, $C_6\ell d^3$ for the block-tridiagonal inverse
7. matrix-vector products with the exact F on the current mini-batch using the approach in Appendix J: $4C_1\ell d^2 m$ with momentum, $2C_1\ell d^2 m$ without momentum
8. additional forward pass required to evaluate the reduction ratio ρ needed to apply the λ adjustment rule described in Appendix E.5: $C_1\ell d^2 m$ every T_1 iterations

Here the C_i are various constants that account for implementation details, and we are assuming the use of the naive cubic matrix-matrix multiplication and inversion algorithms when producing the cost estimates. Note that it it is hard to assign precise values to the constants, as they very much depend on how these various tasks are implemented.

Note that most of the computations required for these tasks will be sped up greatly by performing them in parallel across units, layers, training cases, or all of these. The above cost estimates however measure sequential operations, and thus may not accurately reflect the true computation

times enjoyed by a parallel implementation. In our experiments we used a vectorized implementation that performed the computations in parallel over units and training cases, although not over layers (which is possible for computations that don't involve a sequential forwards or backwards "pass" over the layers).

Tasks 1 and 2 represent the standard stochastic gradient computation.

The costs of tasks 3 and 4 are similar and slightly smaller than those of tasks 1 and 2. One way to significantly reduce them is to use a random subset of the current mini-batch of size $\tau_1 m$ to update the estimates of the required $\bar{A}_{i,j}$'s and $G_{i,j}$'s. One can similarly reduce the cost of task 7 by computing the (factored) matrix-vector product with F using such a subset of size $\tau_2 m$, although we recommend caution when doing this, as using inconsistent sets of data for the quadratic and linear terms in $M(\delta)$ can hypothetically cause instability problems which are avoided by using consistent data (see Martens and Sutskever (2012), Section 13.1). In our experiments in Section 7 we used $\tau_1 = 1/8$ and $\tau_2 = 1/4$, which seemed to have a negligible effect on the quality of the resultant updates, while significantly reducing per-iteration computation time. In a separate set of unreported experiments we found that in certain situations, such as when ℓ_2 regularization isn't used and the network starts heavily overfitting the data, or when smaller mini-batches were used, we had to revert to using $\tau_2 = 1$ to prevent significant deterioration in the quality of the updates.

The cost of task 8 can be made relatively insignificant by making the adjustment period T_1 for λ large enough. We used $T_1 = 5$ in our experiments.

The costs of tasks 5 and 6 are hard to compare directly with the costs associated with computing the gradient, as their relative sizes will depend on factors such as the architecture of the neural network being trained, as well as the particulars of the implementation. However, one quick observation we can make is that both tasks 5 and 6 involve computations that be performed in parallel across the different layers, which is to be contrasted with many of the other tasks which require *sequential* passes over the layers of the network.

Clearly, if $m \gg d$, then the cost of tasks 5 and 6 becomes negligible in comparison to the others. However, it is more often the case that m is comparable or perhaps smaller than d. Moreover, while algorithms for inverses and SVDs tend to have the same asymptotic cost as matrix-matrix multiplication, they are at least several times more expensive in practice, in addition to being harder to parallelize on modern GPU architectures (indeed, CPU implementations are often faster in our experience). Thus, C_3 and C_4 will typically be (much) larger than C_5 and C_6, and so in a basic/naive implementation of K-FAC, task 5 can dominate the overall per-iteration cost.

Fortunately, there are several possible ways to mitigate the cost of task 5. As mentioned above, one way is to perform the computations for each layer in parallel, and even simultaneously with the gradient computation and other tasks. In the case of our block-tridiagonal approximation to the inverse, one can avoid computing any SVDs or matrix square roots by using an iterative Stein-equation solver (see Appendix I). And there are also ways of reducing matrix-inversion (and even matrix square-root) to a short sequence of matrix-matrix multiplications using iterative methods (Pan and Schreiber, 1991). Furthermore, because the matrices in question only change slowly

over time, one can consider hot-starting these iterative inversion methods from previous solutions. In the extreme case where d is very large, one can also consider using low-rank + diagonal approximations of the $\bar{A}_{i,j}$ and $G_{i,j}$ matrices maintained online (e.g. using a similar strategy as Le Roux et al. (2008)) from which inverses and/or SVDs can be more easily computed. Although based on our experience such approximations can, in some cases, lead to a substantial degradation in the quality of the updates.

While these ideas work reasonably well in practice, perhaps the simplest method, and the one we ended up settling on for our experiments, is to simply recompute the approximate Fisher inverse only every T_3 iterations (we used $T_3 = 20$ in our experiments). As it turns out, the curvature of the objective stays relatively stable during optimization, especially in the later stages, and so in our experience this strategy results in only a modest decrease in the quality of the updates.

If m is much smaller than d, the costs associated with task 6 can begin to dominate (provided T_3 is sufficiently large so that the cost of task 5 is relatively small). And unlike task 5, task 6 must be performed at every iteration. While the simplest solution is to increase m (while reaping the benefits of a less noisy gradient), in the case of the block-diagonal inverse it turns out that we can change the cost of task 6 from $C_5 \ell d^3$ to $C_5 \ell d^2 m$ by taking advantage of the low-rank structure of the stochastic gradient. The method for doing this is described below.

Let $\bar{\mathcal{A}}_i$ and $\mathcal{G}_i$ be matrices whose columns are the m $\bar{a}_i$'s and g_i's (resp.) associated with the current mini-batch. Let $\nabla_{W_i} h$ denote the gradient of h with respect to W_i, shaped as a matrix (instead of a vector). The estimate of $\nabla_{W_i} h$ over the mini-batch is given by $\frac{1}{m} \mathcal{G}_i \bar{\mathcal{A}}_{i-1}^\top$, which is of rank-$m$. From Section 3.2, computing the $\breve{F}^{-1} \nabla h$ amounts to computing $U_i = G_{i,i}^{-1} (\nabla_{W_i} h) \bar{A}_{i-1,i-1}^{-1}$. Substituting in our mini-batch estimate of $\nabla_{W_i} h$ gives

$$U_i = G_{i,i}^{-1} \left(\frac{1}{m} \mathcal{G}_i \bar{\mathcal{A}}_{i-1}^\top \right) \bar{A}_{i-1,i-1}^{-1} = \frac{1}{m} \left(G_{i,i}^{-1} \mathcal{G}_i \right) \left(\bar{\mathcal{A}}_{i-1}^\top \bar{A}_{i-1,i-1}^{-1} \right)$$

Direct evaluation of the expression on the right-hand side involves only matrix-matrix multiplications between matrices of size $m \times d$ and $d \times m$ (or between those of size $d \times d$ and $d \times m$), and thus we can reduce the cost of task 6 to $C_5 \ell d^2 m$.

