Gradient Descent on Neurons and Its Link to Approximate Second-Order Optimization

Frederik Benzing
Department of Computer Science,
ETH Zurich,
8092 Zurich, Switzerland

Abstract

Second-order optimizers hold the potential to speed up neural network training, but due to the enormous size of the curvature matrix, they typically require approximations to be computationally tractable. The most successful family of approximations are Kronecker-Factored, block-diagonal curvature estimates (KFAC). While they offer notable performance gains compared to first-order optimizers, it had remained unclear how close they are to exact second-order updates. Our first contribution is to show that exact second-order updates can be evaluated exactly and efficiently, given that the curvature is measured on a moderately sized mini-batch. Secondly, we conduct thorough experiments to show that KFAC's performance is not due to using second-order information. This challenges widely held believes and immediately raises the question why KFAC performs so well. We answer this question by showing that KFAC performs gradient descent on neurons rather than weights. Finally, we propose a new optimizer "FOOF", which improves over KFAC in terms of computational cost and data-efficiency.

1 Introduction

Second-order information of neural networks is of fundamental theoretical interest and has important applications in a number of contexts like optimization, bayesian machine learning, meta-learning, sparsification and continual learning [LeCun et al., 1990, Hochreiter and Schmidhuber, 1997, MacKay, 1992, Bengio, 2000, Martens et al., 2010, Grant et al., 2018, Sutskever et al., 2013, Dauphin et al., 2014, Blundell et al., 2015, Kirkpatrick et al., 2017, Graves, 2011]. However, due to the enormous parameter count of modern neural networks working with the full curvature matrix is infeasible and this has inspired many approximations. Understanding how accurate known approximations are and developing better ones is an important topic at the intersection of theory and practice.

A family of approximations that has been particularly successful are Kronecker-factored, block diagonal approximations of the curvature. They were originally proposed in the context of optimization [Martens and Grosse, 2015] and have lead to many further developments and extensions in this area [Grosse and Salakhudinov, 2015, Ba et al., 2016, Desjardins et al., 2015, Goldfarb et al., 2020, Botev et al., 2017, George et al., 2018, Martens et al., 2018, Osawa et al., 2019] and have also proven influential in various other contexts, like bayesian inference, meta learning and continual learning [Ritter et al., 2018b, Ritter et al., 2018a, Dangel et al., 2020, Zhang et al., 2018a, Wu et al., 2017, Grant et al., 2018].

Our first contribution is to show that second-order updates can be evaluated efficiently, if the curvature is calculated on a moderately sized mini-batch. Our implementation requires less than double the wall-clock time of vanilla SGD per parameter update. This is an important tool to evaluate the quality of approximate second-order updates. Moreover, we show that with an additional trick, it can be used to draw exact samples from Laplace posteriors of neural networks.

We then focus on applying our tool to optimization, studying KFAC [Martens and Grosse, 2015] as an example for Kronecker-factored optimizers and more generally, as one of most effective optimizers for

Preprint.

neural networks [Ba et al., 2016, Osawa et al., 2019, Martens et al., 2021]. We make several surprising discoveries challenging fundamental beliefs about second-order information encoded in Kronecker-factored approximations. (1) We find that Kronecker-factored approximate second-order updates are barely more aligned with true second-order updates than vanilla gradients. (2) Even more surprisingly, the approximate updates significantly outperform their exact updates in terms of optimization performance, also when controlling for the fact that exact updates subsample the curvature matrix.

This is clear evidence that Kronecker-factored methods do not owe their performance to being related to second-order updates and immediately raises the question why they nevertheless perform so well. We answer this question showing theoretically as well as experimentally that due to its approximations, KFAC is similar to a new, principled first-order optimizer. This optimizer performs gradient-descent on neurons rather than weights and we demonstrate that it offers improvements over standard first-order optimizers as well as KFAC, both in terms of computation time and optimization-progress per parameter update.

In summary, our three main contributions are:

- We develop a tool to efficiently and exactly evaluate second-order updates given that the curvature is measured on a mini-batch of data. We also show how underlying techniques can be used to sample from exact Laplace posteriors.
- We give a new explanation for why KFAC, one of the best performing optimization algorithms, is so effective. We demonstrate that the traditional second-order view is inconsistent with experimental results and show that our new first-order perspective offers a simple and improved explanation.
- We develop the algorithm "FOOF" that performs gradient descent on neurons and is computationally cheaper than KFAC and makes more progress per parameter update.

Structure of the Paper. We provide background and an outline how to efficiently compute exact second-order updates in Section 2. The remainder of the paper is split into two parts. In Section 3 we carefully establish that KFAC is not closely related to second-order information. In Section 4 we introduce gradient descent on neurons ("FOOF"). We demonstrate that KFAC's performance relies on similarity to FOOF and that FOOF offers further improvements.

2 Background and Efficient Products of Inverse Curvature and Vectors

The most straight-forward definition of a "curvature matrix" is the Hessian $\mathbf{H}$ of the loss with respect to the parameters. However, in most contexts (e.g. optimization or Laplace posteriors), it is necessary or desirable to work with a positive definite approximations of the Hessian, i.e. an approximation of the form $\mathbf{H} \approx \mathbf{G}\mathbf{G}^T$; examples for such approximations include Generalised Gauss Newton matrices and the Fisher Information. For simplicity, we will now focus on the Fisher, but our methods straightforwardly apply to any case where the columns of $\mathbf{G}$ are Jacobians. The Fisher is defined as

$$\mathbf{F} = \mathbb{E}_{X\sim\mathcal{X}}\mathbb{E}_{y\sim p(X|\mathbf{w})}\left[\mathbf{g}(X,y)\mathbf{g}(X,y)^T\right] \qquad (1)$$

where $X \sim \mathcal{X}$ is a sample from the input distribution, $y \sim p(\cdot \mid X, \mathbf{w})$ is a sample from the model's output distribution (rather than the label given by the dataset, see [Kunstner et al., 2019] for a discussion of this difference). $\mathbf{g}(X,y)$ is the "gradient", i.e. the (columnised) derivative of the negative log-liklihood of (X,y) with respect to the model parameters $\mathbf{w} \in \mathbb{R}^n$.

The natural gradient method preconditions normal first-order updates $\mathbf{v}$ by the inverse Fisher. Concretely, we update parameters in the direction of $(\lambda\mathbf{I}+\mathbf{F})^{-1}\mathbf{v}$. Here, λ is a damping term and can be seen as establishing a trust region. Natural Gradients were proposed by Amari and colleagues, see e.g. [Amari, 1998] and were motivated from an information geometric perspective. The Fisher is equal to the Hessian of the negative log-likelihood under the model's output distribution and thereby closely related to the standard Hessian [Martens, 2014, Pascanu and Bengio, 2013], so that Natural Gradients are often interpreted as a second-order method [Martens and Grosse, 2015].

We will show that, for a subsampled Fisher, natural gradients can be computed efficiently. The insights for this are simple: When the Fisher is subsampled on D datapoints, it can be written as $\mathbf{F} = \mathbf{G}\mathbf{G}^T$, where $\mathbf{G} \in \mathbb{R}^{n\times D}$ (recall our definition of datapoint: r input images give rise to $k\cdot r$ datapoints, where k is the number of possible labels). Then, one can apply the Woodburry matrix inversion lemma and repeatedly exploit the fact that, for a single layer and datapoint, gradients are rank-1 matrices. The computations required on top of a standard for- and backward pass require space $O(D^2)$ and time $O(D^3)$, which is negligible even for moderately large D.

Most of these ideas have been described independently [Agarwal et al., 2019, Ren and Goldfarb, 2019]. In addition, we provide a way to decrease the bias of mini-batch second order updates, present a more efficient implementation and introduce a new amoritisation technique. These differences along with the details omitted above are discussed in Appendix I.

With an additional trick [Doucet, 2010, Hoffman and Ribak, 1991], we can use these insights to sample efficiently and exactly from the Laplace posterior, see Appendix C.

A more detailed summary of related work can be found in Appendix H.

2.1 Notation and Terminology

We typically focus on one layer of a neural network. For simplicity of notation, we consider fully-connected layers, but results can easily be extended to architectures with parameter sharing, like CNNs or RNNs.

We denote the layer's weight matrix by $\mathbf{W} \in \mathbb{R}^{n\times m}$ and its input-activations (after the previous' layer nonlinearity) by $\mathbf{A} \in \mathbb{R}^{m\times D}$, where D is the number of datapoints. The layer's output activations (before the non-linearity) are equal to $\mathbf{B} = \mathbf{W}\mathbf{A} \in \mathbb{R}^{n\times D}$ and we denote the partial derivates of the loss L with respect to these outputs (usually computed by backpropagation) by $\mathbf{E} = \frac{\partial L}{\partial \mathbf{B}}$. If the label is sampled from the model's output distribution, as is the case for the Fisher (1), we will use $\mathbf{E}_F$ rather than $\mathbf{E}$.

We use the term "**datapoint**" for a pair of input and label (X, y). In the context of the Fisher Information, the label will always be sampled from the model's output distribution, see also eq (1).

As is common in the ML context, we will use the term "**second-order method**" for algorithms that use (approximate) second derivatives. The term "**first-order method**" will refer to algorithms which only use first derivatives or quantities that are independent of the loss, i.e. "zero-th" order terms.

3 Exact Natural Gradients and KFAC

In this section, we will show that KFAC is not strongly related to second-order information. We start with a brief review of KFAC, see Appendix E for more details, and then proceed with experiments.

3.1 Review of KFAC

KFAC makes two approximations to the Fisher. Firstly, it only considers diagonal blocks of the Fisher, where each block corresponds to one layer of the network. Secondly, each block is approximated as a Kronecker-product $(\mathbf{A}\mathbf{A}^T) \otimes (\mathbf{E}_F\mathbf{E}_F^T)$. This approximation of the Fisher leads to the following update

$$(\Delta\mathbf{W})^T = \left(\mathbf{A}\mathbf{A}^T + \lambda_A\mathbf{I}\right)^{-1}\left(\mathbf{A}\mathbf{E}^T\right)\left(\mathbf{E}_F\mathbf{E}_F^T + \lambda_E\mathbf{I}\right)^{-1} \tag{2}$$

where λ_A, λ_E are damping terms satisfying $\lambda_A \cdot \lambda_E = \lambda$ for a hyperparmater λ and $\frac{\lambda_A}{\lambda_E} = \frac{\mathrm{Tr}(\mathbf{A}\mathbf{A}^T)}{\mathrm{Tr}(\mathbf{E}_F\mathbf{E}_F^T)}$. The Kronecker-factors $\mathbf{A}\mathbf{A}^T$ and $\mathbf{E}_F\mathbf{E}_F^T$ are updated as exponentially moving averages, so that they incorporate data from several recent mini-batches.

If we use only one datapoint to measure the Fisher, then the Kronecker-factored approximation of each block is exact. For D datapoints, the approximation is equivalent to replacing a rank D matrix by a rank 1 matrix. This seems imprecise and is often justified by an independence assumption. However, this assumption is strongly violated theoretically, see Appendix E.

Comparing subsampled natural gradients to KFAC, there are two differences: (1) KFAC can use more data to estimate the Fisher, due to its exponential moving averages. (2) For a given mini-batch, natural gradients are exact, while KFAC makes two approximations.

A priori, it seems that (1) is a disadvantage for subsampled natural gradients, while (2) is an advantage. However, we will see that this is not the case.

3.2 Experiments

The first set of experiments is conducted on fully connected networks on Fashion MNIST [Xiao et al., 2017] and MNIST [LeCun, 1998]. To demonstrate that our results apply to more complex settings, we will later also show results on CIFAR10 [Krizhevsky, 2009] with a Wide ResNet [He et al., 2016]. All results are averaged across three random seeds. All methods use a constant learning rate, and, if applicable, a constant damping term λ. Following [Martens and Grosse, 2015], the Fisher is approximated by sampling one label for each input, unless noted otherwise. In cases where we want to distinguish whether one label is sampled or whether the full Fisher is computed, we will refer to the former as **MC Fisher** and the latter as **Full Fisher**. We carry out careful and fair hyperparameter tuning and emphasise that, while our results are surprising, they are certainly not artefacts of inadequate hyperparameter choices. Details in Appendix D.