Note that the use of standard ℓ_2 weight-decay is not compatible with this trick. This is because the contribution of the weight-decay term to each $\nabla_{W_i} h$ is νW_i, which will typically not be low-rank. Some possible ways around this issue include computing the weight-decay contribution $\nu \breve{F}^{-1} \theta$ separately and refreshing it only occasionally, or using a different regularization method, such as drop-out (Hinton et al., 2012) or weight-magnitude constraints.

Note that the adjustment technique for γ described in Appendix E.6 requires that, at every T_2 iterations, we compute 3 different versions of the update for each of 3 candidate values of γ. In an ideal implementation these could be computed in parallel with each other, although in the summary analysis below we will assume they are computed serially.

Summarizing, we have that with all of the various efficiency improvements discussed in this section, the average per-iteration computational cost of K-FAC, in terms of *serial* arithmetic oper-

ations, is

$$[(2 + \tau_1 + 2(1 + \chi_{mom})(1 + 2/T_2)\tau_2 + 1/T_1)C_1 + (1 + 2\tau_1)C_2]\ell d^2 m$$
$$+ (1 + 2/T_2)[(C_4/T_3 + C_6)\chi_{tri} + C_3/T_3(1 - \chi_{tri})]\ell d^3 + (1 + 2/T_2)C_5(1 - \chi_{tri})\ell d^2 \min\{d, m\}$$

where $\chi_{mom}, \chi_{tri} \in \{0, 1\}$ are flag variables indicating whether momentum and the block-tridiagonal inverse approximation (resp.) are used.

Plugging in the values of these various constants that we used in our experiments, for the block-diagonal inverse approximation ($\chi_{tri} = 0$) this becomes

$$(3.425C_1 + 1.25C_2)\ell d^2 m + 0.055C_3\ell d^3 + 1.1C_5\ell d^2 \min\{d, m\}$$

and for the block-tridiagonal inverse approximation ($\chi_{tri} = 1$)

$$(3.425C_1 + 1.25C_2)\ell d^2 m + (0.055C_4 + 1.1C_6)\ell d^3$$

which is to be compared to the per-iteration cost of SGD, which is given by

$$(2C_1 + C_2)\ell d^2 m$$

H Pseudocode for K-FAC

Algorithm 2 gives high-level pseudocode for the K-FAC method, with the details of how to perform the computations required for each major step left to their respective sections.

Algorithm 2 High-level pseudocode for K-FAC

• Initialize θ_1 (e.g. using a good method such as the ones described in Martens (2010) or Glorot and Bengio (2010))
• Choose initial values of λ (err on the side of making it too large)
• $\gamma \leftarrow \sqrt{\lambda + \eta}$
• $k \leftarrow 1$
while θ_k is not satisfactory **do**
  • Choose a mini-batch size m (e.g. using a fixed value, an adaptive rule, or some predefined schedule)
  • Select a random mini-batch $S' \subset S$ of training cases of size m
  • Select a random subset $S_1 \subset S'$ of size $\tau_1|S'|$
  • Select a random subset $S_2 \subset S'$ of size $\tau_2|S'|$
  • Perform a forward and backward pass on S' to estimate the gradient $\nabla h(\theta_k)$ (see Algorithm 1)
  • Perform an additional backwards pass on S_1 using random targets generated from the model's predictive distribution (as described in Appendix D)
  • Update the estimates of the required $\bar{A}_{i,j}$'s and $G_{i,j}$'s using the a_i's computed in forward pass for S_1, and the g_i's computed in the additional backwards pass for S_1 (as described Appendix D)
  • Choose a set Γ of new candidate γ's as described in Appendix E.6 (setting $\Gamma = \{\gamma\}$ if not adjusting γ at this iteration, i.e. if $k \not\equiv 0 \pmod{T_2}$)
  for each $\gamma \in \Gamma$ **do**
    if recomputing the approximate Fisher inverse this iteration (i.e. if $k \equiv 0 \pmod{T_3}$ or $k \leq 3$) **then**
      • Compute the approximate Fisher inverse (using the formulas derived in Section 3.2 or Section 3.3) from versions of the current $\bar{A}_{i,j}$'s and $G_{i,j}$'s which are modified as per the factored Tikhonov damping technique described in Appendix E.3 (but using γ as described in Appendix E.6)
    end if
    • Compute the update proposal Δ by multiplying current estimate of approximate Fisher inverse by the estimate of ∇h (using the formulas derived in Section 3.2 or Section 3.3). For layers with size $d < m$ consider using trick described at the end of Appendix G for increased efficiency.
    • Compute the final update δ from Δ as described in Appendix E.4 (or Appendix F if using momentum) where the matrix-vector products with F are estimated on S_2 using the a_i's computed in the forward pass
  end for
  • Select the δ and the new γ computing in the above loop that correspond to the lowest value of $M(\delta)$ (see Appendix E.6)
  if updating λ this iteration (i.e. if $k \equiv 0 \pmod{T_1}$) **then**
    • Update λ with the Levenberg-Marquardt style rule described in Appendix E.5
  end if
  • $\theta_{k+1} \leftarrow \theta_k + \delta$
  • $k \leftarrow k + 1$
end while

I Efficient techniques for inverting $A \otimes B \pm C \otimes D$

It is well known that $(A \otimes B)^{-1} = A^{-1} \otimes B^{-1}$, and that matrix-vector products with this matrix can thus be computed as $(A^{-1} \otimes B^{-1})v = \mathrm{vec}(B^{-1}VA^{-\top})$, where V is the matrix representation of v (so that $v = \mathrm{vec}(V)$).

Somewhat less well known is that there are also formulas for $(A \otimes B \pm C \otimes D)^{-1}$ which can be efficiently computed and likewise give rise to efficient methods for computing matrix-vector products.

First, note that $(A \otimes B \pm C \otimes D)^{-1}v = u$ is equivalent to $(A \otimes B \pm C \otimes D)u = v$, which is equivalent to the linear matrix equation $BUA^{\top} \pm DUC^{\top} = V$, where $u = \mathrm{vec}(U)$ and $v = \mathrm{vec}(V)$. This is known as a generalized Stein equation, and different examples of it have been studied in the control theory literature, where they have numerous applications. For a recent survey of this topic, see Simoncini (2014).

One well-known class of methods called Smith-type iterations (Smith, 1968) involve rewriting this matrix equation as a fixed point iteration and then carrying out this iteration to convergence. Interestingly, through the use of a special squaring trick, one can simulate 2^j of these iterations with only $\mathcal{O}(j)$ matrix-matrix multiplications.

Another class of methods for solving Stein equations involves the use of matrix decompositions (e.g. Chu, 1987; Gardiner et al., 1992). Here we will present such a method particularly well suited for our application, as it produces a formula for $(A \otimes B + C \otimes D)^{-1}v$, which after a fixed overhead cost (involving the computation of SVDs and matrix square roots), can be repeatedly evaluated for different choices of v using only a few matrix-matrix multiplications.

We will assume that A, B, C, and D are symmetric positive semi-definite, as they always are in our applications. We have

$$A \otimes B \pm C \otimes D = (A^{1/2} \otimes B^{1/2})(I \otimes I \pm A^{-1/2}CA^{-1/2} \otimes B^{-1/2}DB^{-1/2})(A^{1/2} \otimes B^{1/2})$$

Inverting both sides of the above equation gives

$$(A \otimes B \pm C \otimes D)^{-1} = (A^{-1/2} \otimes B^{-1/2})(I \otimes I \pm A^{-1/2}CA^{-1/2} \otimes B^{-1/2}DB^{-1/2})^{-1}(A^{-1/2} \otimes B^{-1/2})$$

Using the symmetric eigen/SVD-decomposition, we can write $A^{-1/2}CA^{-1/2} = E_1S_1E_1^{\top}$ and $B^{-1/2}DB^{-1/2} = E_2S_2E_2^{\top}$, where for $i \in \{1, 2\}$ the S_i are diagonal matrices and the E_i are unitary matrices.

This gives

$$\begin{aligned} I \otimes I \pm A^{-1/2}CA^{-1/2} \otimes B^{-1/2}DB^{-1/2} &= I \otimes I \pm E_1S_1E_1^{\top} \otimes E_2S_2E_2^{\top} \\ &= E_1E_1^{\top} \otimes E_2E_2^{\top} \pm E_1S_1E_1^{\top} \otimes E_2S_2E_2^{\top} \\ &= (E_1 \otimes E_2)(I \otimes I \pm S_1 \otimes S_2)(E_1^{\top} \otimes E_2^{\top}) \end{aligned}$$

so that

$$(I \otimes I \pm A^{-1/2} C A^{-1/2} \otimes B^{-1/2} D B^{-1/2})^{-1} = (E_1 \otimes E_2)(I \otimes I \pm S_1 \otimes S_2)^{-1}(E_1^\top \otimes E_2^\top)$$

Note that both $I \otimes I$ and $S_1 \otimes S_2$ are diagonal matrices, and thus the middle matrix $(I \otimes I \pm S_1 \otimes S_2)^{-1}$ is just the inverse of a diagonal matrix, and so can be computed efficiently.

Thus we have

$$\begin{aligned}(A \otimes B \pm C \otimes D)^{-1} &= (A^{-1/2} \otimes B^{-1/2})(E_1 \otimes E_2)(I \otimes I \pm S_1 \otimes S_2)^{-1}(E_1^\top \otimes E_2^\top)(A^{-1/2} \otimes B^{-1/2}) \\ &= (K_1 \otimes K_2)(I \otimes I \pm S_1 \otimes S_2)^{-1}(K_1^\top \otimes K_2^\top)\end{aligned}$$

where $K_1 = A^{-1/2} E_1$ and $K_2 = B^{-1/2} E_2$.

And so matrix-vector products with $(A \otimes B \pm C \otimes D)^{-1}$ can be computed as

$$(A \otimes B \pm C \otimes D)^{-1} v = \operatorname{vec}\left(K_2 \left[(K_2^\top V K_1) \oslash \left(\mathbf{1}\mathbf{1}^\top \pm s_2 s_1^\top\right)\right] K_1^\top\right)$$

where $E \oslash F$ denotes element-wise division of E by F, $s_i = \operatorname{diag}(S_i)$, and $\mathbf{1}$ is the vector of ones (sized as appropriate). Note that if we wish to compute multiple matrix-vector products with $(A \otimes B \pm C \otimes D)^{-1}$ (as we will in our application), the quantities K_1, K_2, s_1 and s_2 only need to be computed the first time, thus reducing the cost of any future such matrix-vector products, and in particular avoiding any additional SVD computations.

In the considerably simpler case where A and B are both scalar multiples of the identity, and ξ is the product of these multiples, we have

$$(\xi I \otimes I \pm C \otimes D)^{-1} = (E_1 \otimes E_2)(\xi I \otimes I \pm S_1 \otimes S_2)^{-1}(E_1^\top \otimes E_2^\top)$$

where $E_1 S_1 E_1^\top$ and $E_2 S_2 E_2^\top$ are the symmetric eigen/SVD-decompositions of C and D, respectively. And so matrix-vector products with $(\xi I \otimes I \pm C \otimes D)^{-1}$ can be computed as

$$(\xi I \otimes I \pm C \otimes D)^{-1} v = \operatorname{vec}\left(E_2 \left[(E_2^\top V E_1) \oslash \left(\xi \mathbf{1}\mathbf{1}^\top \pm s_2 s_1^\top\right)\right] E_1^\top\right)$$

J Computing $v^\top F v$ and $u^\top F v$ more efficiently

Note that the Fisher is given by

$$F = \mathrm{E}_{\hat{Q}_x}\left[J^\top F_R J\right]$$

where J is the Jacobian of $f(x, \theta)$ and F_R is the Fisher information matrix of the network's predictive distribution $R_{y|z}$, evaluated at $z = f(x, \theta)$ (where we treat z as the "parameters").

To compute the matrix-vector product Fv as estimated from a mini-batch we simply compute $J^\top F_R J v$ for each x in the mini-batch, and average the results. This latter operation can be computed in 3 stages (e.g. Martens, 2014), which correspond to multiplication of the vector v first by J, then by F_R, and then by $J^\top$.

Multiplication by J can be performed by a forward pass which is like a linearized version of the standard forward pass of Algorithm 1. As F_R is usually diagonal or diagonal plus rank-1, matrix-vector multiplications with it are cheap and easy. Finally, multiplication by $J^\top$ can be performed by a backwards pass which is essentially the same as that of Algorithm 1. See Schraudolph (2002); Martens (2014) for further details.

The naive way of computing $v^\top Fv$ is to compute Fv as above, and then compute the inner product of Fv with v. Additionally computing $u^\top Fv$ and $u^\top Fu$ would require another such matrix-vector product Fu.

However, if we instead just compute the matrix-vector products Jv (which requires only half the work of computing Fv), then computing $v^\top Fv$ as $(Jv)^\top F_R(Jv)$ is essentially free. And with Ju computed, we can similarly obtain $u^\top Fv$ as $(Ju)^\top F_R(Jv)$ and $u^\top Fu$ as $(Ju)^\top F_R(Ju)$.

This trick thus reduces the computational cost associated with computing these various scalars by roughly half.

K Proofs for Section 4

Proof of Theorem 1. First we will show that the given network transformation can be viewed as reparameterization of the network according to an invertible linear function ζ.