Wall-Clock Time for Natural Gradients: We first measure the wall-clock time of our subsampled natural gradient algorithm, see Figure 1A. The amortised version, while performing equally to the unamortised version, requires less than double the time of vanilla SGD. This shows that natural gradients can be computed efficiently, also in networks with several million parameters. The algorithm requires inverting a matrix of dimensions D, where D is the number of datapoints used to estimate the Fisher. In practice, the cost of this inversion is dominated by other computa-

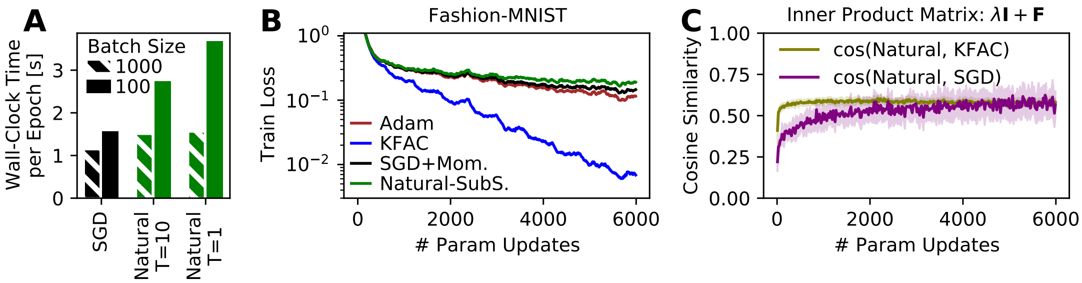

Figure 1: **KFAC, an approximate natural gradient method, significantly outperforms exact natural gradients and is barely more aligned with exact natural gradients than vanilla SGD. (A)** Wall-clock time comparison between SGD and our new, efficient implementation of Natural Gradients. T denotes how frequently the inverse Fisher is computed (implicitly). Implemented with PyTorch and run on a GPU with optimized DataLoader. **(B)** Training loss Fashion MNIST. Perhaps surprisingly, KFAC, an approximate natural gradient method, reaches a loss that is more than 10 times lower than its exact, subsampled counterpart. **(C)** Cosine Similarity between Exact Natural Gradients and KFAC, using identical gradient estimates and identical data samples to estimate the curvature. Inner product space given by curvature matrix, see Appendix J for euclidean inner product and analogous results on MNIST.

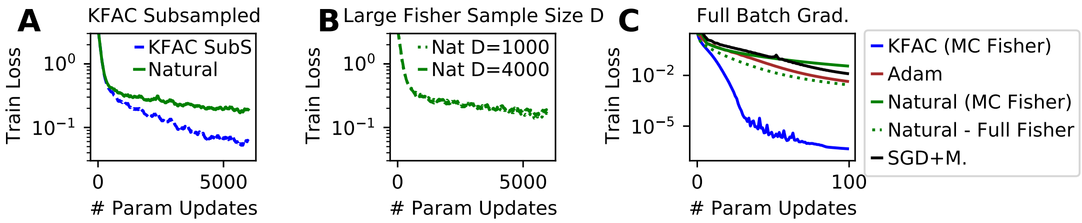

Figure 2: **Advantage of KFAC is not due to using more data to estimate the curvature. (A)** Comparison between Natural Gradients and a version of KFAC, which both use the same amount of data to estimate the Fisher. Very surprisingly, KFAC's advantage persists. **(B)** Versions of subsampled natural gradients which use more data to estimate Fisher (but batch-size for stochastic gradients is kept fixed at 100 for a fair and clean comparison). Increasing sample size does not improve performance. **(C)** Like (A), but training set is restricted to a subset of 1000 images and full-batch gradient descent is performed. Still, KFAC outperforms natural gradients.

tions even for comparatively large batch sizes of 1000 (Figure 1A).

Alignment of KFAC with Exact Second-Order Updates: Next, we measure how aligned the KFAC approximation is with exact natural gradients, see Figure 1C. Both algorithms compute second-order updates using identical gradient estimates and identical datapoints to estimate the curvature. They both use the damping parameter which gives optimal performance for KFAC. The inner product is given by the local (exact) curvature (results for the euclidean inner product in Appendix J). Somewhat surprisingly, KFAC's updates are barely more aligned with exact second-order updates than SGD.

Performance: In addition, we investigate the performance of KFAC and subsampled natural gradients, see Figure 1B. Surprisingly, natural gradients significantly underperform KFAC, which reaches an approximately 10-20x lower loss on both Fashion MNIST and MNIST. This is a concerning finding, requiring further investigation: After all, the exact natural gradient method should in theory perform at least as good as any approximation of it. Theoretically, the only potential advantage of KFAC is that it uses more information to estimate the curvature.

Controlling for Amount of Data used for the Curvature: The above directly leads to the hypothesis that KFAC's advantage over subsampled natural gradients is due to using more data for its approximation of the Fisher. To test this, we perform three experiments. (1) We explicitly restrict KFAC to use the same amount of data to estimate the curvature as the subsampled method. (2) We allow the subsampled method to use larger mini-batches to estimate the Fisher. (3) We restrict the training set to 1000 (randomly chosen) images and perform full batch gradient descent, again with both KFAC and subsampled natural gradients using the same amount of data to

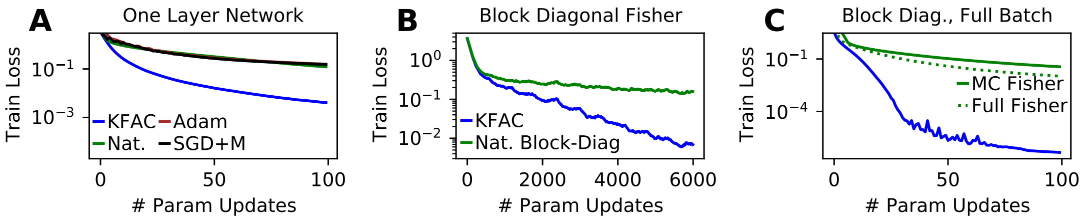

Figure 3: **Advantage of KFAC is not due to block-diagonal structure. (A)** A one layer network is trained on 1000 images and full batch gradients are used. In a one layer network the block-diagonal Fisher coincides with the full Fisher, but KFAC still clearly outperforms natural gradients. **(B)** Comparison between KFAC and layerwise (i.e. block-diagonal) subsampled Natural Gradients on full dataset with a three layer network. **(C)** Same as (B), but training set is restricted to a subset of 1000 images and full-batch gradient descent is performed.

estimate the Fisher. Here, we also include the Full Fisher information as computed on the 1000 training samples, rather than simply sampling one label per datapoint. In particular, we evaluate exact natural gradients (without any disclaimers: The gradient is exact, the Fisher is exact and the inversion is exact). The results are shown in Figure 2 and all lead to the same conclusion: The fact that KFAC uses more data than subsampled natural gradients does not explain its better performance. In particular, subsampled KFAC outperforms exact natural gradients.

Controlling for Block-Diagonal Structure: This begs further investigation into why KFAC outperforms natural gradients. There are two approximations that KFAC makes to the Fisher. (1) It approximates the Fisher as block-diagonal. (2) It approximates each diagonal block as a Kronecker-product. To test whether (1) explains KFAC's performance, we conduct two experiments. First, we train a one layer network on a subset of 1000 images with full-batch gradient descent, i.e. we perform logistic regression. In this case, the block-diagonal Fisher coincides with the Fisher. So, if the block-diagonal approximation were responsible for KFAC's performance, then for the logistic regression case, natural gradients should perform as well as KFAC or better. However, this is not the case as shown in Figure 3A. As an additional experiment, we consider a three layer network and approximate the Fisher by its block-diagonal (but without approximating blocks as Kronecker-products). The resulting computations and inversions can be carried out efficiently akin to our subsampled natural gradient method. We run the layer-wise natural gradient algorithm in two settings: In a mini-batch setting, identical to the one shown in Figure 1 and in a full-batch setting, by restricting to a subset of 1000 training images. The results in Figure 3B,C confirm the finding from our fprevious findings: (1) KFAC significantly outperforms even exact block-diagonal natural gradients (with full Fisher and full gradients). (2) It is not the block-diagonal structure that explains KFAC's performance.

Summary. All evidence points to one unexpected conclusion: KFAC's Kronecker-factors break any strong link to second-order information, but they – somehow – lead to very strong performance. We now turn to why this is the case.

4 First-order Descent on Neurons

We first describe a new optimizer called "Fast First-Order Optimizer" or "FOOF"[1] and then explain KFAC's link to it. The underlying view on optimization is principled and new. The resulting update equations are similar to some prior work, see Appendix H.2.

Recall our notation for one layer of a neural network from Section 2.1, namely $\mathbf{A}, \mathbf{W}$ for input activation and weight matrix as well as $\mathbf{B} = \mathbf{W}\mathbf{A}$ and $\mathbf{E} = \frac{\partial L}{\partial B}$.

Typically, for first-order optimizers, we compute the weights' gradients for each datapoint and average the results. Changing perspective, we can try to find an update of the weight matrix that explicitly changes the layer's outputs $\mathbf{B}$ into their gradient direction $\mathbf{E} = \frac{\partial L}{\partial \mathbf{B}}$. In other words, we want to find a weight update $\Delta\mathbf{W}$ to the parameters $\mathbf{W}$, so that the layer's output changes in the gradient direction, i.e. $(\mathbf{W} + \Delta\mathbf{W})\mathbf{A} = \mathbf{B} + \eta\frac{\partial L}{\partial \mathbf{B}}$ or equivalently $(\Delta\mathbf{W})\mathbf{A} = \eta\mathbf{E}$ for a learning rate η. Formally, we optimize

$$\min_{\Delta\mathbf{W}\in\mathbb{R}^{n\times m}} \|(\Delta\mathbf{W})\mathbf{A} - \eta\mathbf{E}\|^2 + \frac{\lambda}{2}\|\Delta\mathbf{W}\|^2 \qquad (3)$$

where the second summand $\frac{\lambda}{2}\|\Delta\mathbf{W}\|^2$ is a proximity constraint limiting the update size. (3) is a linear regression problem (for each row of $\Delta\mathbf{W}$) solved by

$$(\Delta\mathbf{W})^T = \eta\left(\lambda\mathbf{I} + \mathbf{A}\mathbf{A}^T\right)^{-1}\mathbf{A}\mathbf{E}^T \qquad (4)$$

See Figures 5,6,7 for empirical results of this optimizer, which outperforms not only SGD and Adam, but also KFAC. An intuition for why this optimizer performs

[1] F_2O_2 is a chemical also referred to as "FOOF".

considerably better than standard first-order optimizers is that it trades off conflicting gradients from different data points more effectively than the simple averaging scheme of SGD, see Appendix F for an illustrative toy example for this intuition.

The FOOF udpate can also be seen as preconditioning by $\left((\lambda \mathbf{I} + \mathbf{A}\mathbf{A}^T) \otimes \mathbf{I}\right)^{-1}$ and we emphasise that this preconditioning matrix contains no first-order, let alone second-order, information.

4.1 Stochastic Version of FOOF and Amortisation of Matrix Inversion

The above formulation is implicitly based on full-batch gradients. To apply it in a stochastic setting, we need to take some care to limit the bias of our updates. In particular, for the updates to be completely unbiased one would need to compute $\mathbf{A}\mathbf{A}^T$ for the entire dataset and invert the corresponding matrix at each iteration. This is of course too costly and instead we keep an exponentially moving average of mini-batch estimates of $\mathbf{A}\mathbf{A}^T$, which are computed during the standard forward pass. To amortise the cost of inverting this matrix, we only perform the inversion every T iterations. In principle, this leads to slightly stale values of the inverse, but in practice the algorithm is remarkably robust and allows choosing large values of T as also shown in Figure 5.

4.2 KFAC as First-Order Descent on Neurons: Theory and Experiments

Recall that the KFAC update is given by

$$(\Delta \mathbf{W})^T = \left(\mathbf{A}\mathbf{A}^T + \lambda_A \mathbf{I}\right)^{-1} \left(\mathbf{A}\mathbf{E}^T\right) \left(\mathbf{E}_F \mathbf{E}_F^T + \lambda_E \mathbf{I}\right)^{-1}.$$

Heurisitc Damping: We emphasise that the damping performed here is heuristic: Every Kronecker-factor is damped individually. This deviates from the theoretically "correct" form of damping, which consists of adding a multiple of the identity to the approximate curvature and which corresponds to damping the entire product rather than individual factors. [Martens and Grosse, 2015, Ba et al., 2016, George et al., 2018] all explicitly note that this heuristic works better than standard damping, a somewhat mysterious observation which we will reproduce and come back to later.

Similarity of KFAC to FOOF and Damping: The update of KFAC (eq (2)) differs from the FOOF update (eq (4)) only through the second factor $\left(\mathbf{E}_F \mathbf{E}_F^T + \lambda_E \mathbf{I}\right)^{-1}$. Without heuristic damping, this second factor could of course lead to updates of KFAC that are essentially uncorrelated with FOOF. However, as we use heuristic damping and increase the damping strength, the second factor will be closer to (a multiple of) the identity and KFAC's update will become more and more aligned with FOOF. Based on this derivation, we now test empirically whether heuristic damping indeed makes KFAC similar to FOOF and how it affects performance:
Figure 4A shows that heuristic damping is essential for KFACs performance; with correct rather than heuristic damping KFACs performance is similar to SGD+M. Figure 4B confirms our theoretical argument that heuristic damping drastically increases similarity of KFAC to FOOF and stronger heuristic damping leads to even stronger similarity. This similarity is directly linked to performance of KFAC as shown in Figure 4C. Also importantly, as shown in Figure 4D, FOOF requires lower damping than KFAC to perform well. This strongly suggests that damping in KFAC does not only restrict update size, but is strictly required to limit the effect of $\mathbf{E}_F \mathbf{E}_F^T$ on the update, thus increasing similarity to FOOF. These results indicate that increasing similarity of KFAC to natural gradients hurts performance, while increasing similarity to FOOF increases performance. This strongly suggest that KFAC, rather than being a second-order method, owes its performance to approximating FOOF.

Performance: If the above view of KFAC is correct, and it owes its performance to similarity to FOOF, then one would expect FOOF to perform even better and more stably than KFAC. This is indeed the case as shown in Figure 5.

Computational Cost: We also note that, on top of making more progress per parameter update, FOOF requires strictly less computation than KFAC: It does not require an additional backward pass to estimate the Fisher; it only requires keeping track of, inverting as well as multiplying the gradients by one matrix rather than two (only $\mathbf{A}\mathbf{A}^T$ and not $\mathbf{E}_F \mathbf{E}_F^T$). These savings lead to a 1.5x speed-up in wall-clock time per-update for the amortised versions of KFAC and FOOF as shown in Figure 5A.

Summary: We had already seen that KFAC does not rely on second-order information. In addition, the above results strongly suggest that KFAC owes its strong performance to its similarity to FOOF, which is a principled, well-performing first-order optimizer.

CIFAR10: Finally, to test whether our insights carry over to more complex experimental set ups, we perform experiments on CIFAR10 with a wide ResNet18 trained for 50 epochs. Following [Grosse and Martens, 2016, Ba et al., 2016], we equip KFAC and FOOF with momentum and report results with and without a form of polyak averaging [Polyak and Juditsky, 1992], which

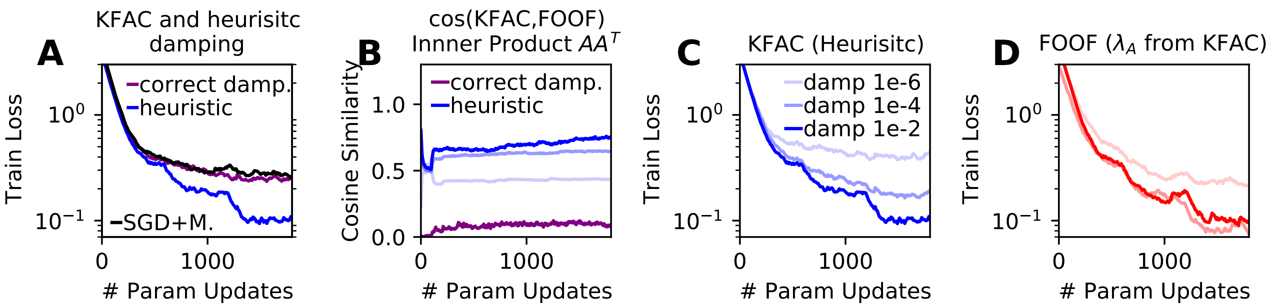

Figure 4: **Damping increases KFAC's performance as well as its similarity to first-order method FOOF. (A)** Heuristic Damping is strictly needed for performance of KFAC; with standard damping, KFAC performs similar to SGD. **(B)** Heuristic Damping significantly increases similarity of KFAC to FOOF. **(C+D)** Performance of KFAC and FOOF across different damping strengths using heuristic damping for KFAC. For a clean and fair comparison, this version of FOOF uses λ_A from KFAC, see Appendix D.6. Notably, FOOF already works well for lower damping terms than KFAC, suggesting that KFAC requires larger damping mainly to guarantee similarity to FOOF. (A-D) x-axis cut after 3 epochs, since KFAC sometimes becomes unstable later. **(A-D)** and our theoretical analysis suggest that KFAC owes its performance to similarity to the first-order method FOOF.

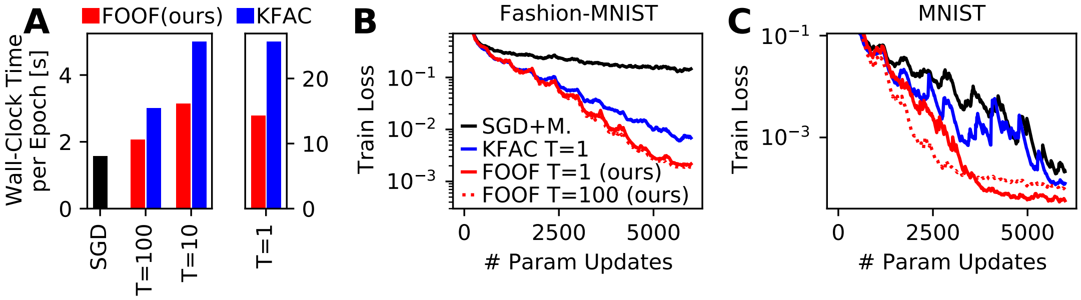

Figure 5: **FOOF outperforms KFAC in terms of both per-update progress and computation cost. (A)** Wall-clock time comparison between FOOF, KFAC and SGD. T denotes how frequently matrix inversions (see eq (4)) are performed. Implemented with PyTorch and run on a GPU (with optimized DataLoader). Increasing T above 100 does not notably improve runtime. FOOF is approximately 1.5x faster than KFAC. **(B, C)** Training loss on Fashion MNIST and MNIST. FOOF is more data efficient and stable than KFAC.

keeps an exponentially moving average of iterates, see [Grosse and Martens, 2016] for details. Since KFAC+momentum was unstable, consistent with previous findings [Grosse and Martens, 2016, Ba et al., 2016], we only switched on momentum after one epoch. While other methods did not require this adaptation for stability, it helped their performance a bit, so that we applied it to all methods.

Computation Time in Convolutional Architectures: The only overhead of KFAC and FOOF which cannot be amortised is performing the matrix multiplications in eqs (4),(2). These are standard matrix-matrix multiplications and are considerably cheaper than convolutions, so that we found that KFAC and FOOF can be amortised to have almost the same wall-clock time per update as SGD for this experiment (∼10% increase for FOOF, ∼15% increase for KFAC) without sacrificing performance, see Appendix D. We note that these results are significantly better than wall-clock times from [Ba et al., 2016, Desjardins et al., 2015], which require approximately twice as much time per update as SGD.[2]

Performance: We show results for SGD+Momentum, KFAC and FOOF in Figures 6,7. We also experimented with Adam and Subsampled Natural Gradients, but found that they performed essentially the same as SGD, as already seen in Figure 5. The results for SGD, KFAC and FOOF are consistent with our previous findings and show that FOOF further improves performance over KFAC.

5 Discussion

One of our main contributions is a new, fundamentally different explanation for why KFAC is such an effective optimizer. To see how this explanation im-

[2] Information reconstructed from Figure 3 in [Ba et al., 2016] and Figure 4 in [Desjardins et al., 2015].

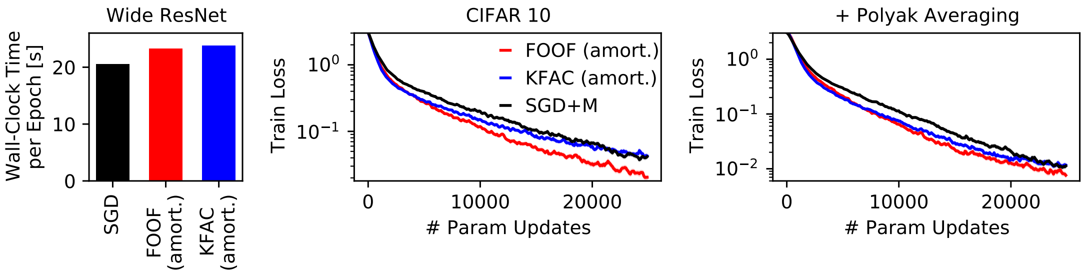

Figure 6: **FOOF outperforms KFAC in a Wide ResNet18 on CIFAR 10. (A)** Wall-clock time comparison between FOOF, KFAC and SGD. In convolutional architectures, FOOF and KFAC can be effectively amortised wihtout sacrificing performance, Appendix D **(B, C)** Training loss with and without a form of polyak averaging.

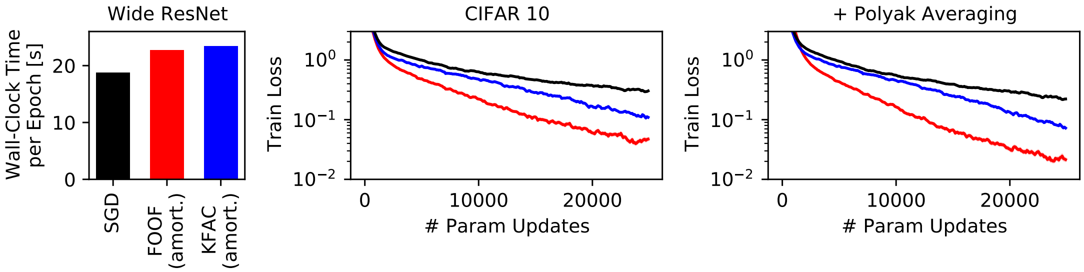

Figure 7: **Same as Figure 6 but without batch norm.** All methods require slightly less wall-clock time without BN and all methods benefit from BN for optimization performance. For KFAC, we report only the best run due to instabilities. For SGD and FOOF we stick to reporting the average across three seeds. KFAC instabilities could probably be mitigated by adaptive clipping from [Grosse and Martens, 2016, Ba et al., 2016], requiring slightly more computations and additional hyperparameters.

proves upon prior knowledge, we summarise how well it explains experimental findings compared to the traditional second-order view.

Damping: The second-order view prescribes damping the entire curvature matrix. In particular, this should work better than approximate damping, like that of KFAC, which damps individual Kronecker-factors rather than the entire curvature. The original KFAC paper [Martens and Grosse, 2015] as well as [Ba et al., 2016, George et al., 2018] report this heuristic damping performs better than standard second-order damping. We reproduced these results, finding that with ”correct” second-order damping, KFACs performance becomes very simliar to SGD. These findings are inconsistent with the second-order view of KFAC. In contrast, the first-order view shows that non-standard damping makes KFAC more similar to gradient descent on neurons and thus should make performance similar to FOOF. This is supported by Figure 4 and offers a sound explanation for KFAC's performance.

Ignoring the second Kronecker-factor $\mathbf{E}_F\mathbf{E}_F^T$ of the approximate Fisher: The first-order algorithm FOOF can be seen as dropping the second Kronecker-factor of KFAC (eq (2)). The first-order view predicts that this helps performance and is confirmed by Figures 5,6,7. From the second-order viewpoint, dropping the second factor is highly questionable: It leads to approximating second-order information by a zeroth order matrix $\mathbf{A}\mathbf{A}^T$. Thus, it should notably harm performance. This prediction is inconsistent with Figures 5,6,7, and thus contradicts the second-order interpretation.

Architectures with Parameter Sharing: Approximating the block-diagonal Fisher by a Kronecker-product in architectures with shared parameters, e.g. CNNs or RNNs, requires additional, sometimes rather complex assumptions [Grosse and Martens, 2016, Martens et al., 2018]. From the second-order view, it is at least somewhat surprising that KFAC works rather well in all these instances, despite the fact that the additional assumptions are not always true. From the first-order viewpoint this is not surprising at all. The KFAC approximations still lead to performing gradient descent on neurons and we would expect this to be as effecitve as before[3].

Exact Natural Gradients vs KFAC vs FOOF: Let us compare what both theories predict for the

[3] In eq (3), we now impose one constraint per datapoint and per “location” at which the parameters are applied.

relative performance of Exact Natural Gradients vs KFAC vs FOOF. The first-order view is agnostic on how FOOF performs relative to natural gradients; additionally it predicts that FOOF performs similar to or better than K-FAC, which is consistent with our results. The traditional second-order view predicts that exact natural gradient perform at least as good as any approximation of it – at least after controlling for how much data is used to estimate the curvature matrix, as we have done. Anything else – in particular our findings that KFAC outperforms natural gradients in standard as well as many control settings – has do be seen as a strong, if not definite, contradiction to the second-order view.