Define $\theta^\dagger = [\mathrm{vec}(W_1^\dagger)^\top \mathrm{vec}(W_2^\dagger)^\top \dots \mathrm{vec}(W_\ell^\dagger)^\top]^\top$, where $W_i^\dagger = \Phi_i^{-1} W_i \Omega_{i-1}^{-1}$ (so that $W_i = \Phi_i W_i^\dagger \Omega_{i-1}$) and let ζ be the function which maps $\theta^\dagger$ to θ. Clearly ζ is an invertible linear transformation.

If the transformed network uses $\theta^\dagger$ in place of θ then we have

$$\bar{a}_i^\dagger = \Omega_i \bar{a}_i \qquad \text{and} \qquad s_i^\dagger = \Phi_i^{-1} s_i$$

which we can prove by a simple induction. First note that $\bar{a}_0^\dagger = \Omega_0 \bar{a}_0$ by definition. Then, assuming by induction that $\bar{a}_{i-1}^\dagger = \Omega_{i-1} \bar{a}_{i-1}$, we have

$$s_i^\dagger = W_i^\dagger \bar{a}_{i-1}^\dagger = \Phi_i^{-1} W_i \Omega_{i-1}^{-1} \Omega_{i-1} \bar{a}_{i-1} = \Phi_i^{-1} W_i \bar{a}_{i-1} = \Phi_i^{-1} s_i$$

and therefore also

$$\bar{a}_i^\dagger = \Omega_i \bar{\phi}_i(\Phi_i s_i^\dagger) = \Omega_i \bar{\phi}_i(\Phi_i \Phi_i^{-1} s_i) = \Omega_i \bar{\phi}_i(s_i) = \Omega_i \bar{a}_i$$

And because $\Omega_\ell = I$, we have $\bar{a}_\ell^\dagger = \bar{a}_\ell$, or more simply that $a_\ell^\dagger = a_\ell$, and thus both the original network and the transformed one have the same output (i.e. $f(x, \theta) = f^\dagger(x, \theta^\dagger)$). From this it follows that $f^\dagger(x, \theta^\dagger) = f(x, \theta) = f(x, \zeta(\theta^\dagger))$, and thus the transformed network can be viewed as a reparameterization of the original network by $\theta^\dagger$. Similarly we have that $h^\dagger(\theta^\dagger) = h(\theta) = h(\zeta(\theta^\dagger))$.

The following lemma is adapted from Martens (2014).

Lemma 5. *Let ζ be some invertible affine function mapping $\theta^\dagger$ to θ, which reparameterizes the objective $h(\theta)$ as $h(\zeta(\theta^\dagger))$. Suppose that B and $B^\dagger$ are invertible matrices satisfying*

$$J_\zeta^\top B J_\zeta = B^\dagger$$

Then, additively updating θ by $\delta = -\alpha B^{-1}\nabla h$ is equivalent to additively updating $\theta^\dagger$ by $\delta^\dagger = -\alpha B^{\dagger -1}\nabla_{\theta^\dagger} h(\zeta(\theta^\dagger))$, in the sense that $\zeta(\theta^\dagger + \delta^\dagger) = \theta + \delta$.

Because $h^\dagger(\theta^\dagger) = h(\theta) = h(\zeta(\theta^\dagger))$ we have that $\nabla h^\dagger = \nabla_{\theta^\dagger} h(\zeta(\theta^\dagger))$. So, by the above lemma, to prove the theorem it suffices to show that $J_\zeta^\top \breve{F}\, J_\zeta = \breve{F}^\dagger$ and $J_\zeta^\top \tilde{F}\, J_\zeta = \tilde{F}^\dagger$.

Using $W_i = \Phi_i W_i^\dagger \Omega_{i-1}$ it is straightforward to verify that

$$J_\zeta = \operatorname{diag}(\Omega_0^\top \otimes \Phi_1, \Omega_1^\top \otimes \Phi_2, \ \ldots, \ \Omega_{\ell-1}^\top \otimes \Phi_\ell)$$

Because $s_i = \Phi_i s_i^\dagger$ and the fact that the networks compute the same outputs (so the loss derivatives are identical), we have by the chain rule that, $g_i^\dagger = \mathcal{D} s_i^\dagger = \Phi_i^\top \mathcal{D} s_i = \Phi_i^\top g_i$, and therefore

$$G_{i,j}^\dagger = \mathrm{E}\left[g_i^\dagger g_j^{\dagger\top}\right] = \mathrm{E}\left[\Phi_i^\top g_i (\Phi_i^\top g_i)^\top\right] = \Phi_i^\top \mathrm{E}\left[g_i g_i^\top\right] \Phi_j = \Phi_i^\top G_{i,j} \Phi_j$$

Furthermore,

$$\bar{A}_{i,j}^\dagger = \mathrm{E}\left[\bar{a}_i^\dagger \bar{a}_j^{\dagger\top}\right] = \mathrm{E}\left[(\Omega_i \bar{a}_i)(\Omega_j \bar{a}_j)^\top\right] = \Omega_i \mathrm{E}\left[\bar{a}_i \bar{a}_j^\top\right] \Omega_j^\top = \Omega_i \bar{A}_{i,j} \Omega_j^\top$$

Using these results we may express the Kronecker-factored blocks of the approximate Fisher $\tilde{F}^\dagger$, as computed using the transformed network, as follows:

$$\begin{aligned}\tilde{F}_{i,j}^\dagger = \bar{A}_{i-1,j-1}^\dagger \otimes G_{i,j}^\dagger = \Omega_{i-1}\bar{A}_{i-1,j-1}\Omega_{j-1}^\top \otimes \Phi_i^\top G_{i,j} \Phi_j &= (\Omega_{i-1} \otimes \Phi_i^\top)(\bar{A}_{i-1,j-1} \otimes G_{i,j})(\Omega_{j-1}^\top \otimes \Phi_j) \\ &= (\Omega_{i-1} \otimes \Phi_i^\top)\tilde{F}_{i,j}(\Omega_{j-1}^\top \otimes \Phi_j)\end{aligned}$$

Given this identity we thus have

$$\begin{aligned}\breve{F}^\dagger &= \operatorname{diag}\left(\tilde{F}_{1,1}^\dagger, \tilde{F}_{2,2}^\dagger, \ \ldots, \ \tilde{F}_{\ell,\ell}^\dagger\right) \\ &= \operatorname{diag}\left((\Omega_0 \otimes \Phi_1^\top)\tilde{F}_{1,1}(\Omega_0^\top \otimes \Phi_1), (\Omega_1 \otimes \Phi_2^\top)\tilde{F}_{2,2}(\Omega_1^\top \otimes \Phi_2), \ \ldots, \ (\Omega_{\ell-1} \otimes \Phi_\ell^\top)\tilde{F}_{\ell,\ell}(\Omega_{\ell-1}^\top \otimes \Phi_\ell)\right) \\ &= \operatorname{diag}(\Omega_0 \otimes \Phi_1^\top, \Omega_1 \otimes \Phi_2^\top, \ \ldots, \ \Omega_{\ell-1} \otimes \Phi_\ell^\top) \operatorname{diag}\left(\tilde{F}_{1,1}, \tilde{F}_{2,2}, \ \ldots, \ , \tilde{F}_{\ell,\ell}\right) \\ &\qquad \cdot \operatorname{diag}(\Omega_0^\top \otimes \Phi_1, \Omega_1^\top \otimes \Phi_2, \ \ldots, \ \Omega_{\ell-1}^\top \otimes \Phi_\ell) \\ &= J_\zeta^\top \breve{F} J_\zeta\end{aligned}$$

We now turn our attention to the $\hat{F}$ (see Section 3.3 for the relevant notation).