6 Limitations

In our experiments, we report training losses and tune hyperparameters with respect to them. While this is the correct way to test our hypotheses and common for developing and testing optimizers [Sutskever et al., 2013], it will be important to test how well the optimizers investigated here generalise. A meaningful investigation of generalisation requires very different experiments than the ones conducted here (as demonstrated in [Zhang et al., 2018b]) and is left to future work.

We have restricted our investigation and the application of our new tool to the context of optimization and more specifically KFAC. While we strongly believe that our findings carry over to other Kronecker-factored optimizers [Desjardins et al., 2015, Goldfarb et al., 2020, Botev et al., 2017, George et al., 2018, Martens et al., 2018, Osawa et al., 2019], we have not explicitly tested this.

Similarly, in Appendix G, we present a fundamentally different view on the effectiveness of KFAC for posterior approximations [Ritter et al., 2018b, Wu et al., 2017, Ritter et al., 2018a, Grant et al., 2018, Zhang et al., 2018a]. We leave testing this hypothesis to future work and note that findings from [Ober and Aitchison, 2021] support our view.

7 Conclusion

We have shown that subsampled natural gradients are efficiently computable in large neural networks. The underlying techniques can also be used to compute subsampled Generalised Gauss-Newton updates efficiently. Moreover, they have potential applications in continual learning, meta learning and for Laplace posteriors, as discussed more thoroughly in Appendix C.

Moreover, through a series of careful experiments, we used our subsampled gradient method to show that KFAC is not closely related to second-order information and we proposed a fundamentally different explanation for why it is one of the best known optimizers for neural networks. This also is an important step to understanding why KFAC works so well in a range of different contexts and may well lead to further improvements in these areas [Grosse and Salakhudinov, 2015, Ba et al., 2016, Wu et al., 2017, Goldfarb et al., 2020, Botev et al., 2017, Desjardins et al., 2015, George et al., 2018, Martens et al., 2018, Osawa et al., 2019, Ritter et al., 2018b, Ritter et al., 2018a, Dangel et al., 2020, Zhang et al., 2018a, Grant et al., 2018].

Finally, we showed that performing gradient descent on neurons rather than weights is unreasonably effective and improves convergence even compared to state-of-the art methods like KFAC.

References

[Agarwal et al., 2019] Agarwal, N., Bullins, B., Chen, X., Hazan, E., Singh, K., Zhang, C., and Zhang, Y. (2019). Efficient full-matrix adaptive regularization. In *International Conference on Machine Learning*, pages 102–110. PMLR.

[Aitchison, 2018] Aitchison, L. (2018). Bayesian filtering unifies adaptive and non-adaptive neural network optimization methods. *arXiv preprint arXiv:1807.07540*.

[Amari, 1998] Amari, S.-I. (1998). Natural gradient works efficiently in learning. *Neural computation*, 10(2):251–276.

[Amari and Nagaoka, 2000] Amari, S.-i. and Nagaoka, H. (2000). *Methods of information geometry*, volume 191. American Mathematical Soc.

[Amid et al., 2021] Amid, E., Anil, R., and Warmuth, M. K. (2021). Locoprop: Enhancing backprop via local loss optimization. *arXiv preprint arXiv:2106.06199*.

[Ba et al., 2016] Ba, J., Grosse, R., and Martens, J. (2016). Distributed second-order optimization using kronecker-factored approximations.

[Bengio, 2000] Bengio, Y. (2000). Gradient-based optimization of hyperparameters. *Neural computation*, 12(8):1889–1900.

[Benzing, 2020] Benzing, F. (2020). Unifying regularisation methods for continual learning. *arXiv preprint arXiv:2006.06357*.

[Bernacchia et al., 2019] Bernacchia, A., Lengyel, M., and Hennequin, G. (2019). Exact natural gradient in deep linear networks and application to the nonlinear case. NIPS.

[Blundell et al., 2015] Blundell, C., Cornebise, J., Kavukcuoglu, K., and Wierstra, D. (2015). Weight uncertainty in neural network. In *International Conference on Machine Learning*, pages 1613–1622. PMLR.

[Botev et al., 2017] Botev, A., Ritter, H., and Barber, D. (2017). Practical gauss-newton optimisation for deep learning. In *International Conference on Machine Learning*, pages 557–565. PMLR.

[Dangel et al., 2020] Dangel, F., Harmeling, S., and Hennig, P. (2020). Modular block-diagonal curvature approximations for feedforward architectures. In *International Conference on Artificial Intelligence and Statistics*, pages 799–808. PMLR.

[Dangel et al., 2021] Dangel, F., Tatzel, L., and Hennig, P. (2021). Vivit: Curvature access through the generalized gauss-newton's low-rank structure. *arXiv preprint arXiv:2106.02624*.

[Dauphin et al., 2014] Dauphin, Y., Pascanu, R., Gulcehre, C., Cho, K., Ganguli, S., and Bengio, Y. (2014). Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. *arXiv preprint arXiv:1406.2572*.

[Daxberger et al., 2021] Daxberger, E., Kristiadi, A., Immer, A., Eschenhagen, R., Bauer, M., and Hennig, P. (2021). Laplace redux–effortless bayesian deep learning. *arXiv preprint arXiv:2106.14806*.

[Desjardins et al., 2015] Desjardins, G., Simonyan, K., Pascanu, R., and Kavukcuoglu, K. (2015). Natural neural networks. *arXiv preprint arXiv:1507.00210*.

[Doucet, 2010] Doucet, A. (2010). A Note on Efficient Conditional Simulation of Gaussian Distributions. [Online; accessed 17-September-2021].

[Frerix et al., 2017] Frerix, T., Möllenhoff, T., Moeller, M., and Cremers, D. (2017). Proximal backpropagation. *arXiv preprint arXiv:1706.04638*.

[George et al., 2018] George, T., Laurent, C., Bouthillier, X., Ballas, N., and Vincent, P. (2018). Fast approximate natural gradient descent in a kronecker-factored eigenbasis. *arXiv preprint arXiv:1806.03884*.

[Glorot and Bengio, 2010] Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256. JMLR Workshop and Conference Proceedings.

[Goldfarb et al., 2020] Goldfarb, D., Ren, Y., and Bahamou, A. (2020). Practical quasi-newton methods for training deep neural networks. *arXiv preprint arXiv:2006.08877*.

[Grant et al., 2018] Grant, E., Finn, C., Levine, S., Darrell, T., and Griffiths, T. (2018). Recasting gradient-based meta-learning as hierarchical bayes. *arXiv preprint arXiv:1801.08930*.

[Graves, 2011] Graves, A. (2011). Practical variational inference for neural networks. *Advances in neural information processing systems*, 24.

[Grosse and Martens, 2016] Grosse, R. and Martens, J. (2016). A kronecker-factored approximate fisher matrix for convolution layers. In *International Conference on Machine Learning*, pages 573–582. PMLR.

[Grosse and Salakhudinov, 2015] Grosse, R. and Salakhudinov, R. (2015). Scaling up natural gradient by sparsely factorizing the inverse fisher matrix. In *International Conference on Machine Learning*, pages 2304–2313. PMLR.

[He et al., 2015] He, K., Zhang, X., Ren, S., and Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034.

[He et al., 2016] He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778.

[Hochreiter and Schmidhuber, 1997] Hochreiter, S. and Schmidhuber, J. (1997). Flat minima. *Neural computation*, 9(1):1–42.

[Hoffman and Ribak, 1991] Hoffman, Y. and Ribak, E. (1991). Constrained realizations of gaussian fields-a simple algorithm. *The Astrophysical Journal*, 380:L5–L8.

[Immer et al., 2021] Immer, A., Bauer, M., Fortuin, V., Rätsch, G., and Khan, M. E. (2021). Scalable marginal likelihood estimation for model selection in deep learning. *arXiv preprint arXiv:2104.04975*.

[Ioffe and Szegedy, 2015] Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In

International conference on machine learning, pages 448–456. PMLR.

[Karakida et al., 2021] Karakida, R., Akaho, S., and Amari, S.-i. (2021). Pathological spectra of the fisher information metric and its variants in deep neural networks. *Neural Computation*, 33(8):2274–2307.

[Kingma and Ba, 2014] Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980.*

[Kirkpatrick et al., 2017] Kirkpatrick, J., Pascanu, R., Rabinowitz, N., Veness, J., Desjardins, G., Rusu, A. A., Milan, K., Quan, J., Ramalho, T., Grabska-Barwinska, A., et al. (2017). Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences*, 114(13):3521–3526.

[Krizhevsky, 2009] Krizhevsky, A. (2009). Learning multiple layers of features from tiny images. Technical report.

[Kunstner et al., 2019] Kunstner, F., Balles, L., and Hennig, P. (2019). Limitations of the empirical fisher approximation for natural gradient descent. *arXiv preprint arXiv:1905.12558.*

[LeCun, 1998] LeCun, Y. (1998). The mnist database of handwritten digits. *http://yann. lecun. com/exdb/mnist/.*

[LeCun et al., 1990] LeCun, Y., Denker, J. S., and Solla, S. A. (1990). Optimal brain damage. In *Advances in neural information processing systems*, pages 598–605.

[MacKay, 1992] MacKay, D. J. (1992). A practical bayesian framework for backpropagation networks. *Neural computation*, 4(3):448–472.

[Marceau-Caron and Ollivier, 2016] Marceau-Caron, G. and Ollivier, Y. (2016). Practical riemannian neural networks. *arXiv preprint arXiv:1602.08007.*

[Martens, 2014] Martens, J. (2014). New insights and perspectives on the natural gradient method. *arXiv preprint arXiv:1412.1193.*

[Martens et al., 2018] Martens, J., Ba, J., and Johnson, M. (2018). Kronecker-factored curvature approximations for recurrent neural networks. In *International Conference on Learning Representations.*

[Martens et al., 2021] Martens, J., Ballard, A., Desjardins, G., Swirszcz, G., Dalibard, V., Sohl-Dickstein, J., and Schoenholz, S. S. (2021). Rapid training of deep neural networks without skip connections or normalization layers using deep kernel shaping. *arXiv preprint arXiv:2110.01765.*

[Martens et al., 2010] Martens, J. et al. (2010). Deep learning via hessian-free optimization. In *ICML*, volume 27, pages 735–742.

[Martens and Grosse, 2015] Martens, J. and Grosse, R. (2015). Optimizing neural networks with kronecker-factored approximate curvature. In *International conference on machine learning*, pages 2408–2417. PMLR.

[Nguyen et al., 2017] Nguyen, C. V., Li, Y., Bui, T. D., and Turner, R. E. (2017). Variational continual learning. *arXiv preprint arXiv:1710.10628.*

[Ober and Aitchison, 2021] Ober, S. W. and Aitchison, L. (2021). Global inducing point variational posteriors for bayesian neural networks and deep gaussian processes. In *International Conference on Machine Learning*, pages 8248–8259. PMLR.

[Ollivier, 2015] Ollivier, Y. (2015). Riemannian metrics for neural networks i: feedforward networks. *Information and Inference: A Journal of the IMA*, 4(2):108–153.

[Ollivier, 2017] Ollivier, Y. (2017). True asymptotic natural gradient optimization. *arXiv preprint arXiv:1712.08449.*

[Ollivier, 2018] Ollivier, Y. (2018). Online natural gradient as a kalman filter. *Electronic Journal of Statistics*, 12(2):2930–2961.

[Osawa et al., 2019] Osawa, K., Tsuji, Y., Ueno, Y., Naruse, A., Yokota, R., and Matsuoka, S. (2019). Large-scale distributed second-order optimization using kronecker-factored approximate curvature for deep convolutional neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 12359–12367.

[Pascanu and Bengio, 2013] Pascanu, R. and Bengio, Y. (2013). Revisiting natural gradient for deep networks. *arXiv preprint arXiv:1301.3584.*

[Paszke et al., 2019] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. (2019). Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32:8026–8037.

[Polyak and Juditsky, 1992] Polyak, B. T. and Juditsky, A. B. (1992). Acceleration of stochastic approximation by averaging. *SIAM journal on control and optimization*, 30(4):838–855.

[Ren and Goldfarb, 2019] Ren, Y. and Goldfarb, D. (2019). Efficient subsampled gauss-newton and natural gradient methods for training neural networks. *arXiv preprint arXiv:1906.02353*.

[Ritter et al., 2018a] Ritter, H., Botev, A., and Barber, D. (2018a). Online structured laplace approximations for overcoming catastrophic forgetting. *arXiv preprint arXiv:1805.07810*.

[Ritter et al., 2018b] Ritter, H., Botev, A., and Barber, D. (2018b). A scalable laplace approximation for neural networks. In *6th International Conference on Learning Representations, ICLR 2018-Conference Track Proceedings*, volume 6. International Conference on Representation Learning.

[Roux et al., 2007] Roux, N., Manzagol, P.-a., and Bengio, Y. (2007). Topmoumoute online natural gradient algorithm. *Advances in Neural Information Processing Systems*, 20:849–856.

[Sutskever et al., 2013] Sutskever, I., Martens, J., Dahl, G., and Hinton, G. (2013). On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pages 1139–1147. PMLR.

[Wu et al., 2017] Wu, Y., Mansimov, E., Grosse, R. B., Liao, S., and Ba, J. (2017). Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation. *Advances in neural information processing systems*, 30:5279–5288.

[Xiao et al., 2017] Xiao, H., Rasul, K., and Vollgraf, R. (2017). Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms.

[Zhang et al., 2019] Zhang, G., Martens, J., and Grosse, R. (2019). Fast convergence of natural gradient descent for overparameterized neural networks. *arXiv preprint arXiv:1905.10961*.

[Zhang et al., 2018a] Zhang, G., Sun, S., Duvenaud, D., and Grosse, R. (2018a). Noisy natural gradient as variational inference. In *International Conference on Machine Learning*, pages 5852–5861. PMLR.

[Zhang et al., 2018b] Zhang, G., Wang, C., Xu, B., and Grosse, R. (2018b). Three mechanisms of weight decay regularization. *arXiv preprint arXiv:1810.12281*.

APPENDIX

A Tuning Natural Gradients

While, for the sake of fairness, all results in the paper are based on the hyperparameter optimization scheme described further below, we emphasise that we also invested effort into hand-tuning subsampled natural gradient methods, but that this did not give notably better results than the ones reported in the paper. We also implemented and tested an adaptive damping scheme as described in [Martens and Grosse, 2015], tried combining it with automatic step size selection and also a form of momentum, all as described in [Martens and Grosse, 2015]. None of these techniques gave notable improvements.

B Software Validation

To validate that our algorithm of computing products between the damped, inverse Fisher and vectors (as described in Appendix I) is correct, we considered small networks in which we could explicitly compute and invert the Fisher Information and confirmed that our implicit calculations agree with the explicit calculations for both fully connected as well as convolutional architectures.

C Further Applications of Implicit, Fast Fisher Inversion

C.1 Bayesian Laplace Posterior Approximation

Laplace Approximations are a common approximation to posterior weight distributions and various techniques have been proposed to approximate them, for a recent overview and evaluation we refer to [Daxberger et al., 2021]. In this context, the Hessian is the posterior precision matrix and it is often approximated by the Fisher or empirical Fisher, since these are positive semi-definite by construction.

It is often stated that sampling from a full covariance, Laplace posterior is computationally intractable. However, combining our insights with an additional trick allows us to sample from an exact posterior, given a subsampled

Fisher, as shown below. We adapted the trick from [Doucet, 2010], who credits [Hoffman and Ribak, 1991]. We also remark that [] sample from the predictive distribution of a linearised model, arguing theoretically and empirically that the linearised model is the more principled choice when approximating the Hessian by the Fisher.

C.1.1 Sampling from the full covariance Laplace posterior

We write $\mathbf{\Lambda}_{\text{prior}}$ for the prior precision and $\mathbf{\Lambda} = \mathbf{\Lambda}_{\text{prior}} + D\mathbf{F}$ for the posterior precision, where $\mathbf{F}$ is the Fisher. All expressions below can be evaluated efficiently for example if the prior precision is diagonal and constant across layers, as is usually the case. As for the natural gradients, we factorise $\mathbf{F} = \mathbf{G}\mathbf{G}^T$, where $\mathbf{G}$ is a $N \times D$ matrix (N is the number of parameters, D the number of datapoints). Using Woodbury's identity, we can write the posterior variance as

$$\mathbf{\Lambda}^{-1} = \left(\mathbf{\Lambda}_{\text{prior}} + D\mathbf{G}\mathbf{G}^T\right)^{-1} = \mathbf{\Lambda}_{\text{prior}}^{-1} - D\mathbf{\Lambda}_{\text{prior}}^{-1}\mathbf{G}\left(\mathbf{I} + D\mathbf{G}^T\mathbf{\Lambda}_{\text{prior}}^{-1}\mathbf{G}\right)^{-1}\mathbf{G}^T\mathbf{\Lambda}_{\text{prior}}^{-1} \tag{5}$$

We now show how to obtain a sample from this posterior. To this end, define matrices $\mathbf{V}, \mathbf{U}$ as follows:

$$\mathbf{V} = \left(D^{1/2}\mathbf{\Lambda}_{\text{prior}}^{-1/2}\right)\mathbf{G} \tag{6}$$

$$\mathbf{U} = \mathbf{I} + \mathbf{V}^T\mathbf{V} = \mathbf{I} + D\mathbf{G}^T\mathbf{\Lambda}_{\text{prior}}^{-1}\mathbf{G} \tag{7}$$

$$\tag{8}$$

Let $\mathbf{y} \sim \mathcal{N}\left(\mathbf{0}, \mathbf{I}_N\right)$ and $\mathbf{z} \sim \mathcal{N}\left(\mathbf{0}, \mathbf{I}_D\right)$, and define $\mathbf{x}$

$$\mathbf{x} = \mathbf{y} - \mathbf{V}\mathbf{U}^{-1}\left(\mathbf{V}^T\mathbf{y} + \mathbf{z}\right) \tag{9}$$

We will confirm by calculation that $\mathbf{\Lambda}_{\text{prior}}^{-1/2}\mathbf{x}$ is a sample from the full covariance posterior.

$\mathbf{x}$ clearly has zero mean. The covariance $\mathbb{E}\left[\mathbf{x}\mathbf{x}^T\right]$ can be computed as

$$\mathbb{E}\left[\mathbf{x}\mathbf{x}^T\right] = \mathbf{I} + \mathbf{V}\mathbf{U}^{-1}\left(\mathbf{V}^T\mathbf{V} + \mathbf{I}\right)\mathbf{U}^{-T}\mathbf{V}^T - 2\mathbf{V}\mathbf{V}^T \tag{10}$$

Since $\mathbf{U}$ is symmetric and since we chose $\mathbf{V}^T\mathbf{V} + \mathbf{I} = \mathbf{U}$, the above simplifies to

$$\mathbb{E}\left[\mathbf{x}\mathbf{x}^T\right] = \mathbf{I} - \mathbf{V}\mathbf{U}^{-1}\mathbf{V}^T \tag{11}$$

By our choice of $\mathbf{U}, \mathbf{V}$, this expression equals

$$\mathbb{E}\left[\mathbf{x}\mathbf{x}^T\right] = \mathbf{I} - D\mathbf{\Lambda}_{\text{prior}}^{-1/2}\mathbf{G}\left(\mathbf{I} + D\mathbf{G}^T\mathbf{\Lambda}_{\text{prior}}^{-1}\mathbf{G}\right)^{-1}\mathbf{G}^T\mathbf{\Lambda}_{\text{prior}}^{-1/2} \tag{12}$$

In other words, $\mathbf{\Lambda}_{\text{prior}}^{-1/2}\mathbf{x}$ is a sample form the full covariance posterior.

C.1.2 Efficient Evaluation of the above procedure

The computational bottlenecks are computing $\mathbf{G}^T\mathbf{\Lambda}_{\text{prior}}^{-1}\mathbf{G}^T$, calculating vector products with $\mathbf{G}$ and $\mathbf{G}^T$.

Note that for a subsampled Fisher with moderate D, we can invert $\mathbf{U}$ explicitly.

We have already encountered all these bottlenecks in the context of natural gradients and they can be solved efficiently in the same way, see Section I. The only modification is multiplication by $\mathbf{\Lambda}_{\text{prior}}^{-1}$ and for any diagonal prior, this can be solved easily.

C.2 Continual Learning

Closely related to bayesian posteriors is a number of continual learning algorithms [Kirkpatrick et al., 2017, Nguyen et al., 2017, Benzing, 2020]. For example, EWC [Kirkpatrick et al., 2017] relies on the Fisher to approximate posteriors. Formally, it only requires evaluating products of the form $\mathbf{v}^T\mathbf{F}\mathbf{v}$. Since the Fisher is large, EWC uses a diagonal approximation. From the exposition below, it is not to difficult to see that $\mathbf{v}^T\mathbf{F}\mathbf{v}$ is easy to evaluate for a subsampled Fisher and the memory cost is roughly equal to that used for a standard for- and backward pass through the model: It scales as the product of the number of datapoints and the number of neurons (rather than weights).

C.3 Meta Learning / Bilevel optimization

Some bilevel optimization algorithms rely on the Inverse Function Theorem to estimate outer-loop gradients after the inner loop has converged. They require evaluating a product of the form $(\lambda\mathbf{I}+\mathbf{H})^{-1}\mathbf{v}$, where $\mathbf{H}$ is the Hessian and $\mathbf{v}$ a vector, see e.g. [Bengio, 2000]. If we approximate the Hessian by a subsampled Fisher, our methods are directly applicable to compute this product.

D Experimental Details

All experiments were implemented in PyTorch [Paszke et al., 2019].[4] All models use Kaiming-He initialisation [He et al., 2015, Glorot and Bengio, 2010].

D.1 Exponentially Moving Averages and Subsampled KFAC

As mentioned in the main text, we use exponentially moving averages to estimate the matrices $\mathbf{A}\mathbf{A}^T$ and $\mathbf{E}_F\mathbf{E}_F^T$. For a quantity x_t the exponentially moving average $\hat{x}_t$ is defined as:

$$\hat{x}_{t+1} = m \cdot \hat{x}_t + (1-m)x_{t+1}$$

We also normalise our exponentially moving averages (or equivalently initialise $\hat{x}_1 = x_1$). Following [Martens and Grosse, 2015], we set $m = 0.95$. Preliminary experiments with $m = 0.999$ showed very similar performance.

For subsampled KFAC, we simply set $m = 0$, implying that KFAC, like Subsampled Natural Gradients, only uses one mini-batch to estimate the curvature (we made sure that both methods use exactly the same datapoints – this includes drawing the same samples from the model distribution). This is also what we used for Figure 1C along with the optimal hyperparameters for KFAC. For this experiment, we evaluate gradients and curvature on independent mini-batches, which is consistent with Section I.5 and analogous to “full” KFAC, which, due to the exponentially moving average, gives little weight to the current batch (the amortised version even gives no weight to the current mini-batch for the majority of updates).

D.2 Hyperparameter Tuning

Learning rates for all methods were tuned by a grid search, considering values of the form $1 \cdot 10^i, 3 \cdot 10^i$ for suitable (usually negative) integers i. The damping terms for KFAC and FOOF were determined by a grid searcher over $10^{-6}, 10^{-4}, 10^{-2}, 10^0, 10^2, 10^4, 10^6$. For CIFAR, we also experimented with a finer grid of the form 10^i, and this did not affect results. For SGD, momentum was grid-searched from $0.0, 0.9$. For Adam, we kept most hyperparameters fixed and tuned only learning rate.

Each ablation / experiment got its own hyperparameter search. The only exception is FOOF ($T = 100$), which uses exactly the same hyperparameters as FOOF ($T = 1$).

We always chose the hyperparameters which gives best training loss at the end of training. Usually, these hyperparameters also outperform others in the early training stage. We also note that several hyperparametrisations of KFAC became unstable towards the end/middle of training, while this did not occur for FOOF.

D.3 Hyperparameter Robustness

On top of the learning rate, FOOF requires additional hyperparameters, most of which seem very robust as described below. Additional hyperparameters always include damping and further may include the momentum for exponentially moving averages, and also may include amortisation hyperparameters S,T described in Algorithm 1.

- Stepsize: Tuning the stepsize of FOOF was as easy/difficult as tuning the stepsize of SGD in our experience. In particular, BatchNorm allows using a wider range of learning rates without large changes in performance. We recommend to parametrise the stepsize as α/λ (where λ is the damping term) and (grid-)search over α.

[4] Thanks for all the hooks.

- Damping λ: We searched from a grid of the form 10^{2i} for integers i and this was sufficient for our experimental setup. Refining this grid only gave very small improvements (at least for the experimental setup in our paper). Using values that differed by a factor of 100 from the optimal value, usually still gave good results and clearly outperformed SGD.
- Exponential Moving Average m for $\mathbf{A}\mathbf{A}^T$: Following [Martens and Grosse, 2015], we set this value to 0.95. Preliminary experiments with 0.999 gave similar performance. Results seem very robust with respect to this choice. It is conceivable that very small batch sizes require slightly larger values of m to estimate $\mathbf{A}\mathbf{A}^T$
- Amortisation parameters S, T: In our experiments it was sufficient to perform inversions once per epoch. Setting S to 50 is a mathematically save choice (given $m = 0.95$), and setting $S = 10$ empirically did not decrease performance either.