First note that

$$\begin{aligned}\Psi^{\dagger}_{i,i+1} &= \tilde{F}^{\dagger}_{i,i+1}\tilde{F}^{\dagger-1}_{i+1,i+1} = (\Omega_{i-1}\otimes\Phi_i^{\top})\tilde{F}_{i,i+1}(\Omega_i^{\top}\otimes\Phi_{i+1})\left((\Omega_i\otimes\Phi_{i+1}^{\top})\tilde{F}_{i+1,i+1}(\Omega_i^{\top}\otimes\Phi_{i+1})\right)^{-1}\\ &= (\Omega_{i-1}\otimes\Phi_i^{\top})\tilde{F}_{i,i+1}(\Omega_i^{\top}\otimes\Phi_{i+1})(\Omega_i^{\top}\otimes\Phi_{i+1})^{-1}\tilde{F}^{-1}_{i+1,i+1}(\Omega_i\otimes\Phi_{i+1}^{\top})^{-1}\\ &= (\Omega_{i-1}\otimes\Phi_i^{\top})\tilde{F}_{i,i+1}\tilde{F}^{-1}_{i+1,i+1}(\Omega_i\otimes\Phi_{i+1}^{\top})^{-1}\\ &= (\Omega_{i-1}\otimes\Phi_i^{\top})\Psi_{i,i+1}(\Omega_i\otimes\Phi_{i+1}^{\top})^{-1}\end{aligned}$$

and so

$$\begin{aligned}\Sigma^{\dagger}_{i|i+1} &= \tilde{F}^{\dagger}_{i,i} - \Psi^{\dagger}_{i,i+1}\tilde{F}^{\dagger}_{i+1,i+1}\Psi^{\dagger\top}_{i,i+1}\\ &= (\Omega_{i-1}\otimes\Phi_i^{\top})\tilde{F}_{i,i}(\Omega_{i-1}^{\top}\otimes\Phi_i)\\ &\quad - (\Omega_{i-1}\otimes\Phi_i^{\top})\Psi_{i,i+1}(\Omega_i\otimes\Phi_{i+1}^{\top})^{-1}(\Omega_i\otimes\Phi_{i+1}^{\top})\tilde{F}_{i+1,i+1}(\Omega_i^{\top}\otimes\Phi_{i+1})(\Omega_i\otimes\Phi_{i+1}^{\top})^{-\top}\\ &\qquad\qquad\cdot\Psi^{\top}_{i,i+1}(\Omega_{i-1}\otimes\Phi_i^{\top})^{\top}\\ &= (\Omega_{i-1}\otimes\Phi_i^{\top})(\tilde{F}_{i,i} - \Psi_{i,i+1}\tilde{F}_{i+1,i+1}\Psi^{\top}_{i,i+1})(\Omega_{i-1}^{\top}\otimes\Phi_i)\\ &= (\Omega_{i-1}\otimes\Phi_i^{\top})\Sigma_{i|i+1}(\Omega_{i-1}^{\top}\otimes\Phi_i)\end{aligned}$$

Also, $\Sigma^{\dagger}_{\ell} = \tilde{F}^{\dagger}_{\ell,\ell} = (\Omega_{\ell-1}\otimes\Phi_{\ell}^{\top})\tilde{F}_{\ell,\ell}(\Omega_{\ell-1}^{\top}\otimes\Phi_{\ell}) = (\Omega_{\ell-1}\otimes\Phi_{\ell}^{\top})\Sigma_{\ell}(\Omega_{\ell-1}^{\top}\otimes\Phi_{\ell})$.

From these facts it follows that

$$\begin{aligned}\Lambda^{\dagger-1} &= \operatorname{diag}\left(\Sigma^{\dagger}_{1|2},\Sigma^{\dagger}_{2|3},\,\ldots,\,\Sigma^{\dagger}_{\ell-1|\ell},\Sigma^{\dagger}_{\ell}\right)\\ &= \operatorname{diag}\left((\Omega_0\otimes\Phi_1^{\top})\Sigma_{1|2}(\Omega_0\otimes\Phi_1^{\top}),(\Omega_1\otimes\Phi_2^{\top})\Sigma_{2|3}(\Omega_1\otimes\Phi_2^{\top}),\,\ldots,\right.\\ &\qquad\left.(\Omega_{\ell-2}\otimes\Phi_{\ell-1}^{\top})\Sigma_{\ell-1|\ell}(\Omega_{\ell-2}\otimes\Phi_{\ell-1}^{\top}),(\Omega_{\ell-1}\otimes\Phi_{\ell}^{\top})\Sigma_{\ell}(\Omega_{\ell-1}^{\top}\otimes\Phi_{\ell})\right)\\ &= \operatorname{diag}(\Omega_0\otimes\Phi_1^{\top},\Omega_1\otimes\Phi_2^{\top},\,\ldots,\,\Omega_{\ell-2}\otimes\Phi_{\ell-1}^{\top},\Omega_{\ell-1}\otimes\Phi_{\ell}^{\top})\operatorname{diag}\left(\Sigma_{1|2},\Sigma_{2|3},\,\ldots,\,\Sigma_{\ell-1|\ell},\Sigma_{\ell}\right)\\ &\qquad\operatorname{diag}(\Omega_0^{\top}\otimes\Phi_1,\Omega_1^{\top}\otimes\Phi_2,\,\ldots,\,\Omega_{\ell-2}^{\top}\otimes\Phi_{\ell-1},\Omega_{\ell-1}^{\top}\otimes\Phi_{\ell})\\ &= J_{\zeta}^{\top}\Lambda^{-1}J_{\zeta}\end{aligned}$$

Inverting both sides gives $\Lambda^{\dagger} = J_{\zeta}^{-1}\Lambda J_{\zeta}^{-\top}$.