D.4 Remaining details

Experiments were carried out on fully connected networks, with 3 hidden layers of size 1000. The only exception is the network in Figure 3A, which has no hidden layer. For simplicity of implementation, we omitted biases. Unless noted otherwise, we trained networks for 10 epochs which batch size 100 on MNIST or Fashion MNIST. MNIST was preprocessed to have zero mean and unit variance, and – due to an oversight – Fashion MNIST was preprocessed in the same way, using the mean+std-dev of MNIST. Our comparisons remain meaningful, as this affects all methods equally and perhaps it even makes our results more comparable to prior work. Several experiments were carried out on subsets of the training set consisting of 1000 images, in order to make full batch gradient evaluation cheaper and were trained for 100 epochs.

For the wall-clock time experiments, we used PyTorch DataLoaders and optimized the "pin_memory" and "num_workers" arguments for each method/setting.

D.5 CIFAR 10

For CIFAR10 we use a Wide ResNet18 and standard data-augmentation consisting of random horizontal flips as well as padding with 4 pixels followed by random cropping. The RGB channels are normalised to have zero mean and unit variance.

The SGD baseline uses standard batch norm [Ioffe and Szegedy, 2015]. For FOOF and KFAC we include batch norm parameters, but do not train them. For results without batch norm, see Figure 7.

Networks are trained for 50 epochs with batchsize 100. All methods use momentum of 0.9 unless noted otherwise.

For Polyak Averaging, we follow [Grosse and Martens, 2016] and also use the decay value recommended there without further tuning.

Amortisation: We found that KFAC and FOOF can be amortised fairly strongly without sacrificing performance. We invert the Kronecker-factors every $T = 500$ timesteps (i.e. once per epoch) and only update the exponentially moving averages for the Kronecker-factors for the $S = 10$ steps immediately before inversion. We also performed thorough hyperparameter searches with $T = 250, S = 50$ (larger values of S make little sense, due to their limited influence on the exponentially moving averages) which gave essentially identical results. Preliminary experiments with $T = 100$ also did not perform better than the schedule described above and used for our experiments.

D.6 Version of FOOF with λ_A from KFAC

In Figure 4 (and there only), we use a version of FOOF which always uses exactly the same damping term λ_A as KFAC to obtain a comparison that's as clean as possible. Concretely, this means carrying out most computations as in KFAC and computing $\mathbf{E}_F\mathbf{E}_F^T$ and λ_A, λ_E as in KFAC and then computing the parameter update only using the first kronecker-factor $\mathbf{A}\mathbf{A}^T$. Note that λ_A varies during training, and typically increases so that this version of FOOF is slightly different (and worse in terms of performance) from standard FOOF.

For Figure 4D, we made another modification: An increase in λ_A leads to a decrease in effective step-size of the above version of FOOF. In KFAC this is compensated by a decrease in damping of the second factor so that the effective stepsize is roughly constant. To compensate analogously in FOOF, we simply multiply the update by the scalar $\lambda_E^{-1} = \lambda_A/\lambda$ to maintain a constant stepsize. For standard FOOF (in all other figures) we use constant λ_A and make no such modifications.

E Derivation of KFAC

KFAC was invented, and of course derived in [Martens and Grosse, 2015]. Here, we simply restate the derivation and point out when it can be expected to be accurate or inaccurate.

We focus on one layer and use previous notation. In particular, we consider one datapoint (X_i, y_i) (remember that y_i is a sample from the model distribution) and denote the input of the layer as $\mathbf{a}_i$, the output as $\mathbf{b}_i = \mathbf{W}\mathbf{a}_i$ and $\mathbf{e}_i = \frac{\partial L}{\partial \mathbf{b}_i}$. For this single datapoint, the block-diagonal part of the Fisher given by this datapoint is exactly equal to

$$\mathbf{F}_i = (\mathbf{a}_i\mathbf{a}_i^T) \otimes (\mathbf{e}_i\mathbf{e}_i^T)$$

Recall that the Fisher is defined as an expectation over datapoints $\mathbf{F} = \mathbb{E}[\mathbf{F}_i]$. If we use a Monte-Carlo approximation of this expectation, the Fisher is approximated as

$$\mathbf{F} = \sum_{i \in I} \left(\mathbf{a}_i\mathbf{a}_i^T \otimes \mathbf{e}_i\mathbf{e}_i^T\right)$$

Now, KFAC makes the following further approximation:

$$\mathbf{F} = \sum_{i \in I} \left(\mathbf{a}_i\mathbf{a}_i^T \otimes \mathbf{e}_i\mathbf{e}_i^T\right) \approx \left(\sum_{i \in I} \mathbf{a}_i\mathbf{a}_i^T\right) \otimes \left(\sum_{i \in I} \mathbf{e}_i\mathbf{e}_i^T\right)$$

As we have demonstrated in our experiments, this is exactly the approximation that makes KFAC such an effective optimizer.

In general, this approximation is imprecise: It is equivalent to approximating a rank $|I|$ matrix by a rank 1 matrix[5]. If we take for example the MNIST dataset, which has 60,000 inputs with 10 labels each, the full diagonal block of the Fisher has rank 600,000, while the approximation has rank 1. In concolutional networks this is even more extreme: The full rank of the Fisher is multiplied by the number of locations at which the filer is applied, while our approximation stays at rank 1.

So in general, this approximation does not hold. In the literature it is usually justified by an independence assumption. Concretely, we view $\mathbf{a}_i, \mathbf{e}_i$ as random variables, where the randomness jointly depends on which datapoint we draw. However, note that in non-degenerate neural networks we will usually be able to uniquely identify what datapoint was used, if we are given $\mathbf{e}_i$ (It is extremly unlikely that a back-propagated derivative is the same for two different datapoints – so there is a one-to-one mapping from datapoints to $\mathbf{e}_i$). This implies that the conditional entropy $H(\mathbf{a}_i \mid \mathbf{e}_i) = 0$, in particular $\mathbf{a}_i, \mathbf{e}_i$ are not independent.

We also note that if we consider linear networks and regression problems with homoscedastic gaussian noise, then the backpropaged derivatives $\mathbf{e}_i$ are completely independent of the datapoint – and in particular independent of $\mathbf{a}_i$. This is because the derivatives at the last layer are a function of the covariance matrix of the noise (and independent of the in-/output of the network), and all derivatives are backpropagated through the same linear network. This is used in [Bernacchia et al., 2019] for the analysis in linear neural networks. The above reasoning also explains where this breaks down for non-linear networks and non-homoscedastic regression problems.

F Toy Example Illustrating the Difference between SGD and FOOF

In the main paper, we pointed out that FOOF trades-off conflicting gradients differently (seemingly better) than SGD. Here, we provide a toy example illustrating this point.

Roughly, the example will show that in SGD, gradients from one datapoint can ”overwrite” gradients from other datapoints, so that SGD does not decrease the loss on the latter, while in FOOF the update will make progress

[5] To see this, simply reshape the matrix F by first flattening $\mathbf{a}_i\mathbf{a}_i^T, \mathbf{e}_i\mathbf{e}_i^T$ and then replacing the Kronecker-product by the standard outer product of vectors. The resulting matrix has the same entries as F and each F_i has rank 1. But in general, the sum over F_i has rank $|I|$.

on both datapoints simultaneously.
Our example will consist of a linear regression network with two inputs and one output and will feature two datapoints. The datapoints have inputs (3,1) and (1,0) and labels (1), (-1) and the weight vector (consisting of two weights) is initialised to (0,0). Taking the squared distance as loss function, It is easily verified that the gradients for the datapoitns are (3,1) and (-1,0). The SGD update direction is (2, 1) and will increase the loss on the second datapoint independent of step size. In contrast, the (un-damped) FOOF update is given by (-1, 4) which decreases the loss on both datapoints for suitably small stepsize. In particular a single update-step of FOOF with stepsize 1 converges to a global optimum.

In this context, it may be worth noting that applying FOOF to single datapoints and averaging resulting updates (rather than computing the FOOF update jointly on the entire batch), corresponds to a version of SGD, in which each datapoint has a slightly different learning rate. We tested this version and found that it does not perform notably better than SGD. This further supports the intuition that FOOFs advantgae over SGD comes from combining conflicting gradient directions more effectively.

G Kronecker-Factored Curvature Approximations for Laplace Posteriors

A Kronecker-factored approximation of the curvature has also been used in the context of laplace posteriors [Ritter et al., 2018b] and this has been applied to continual learning [Ritter et al., 2018a]. In both context, empirical results are very encouraging. Our finding that, in the context of optimization, the effectiveness of KFAC does not rely on its similarity to the Fisher raises the question whether these other applications [Ritter et al., 2018b, Ritter et al., 2018a] of Kronecker-factorisations of the curvature rely on proximity to the curvature matrix.

Here, we provide an alternative hypothesis. For simplicity of notation, we assume that the network has only one layer, but the analysis straightforwardly generalises to more layers. Suppose $\mathbf{W}_0$ is a local minimum of the negative log-likelihood of the parameters. Further denote the approximation to the posterior covariance by $\mathbf{\Sigma}$. For an approximate posterior to be effective, we require that parameters which are assigned high likelihood by the posterior actually do have high likelihood according to the data distribution. In other words, when a weight pertubation $\mathbf{V}$ satisfies that $\text{vec}\left(\mathbf{V}\right)^T \mathbf{\Sigma}\text{vec}\left(\mathbf{V}\right)$ is small (high likelihood according according to the approximate posterior), then the parameter $\mathbf{W}_0 + \mathbf{V}$ should have low loss (i.e. high likelihood according to the data).

For the Kronecker-factorisation, the first factor again is given by $\mathbf{A}\mathbf{A}^T$. Let us assume again that the second factor is dominated by a damping term (a very similar argument works if the second factor is predominantly diagonal), so that the posterior covariance is approximately $\mathbf{\Sigma} \approx \mathbf{A}\mathbf{A}^T \otimes \mathbf{I}$. Then, some easily verified calculations give

$$\text{vec}\left(\mathbf{V}\right)^T \mathbf{\Sigma}\text{vec}\left(\mathbf{V}\right) \approx \text{vec}\left(\mathbf{V}\right)^T (\mathbf{A}\mathbf{A}^T \otimes \mathbf{I})\text{vec}\left(\mathbf{V}\right) = \sum_i \mathbf{v}_i^T (\mathbf{A}\mathbf{A}^T)\mathbf{v}_i \tag{13}$$

where $\mathbf{v}_i$ is the i-th row of $\mathbf{V}$, i.e. the set of weights connected to the i-th output neuron.[6] This expression being small means that each row of the pertubation $\mathbf{V}$ is near orthogonal to the input activations $\mathbf{A}$ (or more formally, it aligns with singular vectors of $\mathbf{A}$ which correspond to small singular values). This means that the layer's output, and consequently the networks output, get perturbed very little. This in turn means, that $\mathbf{W}_0 + \mathbf{V}$ has high-likelihood.

A simple test of this hypothesis would be to keep only the first kronecker-factor $\mathbf{A}\mathbf{A}^T$, replace the second one by the identity and check if the method performs equally well or better. Further, it would be interesting to compare the performance of kronecker-factored posterior to a full-laplace posterior (controlling for the amount of data given to both) and check if – analogous to our results for optimization – the kronecker-factored posterior outperforms the exact laplace posterior.

In fact, a very similar algorithm has already been developed independently [Ober and Aitchison, 2021]. It shows strong performance, consistent with our prediction.

[6] If the second kronecker-factor is not the identity, then there are additional cross terms of the form $b_{ij}\mathbf{v}_i^T(\mathbf{A}\mathbf{A}^T)\mathbf{v}_j$, where b_{ij} is the i, j-th entry of the second kronecker-factor.

In addition, we note that a drawback of the continual learning method [Ritter et al., 2018a] based on Kronecker-factors is that it needs to store kronecker-factors for each task, resulting in very large memory requirements as well as increased computational cost, especially as the number of tasks grows. Formally, individual Kronecker-factors from each task need to be stored, since the sum of two Kronecker-products (from different tasks) can not generally be written as a single product. If our hypothesis is true, this would theoretically prescribe ignoring the second factor of each product and moreover adding the first factors (from different tasks, for a fixed layer) directly. This would significantly reduce the memory footprint of the resulting method. In fact, this algorithm would simply be an application of [Ober and Aitchison, 2021] to continual learning.

H Related Work

We review generally related work as well as more specifically algorithms with similar updates rules to FOOF.

H.1 Generally Related Work

Natural gradients were proposed by Amari and colleagues, see e.g. [Amari, 1998] and its original motivation stems from information geometry [Amari and Nagaoka, 2000]. It is closely linked to classical second-order optimization through the link of the Fisher to the Hessian and the Generalised Gauss Newton matrix [Martens, 2014, Pascanu and Bengio, 2013]. Moreover, natural gradients can be seen as a special case of Kalman filtering [Ollivier, 2018]. Interestingly, different filtering equations can be used to justify Adam's [Kingma and Ba, 2014] update rule [Aitchison, 2018].

There is a long history of approximating natural gradients and second order methods. For example, HF [Martens et al., 2010] exploits that Hessian-vector products are efficiently computable and uses the conjugate-gradient method to approximate products of the inverse Hessian and vectors. In this case, similarly to our application, the Hessian is usually subsampled, i.e. evaluated on a mini-batch. Other approximations of natural gradients include [Roux et al., 2007, Ollivier, 2015, Ollivier, 2017, Grosse and Salakhudinov, 2015, Desjardins et al., 2015, Martens et al., 2010, Marceau-Caron and Ollivier, 2016].

The intrinsic low rank structure of the Fisher has been exploited in a number of setups by a number of papers including [Agarwal et al., 2019, Goldfarb et al., 2020, Immer et al., 2021, Dangel et al., 2021]. For more discussion of [Agarwal et al., 2019, Goldfarb et al., 2020] see Appendix I.

Kronecker-factored approximations [Martens and Grosse, 2015, Grosse and Salakhudinov, 2015] have become the basis of several optimization algorithms [Botev et al., 2017, Goldfarb et al., 2020, George et al., 2018, Bernacchia et al., 2019]. Our contribution may shed light on why this is the case.

Moreover, Kronecker-factored approximation of the curvature can be used in the context of Laplace Posteriors [Ritter et al., 2018b], which can also be applied to continual learning [Ritter et al., 2018a]. A more detailed discussion of how this relates to our findings can be found in Section G.

H.1.1 Theoretical Work

There also is a large body of work on theoretical convergence properties of Natural Gradients. We give a brief, incomplete overview here and refer to [Zhang et al., 2019] for a more thorough discussion.

[Bernacchia et al., 2019] analyse the convergence of natural gradients in linear networks. Interestingly, they show that for linear networks applied to regression problems (with homoscedastic noise), inverting a block-diagonal, Kronecker-Factored approximation of the curvature results in exact natural gradients, see also Appendix E for a brief justification of a part of their findings. Our results empirically show that this link breaks down in the non-linear case and we explain theoretically why this is the case in Appendix E.

For non-linear, overparametrised two-layer networks in which only the first layer is trained, [Zhang et al., 2019] recently gave a convergence analysis of both natural gradients and KFAC. Note that our results, investigating how related KFAC and natural gradients are, are orthogonal to these findings, as [Zhang et al., 2019] do not establish similarity between KFAC and Natural Gradients but rather give two separate convergence proofs.

A set of interesting theoretical results by [Karakida et al., 2021] shows that the Fisher Information in deep neural networks has a pathological spectrum – in particular, they show that the Fisher is flat in most directions. This view may well give a theoretical intuition for why Subsampled Natural Gradients do not notably outperform SGD.

H.1.2 Related Work from Bayesian ML

Similar to our new view on optimization is [Ober and Aitchison, 2021], which is a Bayesian Posterior approximation and can (roughly) be viewed as considering distributions over neuron activations rather than in weight space directly, similarly to how FOOF performs optimization steps on neuron activations rather than on weights directly. The findings of [Ober and Aitchison, 2021] also indirectly support our interpretation of [Ritter et al., 2018b] (see Appendix G).

H.2 Algorithms with similar update rules

While it is not immediately visible due to a re-parametrisation employed in [Desjardins et al., 2015], Natural Neural Networks [Desjardins et al., 2015] (NNN) propose a mathematically very similar update rule to FOOF (and KFAC). Unlike FOOF, NNN centers layer inputs by subtracting the mean activation (or an estimate thereof), but like FOOF they ignore the second kronecker factor of KFAC.

Like KFAC, NNN is derived as a block-diagonal, kronecker-factored approximation of the Fisher. As we already pointed out, this is very puzzling, since NNN approximates the Fisher by a zero-th order matrix, ignoring all first- and second-order information. In this sense, NNN should not be seen as a second-order optimizer and our results explain why it is nevertheless so effective.

From an implementational viewpoint, FOOF is preferable to NNN mainly because it requires inverting matrices rather than computing SVDs. In practice computing inverses is both considerably faster (a factor of 10 or so as found in some quick experiments) and more stable then computing the SVD, as NNN does (SVD algorithms don't always converge).

With yet another context and motivation, [Frerix et al., 2017] also proposes a similar update rule focussing on full-batch descent. The motivation can be roughly rephrased as imposing proximity constraints on neuron activations. Very recently, their motivation and algorithm seems to have been re-described in [Amid et al., 2021] without noting this link. Among other differences, both of these papers (1) seem not to discuss unbiased stochastic versions of their algorithms, (2) seem less computationally efficient: results in [Frerix et al., 2017] fall short of adam in terms of wall-clock time and [Amid et al., 2021] does not provide direct wall-clock time comparisons with standard first-order optimizers, (3) only discuss fully-connected architectures (4) do not perform investigations into the connection of KFAC to natural gradients or first-order methods.

I Details for Efficiently Computing $\mathbf{F}^{-1}$-vector products for a Subsampled Fisher

I.1 Notation

For simplicity, we restrict the exposition here to fully connected neural networks without biases and to classification problems. Our method is also applicable to regression problems, can easily be extended to include biases and to handle for example convolutional layers.

We denote the network's weight matrices by $\mathbf{W} = \left(\mathbf{W}^0, \ldots, \mathbf{W}^{(\ell-1)}\right)$, where $W^{(i)}$ has dimensions $n_i \times n_{i+1}$. The pointwise non-linearity will be denoted $\sigma(\cdot)$. For a single input $\mathbf{x} = \mathbf{a}^{(0)}$ the network iteratively computes

$$s^{(k)} = \mathbf{W}^{(k-1)}\mathbf{a}^{(k-1)} \qquad \text{for } k = 1, \ldots, \ell \tag{14}$$

$$\mathbf{a}^{(k)} = \sigma(\mathbf{s}^{(k)}) \qquad \text{for } k = 1, \ldots, \ell - 1 \tag{15}$$

$$f(\mathbf{x}) = f(\mathbf{x}; \mathbf{W}) = \text{softmax}(a^{(\ell)}) \tag{16}$$

We use the cross-entropy loss $L(f(\mathbf{x}), y)$ throughout. We will write $\mathbf{e}^k = \frac{\partial L(f(x,y))}{\partial s_k}$ for the errors, which are usually computed by backpropagation.

If we process a batch of data, we will use upper case letters for activations and "errors", i.e. $\mathbf{A}^{(k)}, \mathbf{S}^{(k)}, \mathbf{E}^{(k)}$ which have dimensions $n_k \times B$, where B is the batch size. The i-th column of these matrices will be denoted by corresponding lower case letters, e.g. $\mathbf{a}_i^{(k)}$.

We will write $\mathbf{A} \odot \mathbf{B}$ for the pointwise (or Hadamard) product of $\mathbf{A}, \mathbf{B}$ and $\mathbf{A} \otimes \mathbf{B}$ for the Kronecker product. The euclidean inner product (or dot product) will be denoted by $\mathbf{A} \cdot \mathbf{B}$ for both vectors and matrices.

The number of parameters will be called $n = \sum_{k=0}^{\ell-1} n_k n_{k+1}$, the batch size B, the output dimension of the network n_ℓ.

We will generally assume derivatives to be one dimensional column vectors and will often write $\mathbf{g} = \mathbf{g}(\mathbf{x}, y) = \frac{\partial L(f(\mathbf{x}),y)}{\partial \mathbf{W}}$ and $\mathbf{g}^{(k)} = \frac{\partial L(f(\mathbf{x}),y)}{\partial \mathbf{W}}$. Generally, for a vector $\mathbf{u}$ of dimension n, the superscript $\mathbf{u}^{(k)}$ will denote the entries of $\mathbf{u}$ corresponding to layer k, and $\operatorname{mat}\left(\mathbf{u}^{(k)}\right)$ will be a matrix with the same entries as $\mathbf{u}^{(k)}$ and of the same dimensions as $\mathbf{W}^{(k)}$.

I.2 Overview

The technique described here is similar to [Agarwal et al., 2019, Ren and Goldfarb, 2019]. However, the implementation of [Ren and Goldfarb, 2019] requires several for- and backward passes for each mini-batch, which is used to compute the Fisher, while [Agarwal et al., 2019] uses the same ideas, but applies them in a different context, which does not require computing the Fisher. Another key difference, both in terms of computation time and update-direction quality (or bias of updates) is discussed in Section I.5.

We now outline how to efficiently compute exact natural gradients under the assumption that the Fisher is estimated from a mini-batch of moderate size (where 'moderate' can be on the order of thousands without large difficulties). Let's assume we have B samples $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_B, y_B)$ from the model distribution and use this for a MC estimate of the Fisher, i.e.

$$\mathbf{F} = \frac{1}{B} \sum_{i=1}^{B} \mathbf{g}_i \mathbf{g}_i^T = \mathbf{G}\mathbf{G}^T \tag{17}$$

where we define $\mathbf{G}$ to be the matrix whose i-th column is given by $\frac{1}{\sqrt{B}} \mathbf{g}_i$. An application of the matrix inversion lemma now gives

$$(\lambda \mathbf{I} + \mathbf{F})^{-1} \mathbf{u} = (\lambda \mathbf{I} + \mathbf{G}\mathbf{G}^T)^{-1} = \lambda^{-1} \mathbf{I}\mathbf{u} - \lambda^{-2} \mathbf{G}(\mathbf{I} + \frac{1}{\lambda} \mathbf{G}^T \mathbf{G})^{-1} \mathbf{G}^T \mathbf{u} \tag{18}$$

We will not compute $\mathbf{G}$ explicitly. Rather, we will see that all needed quantities can be computed efficiently from the quantities obtained during a single standard for- and backward pass on the batch $\{(\mathbf{x}_i, y_i)\}_{i=1}^B$, namely the preactivations $\mathbf{A}^{(k)}$ and error $\mathbf{E}^{(k+1)}$.[7]

Overall, the computation can be split into three steps. (1) We need to compute $\mathbf{v} = \mathbf{G}^T \mathbf{u}$. (2) We need to compute $\mathbf{G}^T \mathbf{G}$, after which evaluating $\mathbf{w} = (\mathbf{I} + \frac{1}{\lambda} \mathbf{G}^T \mathbf{G})^{-1} \mathbf{v}$ is easy by explicitly computing the inverse. (3) We need to evaluate $\mathbf{G}\mathbf{w}$.

Very briefly, the techniques to compute (1)-(3) all rely on the fact that, for a single datapoint, the gradient with respect to a weight matrix is a rank 1 matrix and that, consequently, gradient-vector and gradient-gradient dot products can be computed and vectorised efficiently.

I.3 Details

We now go through the steps (1)-(3) described above.

For (1), note that the i-th entry of $\mathbf{v} = \mathbf{G}^T \mathbf{u}$ is the dot-product between $\mathbf{g}_i$ and $\mathbf{u}$, which in turn is the sum over layer-wise dot-products $\mathbf{g}_i^{(k)} \cdot \mathbf{u}^{(k)}$. Note that $\operatorname{mat}\left(\mathbf{g}_i^{(k)}\right) = \mathbf{a}_i^{(k)} \mathbf{e}_i^{(k)T}$ is a rank one matrix, so that $\mathbf{g}_i^{(k)} \cdot \mathbf{u}^{(k)} = \mathbf{a}_i^{(k)T} \operatorname{mat}\left(\mathbf{u}^{(k)}\right) \mathbf{e}^{(k)}$. A sufficiently efficient way to vectorise these computations is the following:

$$\mathbf{v} = \mathbf{G}^T \mathbf{u} = \sum_{k=0}^{\ell-1} \operatorname{diag}\left(\mathbf{A}^{(k),T} \operatorname{mat}\left(\mathbf{u}^{(k)}\right) \mathbf{E}^{(k+1)}\right) \tag{19}$$

[7] We note that many of the required quantities can be seen as Jacobian-vector products and could be computed with autograd and additional for- and backward passes. Here, we simply store preactivations and errors from a single for- and backward pass to avoid additional passes through the model.

For (2), similar considerations give

$$\mathbf{G}^T\mathbf{G} = \sum_{k=0}^{\ell-1} (\mathbf{A}^{(k)T}\mathbf{A}^{(k)}) \odot (\mathbf{E}^{(k+1)T}\mathbf{E}^{(k+1)}) \tag{20}$$

Finally, for (3), note that $\mathbf{Gw}$ is a linear combination of the gradients (columns) of $\mathbf{G}$. Writing $\mathbf{1}$ for a column vector with n_k ones, this can be computed layer-wise as

$$\operatorname{mat}\left((\mathbf{Gw})^{(k)}\right) = \mathbf{A}^{(k)T}\left(\mathbf{E} \odot (\mathbf{w}\mathbf{1}^T)\right) \tag{21}$$

We re-emphasise that we only require to know $\mathbf{A}^{(k)}$ and $\mathbf{E}^{(k+1)}$, which can be computed in a single for- and backward pass and then stored and re-used for computations.