Next, observe that

$$\Psi^{\dagger\top}_{i,i+1}(\Omega_{i-1}^{\top}\otimes\Phi_i)^{-1} = (\Omega_i\otimes\Phi_{i+1}^{\top})^{-\top}\Psi^{\top}_{i,i+1}(\Omega_{i-1}\otimes\Phi_i^{\top})^{\top}(\Omega_{i-1}^{\top}\otimes\Phi_i)^{-1} \quad = (\Omega_i^{\top}\otimes\Phi_{i+1})^{-1}\Psi^{\top}_{i,i+1}$$

from which it follows that

$$\Xi^{\dagger\top} J_\zeta^{-1} = \begin{bmatrix} I & & & & \\ -\Psi_{1,2}^{\dagger\top} & I & & & \\ & -\Psi_{2,3}^{\dagger\top} & I & & \\ & & \ddots & \ddots & \\ & & & -\Psi_{\ell-1,\ell}^{\dagger\top} & I \end{bmatrix} \operatorname{diag}((\Omega_0^\top \otimes \Phi_1)^{-1}, (\Omega_1^\top \otimes \Phi_2)^{-1}, \ldots, (\Omega_{\ell-1}^\top \otimes \Phi_\ell)^{-1})$$

$$= \begin{bmatrix} (\Omega_0^\top \otimes \Phi_1)^{-1} & & & & \\ -\Psi_{1,2}^{\dagger\top}(\Omega_0^\top \otimes \Phi_1)^{-1} & (\Omega_1^\top \otimes \Phi_2)^{-1} & & & \\ & -\Psi_{2,3}^{\dagger\top}(\Omega_1^\top \otimes \Phi_2)^{-1} & (\Omega_2^\top \otimes \Phi_3)^{-1} & & \\ & & \ddots & \ddots & \\ & & & -\Psi_{\ell-1,\ell}^{\dagger\top}(\Omega_{\ell-2}^\top \otimes \Phi_{\ell-1})^{-1} & (\Omega_{\ell-1}^\top \otimes \Phi_\ell)^{-1} \end{bmatrix}$$

$$= \begin{bmatrix} (\Omega_0^\top \otimes \Phi_1)^{-1} & & & & \\ -(\Omega_0^\top \otimes \Phi_1)^{-1}\Psi_{1,2}^\top & (\Omega_1^\top \otimes \Phi_2)^{-1} & & & \\ & -(\Omega_1^\top \otimes \Phi_2)^{-1}\Psi_{2,3}^\top & (\Omega_2^\top \otimes \Phi_3)^{-1} & & \\ & & \ddots & \ddots & \\ & & & -(\Omega_{\ell-2}^\top \otimes \Phi_{\ell-1})^{-1}\Psi_{\ell-1,\ell}^\top & (\Omega_{\ell-1}^\top \otimes \Phi_\ell)^{-1} \end{bmatrix}$$

$$= \operatorname{diag}((\Omega_0^\top \otimes \Phi_1)^{-1}, (\Omega_1^\top \otimes \Phi_2)^{-1}, \ldots, (\Omega_{\ell-1}^\top \otimes \Phi_\ell)^{-1}) \begin{bmatrix} I & & & & \\ -\Psi_{1,2}^\top & I & & & \\ & -\Psi_{2,3}^\top & I & & \\ & & \ddots & \ddots & \\ & & & -\Psi_{\ell-1,\ell}^\top & I \end{bmatrix}$$

$$= J_\zeta^{-1}\Xi^\top$$

Combining $\Lambda^\dagger = J_\zeta^{-1}\Lambda J_\zeta^{-\top}$ and $\Xi^{\dagger\top} J_\zeta^{-1} = J_\zeta^{-1}\Xi^\top$ we have

$$\begin{aligned} \hat{F}^{\dagger-1} = \Xi^{\dagger\top}\Lambda^\dagger\Xi^\dagger = \Xi^{\dagger\top} J_\zeta^{-1}\Lambda J_\zeta^{-\top}\Xi^\dagger = (\Xi^{\dagger\top} J_\zeta^{-1})\Lambda(\Xi^{\dagger\top} J_\zeta^{-1})^\top &= (J_\zeta^{-1}\Xi^\top)\Lambda(J_\zeta^{-1}\Xi^\top)^\top \\ &= J_\zeta^{-1}\Xi^\top\Lambda\Xi J_\zeta^{-\top} \\ &= J_\zeta^{-1}\hat{F}^{-1}J_\zeta^{-\top} \end{aligned}$$

Inverting both sides gives $\hat{F}^\dagger = J_\zeta^\top \hat{F} J_\zeta$ as required.

□

Proof of Corollary 3. First note that a network which is transformed so that $G_{i,i}^\dagger = I$ and $\bar{A}_{i,i}^\dagger = I$ will satisfy the required properties. To see this, note that $\mathrm{E}[g_i^\dagger g_i^{\dagger\top}] = G_{i,i}^\dagger = I$ means that $g_i^\dagger$ is whitened with respect to the model's distribution by definition (since the expectation is taken with respect to the model's distribution), and furthermore we have that $\mathrm{E}[g_i^\dagger] = 0$ by default (e.g. using Lemma 4), so $g_i^\dagger$ is centered. And since $\mathrm{E}[a_i^\dagger a_i^{\dagger\top}]$ is the square submatrix of $\bar{A}_{i,i}^\dagger = I$ which leaves

out the last row and column, we also have that $\mathrm{E}[a_i^\dagger a_i^{\dagger\top}] = I$ and so $a_i^\dagger$ is whitened. Finally, observe that $\mathrm{E}[a_i^\dagger]$ is given by the final column (or row) of $\bar{A}_{i,i}$, excluding the last entry, and is thus equal to 0, and so $a_i^\dagger$ is centered.

Next, we note that if $G_{i,i}^\dagger = I$ and $\bar{A}_{i,i}^\dagger = I$ then

$$\breve{F}^\dagger = \mathrm{diag}\left(\bar{A}_{0,0}^\dagger \otimes G_{1,1}^\dagger, \bar{A}_{1,1}^\dagger \otimes G_{2,2}^\dagger, \ \ldots, \ \bar{A}_{\ell-1,\ell-1}^\dagger \otimes G_{\ell,\ell}^\dagger\right) = \mathrm{diag}\left(I \otimes I, I \otimes I, \ \ldots, \ I \otimes I\right) = I$$

and so $-\alpha \breve{F}^{-1} \nabla h^\dagger = -\alpha \nabla h^\dagger$ is indeed a standard gradient descent update.

Finally, we observe that there are choices of Ω_i and Φ_i which will make the transformed model satisfy $G_{i,i}^\dagger = I$ and $\bar{A}_{i,i}^\dagger = I$. In particular, from the proof of Theorem 1 we have that $G_{i,j}^\dagger = \Phi_i^\top G_{i,j} \Phi_j$ and $\bar{A}_{i,j}^\dagger = \Omega_i \bar{A}_{i,j} \Omega_j^\top$, and so taking $\Phi_i = G_{i,i}^{-1/2}$ and $\Omega_i = \bar{A}_{i,i}^{-1/2}$ works.

The result now follows from Theorem 1.

□

L Additional related work

The Hessian-free optimization method of Martens (2010) uses linear conjugate gradient (CG) to optimize local quadratic models of the form of eqn. 5 (subject to an adaptive Tikhonov damping technique) in lieu of directly solving it using matrix inverses. As discussed in the introduction, the main advantages of K-FAC over HF are twofold. Firstly, K-FAC uses an efficiently computable direct solution for the inverse of the curvature matrix and thus avoids the costly matrix-vector products associated with running CG within HF. Secondly, it can estimate the curvature matrix from a lot of data by using an online exponentially-decayed average, as opposed to relatively small-sized fixed mini-batches used by HF. The cost of doing this is of course the use of an inexact approximation to the curvature matrix.

Le Roux et al. (2008) proposed a neural network optimization method known as TONGA based on a block-diagonal approximation of the *empirical* Fisher where each block corresponds to the weights associated with a particular unit. By contrast, K-FAC uses *much* larger blocks, each of which corresponds to all the weights within a particular layer. The matrices which are inverted in K-FAC are roughly the same size as those which are inverted in TONGA, but rather than there being one per unit as in TONGA, there are only two per layer. Therefore, K-FAC is significantly less computationally intensive than TONGA, despite using what is arguably a much more accurate approximation to the Fisher.

Concurrently with this work Povey et al. (2015) has developed a neural network optimization method which uses a block-diagonal Kronecker-factored approximation similar to the one from Heskes (2000). This approach differs from K-FAC in numerous ways, including its use of the empirical Fisher (which doesn't work as well as the standard Fisher in our experience – see Appendix D), and its use of only a basic factored Tikhonov damping technique without adaptive re-scaling

or any form of momentum. One interesting idea introduced by Povey et al. (2015) is a particular method for maintaining an online low-rank plus diagonal approximation of the factor matrices for each block, which allows their inverses to be computed more efficiently (although subject to an approximation). While our experiments with similar kinds of methods for maintaining such online estimates found that they performed poorly in practice compared to the solution of refreshing the inverses only occasionally (see Appendix G), the particular one developed by Povey et al. (2015) could potentially still work well, and may be especially useful for networks with very wide layers.

M Additional experiments and results plots

In our initial experiment we examined the relationship between the mini-batch size m and the per-iteration rate of progress made by K-FAC and the baseline on the MNIST problem. The results from this experiment are plotted in Figure 8. They strongly suggest that the per-iteration rate of progress of K-FAC tends to a superlinear function of m (which can be most clearly seen by examining the plots of training error vs training cases processed), which is to be contrasted with the baseline, where increasing m has a much smaller effect on the per-iteration rate of progress, and with K-FAC without momentum, where the per-iteration rate of progress seems to be a linear or slightly sublinear function of m. It thus appears that the main limiting factor in the convergence of K-FAC (with momentum applied) is the noise in the gradient, at least in later stages of optimization, and that this is not true of the baseline to nearly the same extent. This would seem to suggest that K-FAC, much more than SGD, would benefit from a massively parallel distributed implementation which makes use of more computational resources than a single GPU.

But even in the single CPU/GPU setting, the fact that the per-iteration rate of progress tends to a *superlinear* function of m, while the per-iteration computational cost of K-FAC is a roughly linear function of m, suggests that in order to obtain the best per-second rate of progress with K-FAC, we should use a rapidly increasing schedule for m. To this end we designed an exponentially increasing schedule for m, given by $m_i = \min(m_1 \exp((i-1)/b), |S|)$, where i is the current iteration, $m_1 = 1000$, and where b is chosen so that $m_{500} = |S|$. The approach of increasing the mini-batch size in this way is analyzed by Friedlander and Schmidt (2012). Note that for other neural network optimization problems, such as ones involving larger training datasets than these autoencoder problems, a more slowly increasing schedule, or one that stops increasing well before m reaches $|S|$, may be more appropriate.

From Figure 9, which consists of results from our second (main) experiment described in Section 7, we can see that the block-tridiagonal version of K-FAC has a per-iteration rate of progress which is typically 25% to 40% larger than the simpler block-diagonal version. This observation provides empirical support for the idea that the block-tridiagonal approximate inverse Fisher $\hat{F}^{-1}$ is a more accurate approximation of F^{-1} than the block-diagonal approximation $\breve{F}^{-1}$. However, due to the higher cost of the iterations in the block-tridiagonal version, its overall per-second rate of progress seems to be only moderately higher than the block-diagonal version's (as seen in Figure 4), depending on the problem.

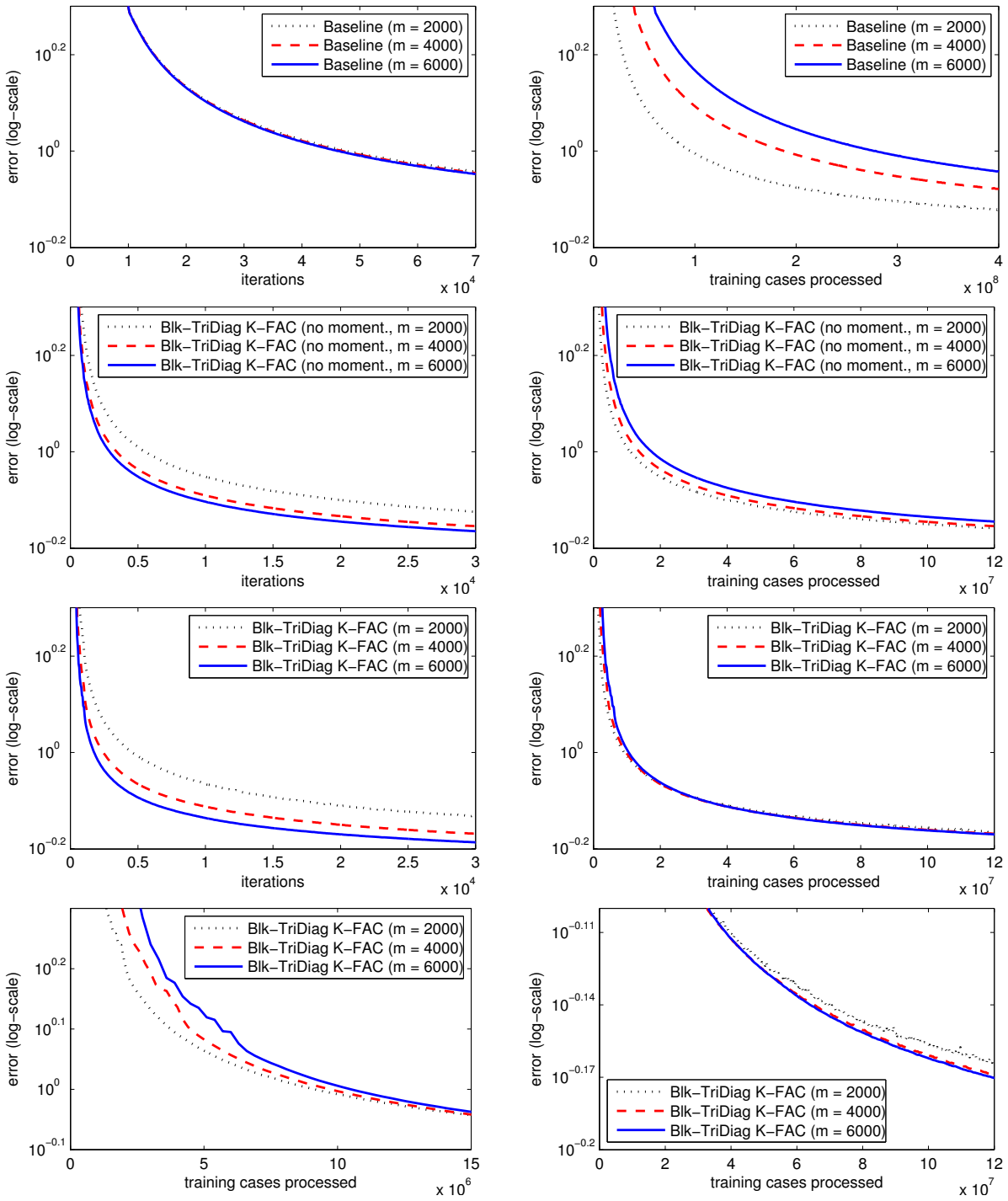

Figure 8: Results from our first experiment examining the relationship between the mini-batch size m and the per-iteration progress (**left column**) or per-training case progress (**right column**) progress made by K-FAC on the MNIST deep autoencoder problem. Here, "Blk-TriDiag K-FAC" is the block-tridiagonal version of K-FAC, and "Blk-Diag K-FAC" is the block-diagonal version, and "no moment." indicates that momentum was not used. The **bottom row** consists of zoomed-in versions of the right plot from the row above it, with the left plot concentrating on the beginning stage of optimization, and the right plot concentrating on the later stage. Note that the x-axes of these two last plots are at significantly different scales (10^6 vs 10^7).

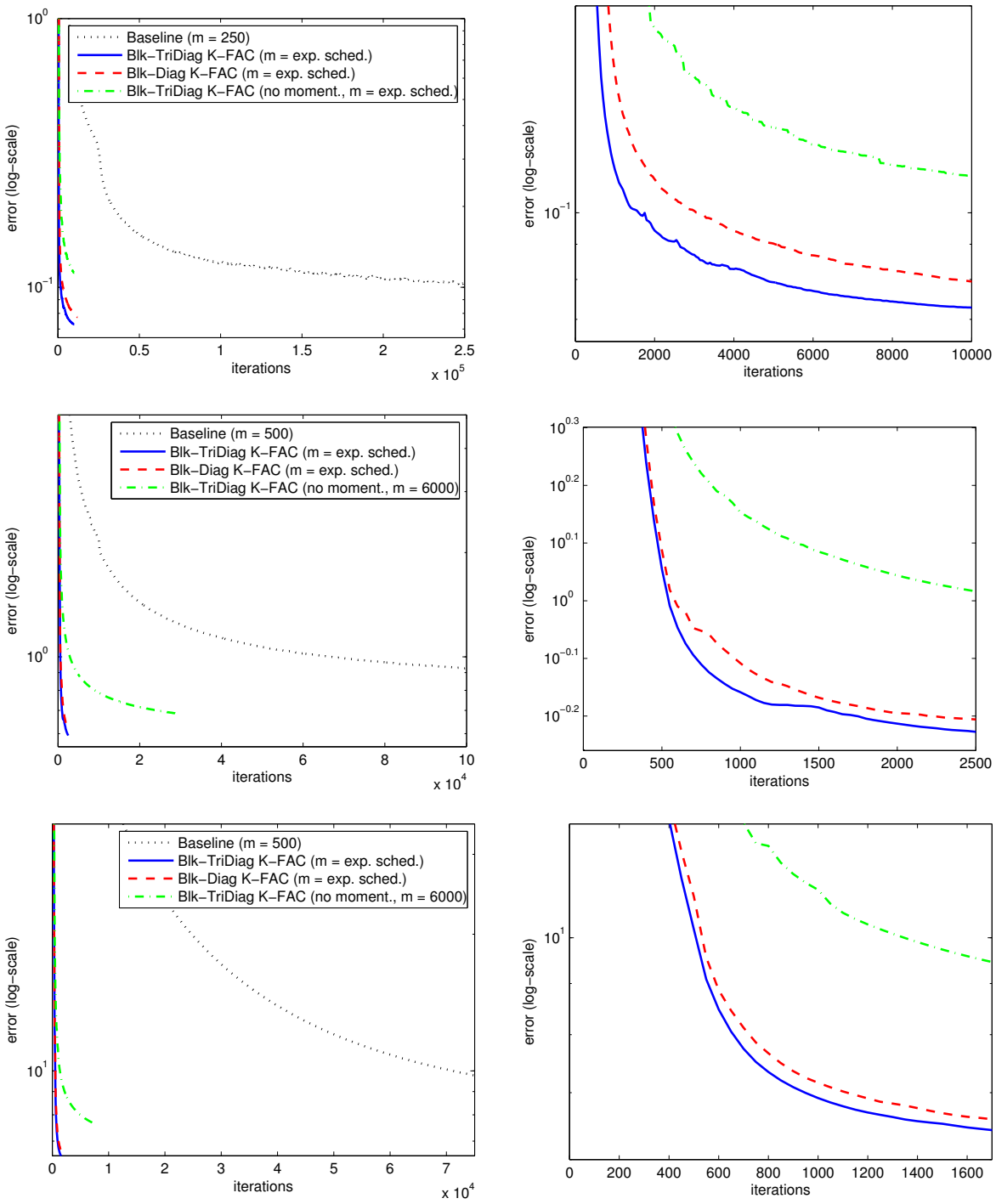

Figure 9: More results from our second experiment showing training error versus iteration on the CURVES (**top** row), MNIST (**middle** row), and FACES (**bottom** row) deep autoencoder problems. The plots on the right are zoomed in versions of those on the left which highlight the difference in per-iteration progress made by the different versions of K-FAC.