I.4 Computational Complexity

In terms of memory, we need to store $\mathbf{A}^k, \mathbf{E}^k$, which requires at most as much space as a single backward pass. Storing $\mathbf{G}^T\mathbf{G}$ requires space $B \times B$, which is typically negligible. as are results of intermediate computation.

In terms of time, computations (19) takes time $O(B\sum_k n_k^2))$, (20) takes time $O(B\sum_k n_k^2)$, (21) requires $O(Bn)$. The matrix inverision requires $O(B^3)$, but note that technically we only need to evaluate the product of the inverse with a single vector, which theoretically can be done slightly faster (so can some of the matrix multiplications).

I.5 Less biased Subsampled Natural Gradients

Our aim is to estimate $(\lambda\mathbf{I} + \mathbf{F})^{-1}\mathbf{g}$ and we use mini-batches estimates $\bar{\mathbf{F}}$ and $\bar{\mathbf{g}}$. Ideally, we would want an unbiased estimate, i.e. an estimate with mean $(\lambda\mathbf{I} + \mathbb{E}[\bar{\mathbf{F}}])^{-1} \cdot \mathbb{E}[\bar{\mathbf{g}}]$.

One problem that seems hard to circumvent is that, while our estimate $\bar{\mathbf{F}}$ of $\mathbf{F}$ is unbiased, the expectation of $(\lambda\mathbf{I} + \bar{\mathbf{F}})^{-1}$ will not be equal to $(\lambda\mathbf{I} + \mathbf{F})^{-1}$. We shall not resolve this problem here and simply hope that its impact is not detrimental.

Another problem is that using the same mini-batch to estimate Fisher $\mathbf{F}$ and gradient $\mathbf{g}$ will introduce additional bias: Even if $X = (\lambda\mathbf{I} + \bar{\mathbf{F}})^{-1}$ were an unbiased estimate of $(\lambda\mathbf{I} + \mathbf{F})^{-1}$ and $Y = \bar{\mathbf{g}}$ is an unbiased estimate of $\mathbf{g}$, it does not automatically hold that XY is an unbiased estimate of $\mathbb{E}[X]\mathbb{E}[Y]$. This does however hold, if X, Y are independent, which can be achieved by estimating them based on independent mini-batches. The fact that a bias of this kind can meaningfully affect results, also in the context of modern neural networks and standard benchmarks, has already been observed in [Benzing, 2020].

Thus, we propose using independent mini-batches to estimate $\bar{\mathbf{F}}$ and $\bar{\mathbf{g}}$. On top of removing bias from our estimate, this has the additional benefit that we do not have to update $\bar{\mathbf{F}}$ (or rather the quantities related to it) at every time step. This gives further computational savings.

We perform an ablation experiment for this choice in Figures J.13 and J.14.

J Additional Experiments

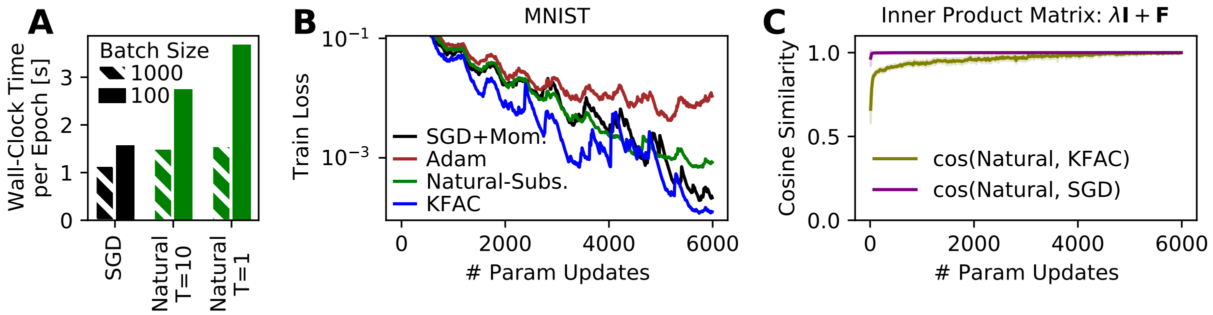

Figure J.8: Same as Figure 1 but on MNIST. Note that the optimal damping hyperparameter for KFAC is comparatively large, leading to high similarity between all methods with SGD.

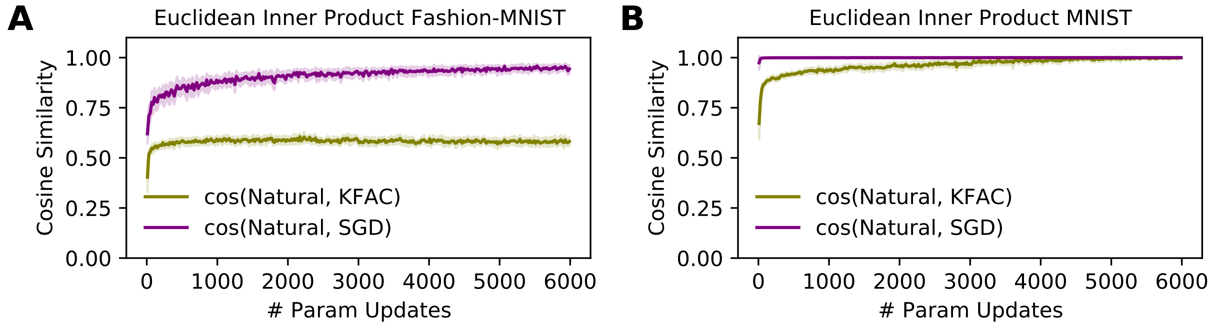

Figure J.9: Same as Figures 1, J.8C but with the euclidean inner product. Recall that the optimal damping hyperparameter for KFAC on MNIST is comparatively large, leading to high similarity between all methods with SGD.

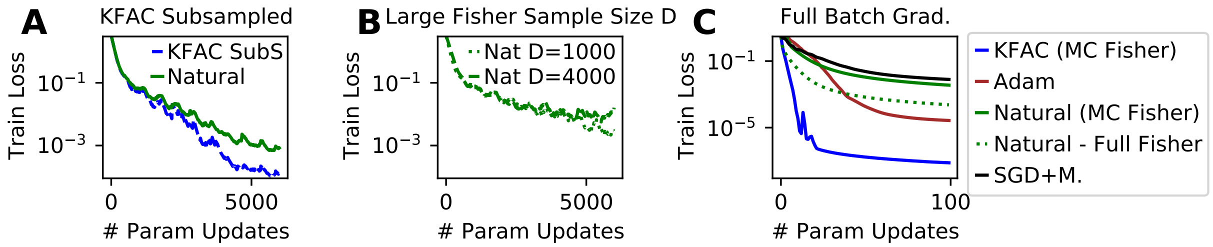

Figure J.10: **Advantage of KFAC is not due to using more data to estimate the Fisher.** Same as Figure 2 but on MNIST rather than Fashion MNIST.

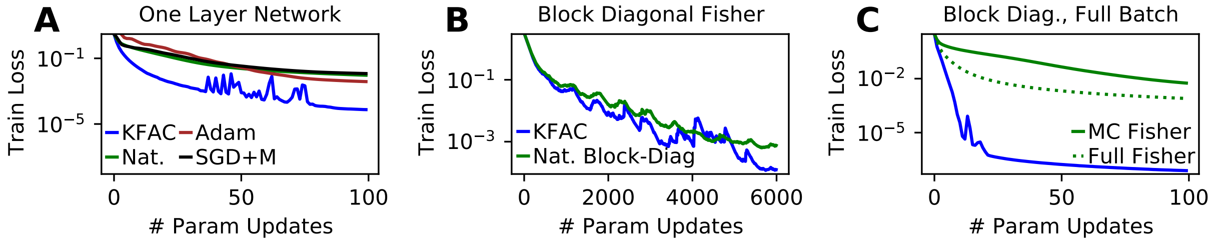

Figure J.11: **Advantage of KFAC is not due to block-diagonal structure.** Same as Figure 3 but on MNIST rather than Fashion MNIST.

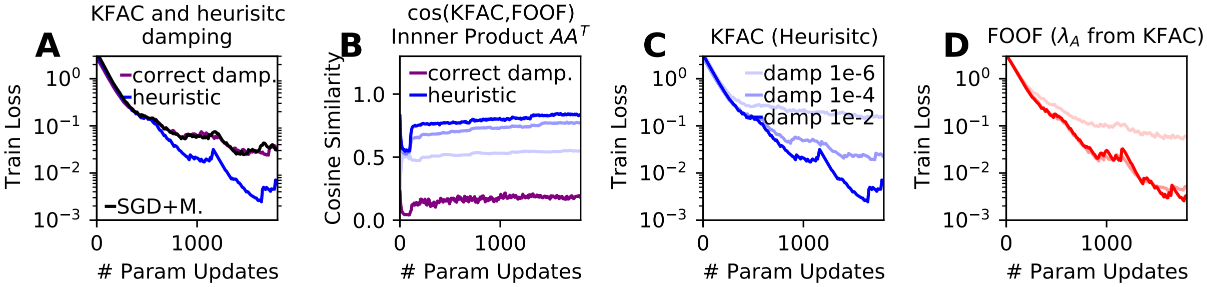

Figure J.12: **Damping increases KFAC's performance as well as its similarity to our first order method FOOF.** Same as Figure 4 but on MNIST rather than Fashion MNIST.

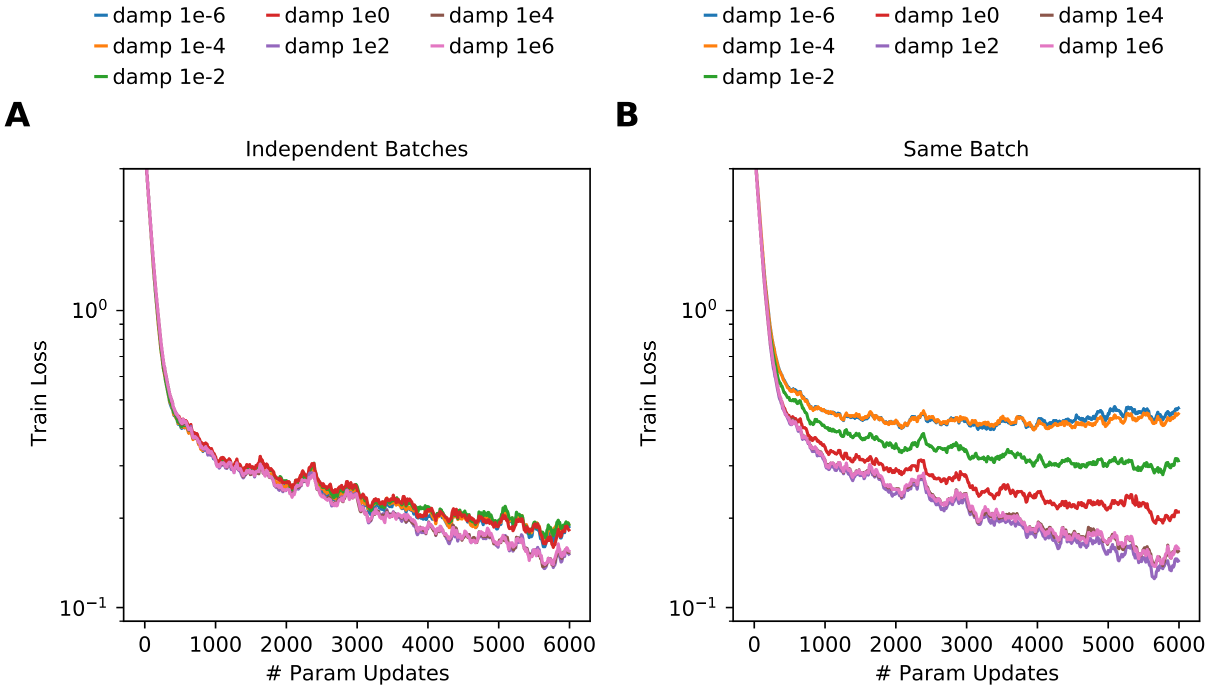

Figure J.13: Comparison between computing Fisher Information and Gradient on independent or same mini-batches. In line with our usual procedure, the learning rate was re-optimised for each damping strength and each of the two options. This plot is on Fashion MNIST. For the same plot on MNIST see below.

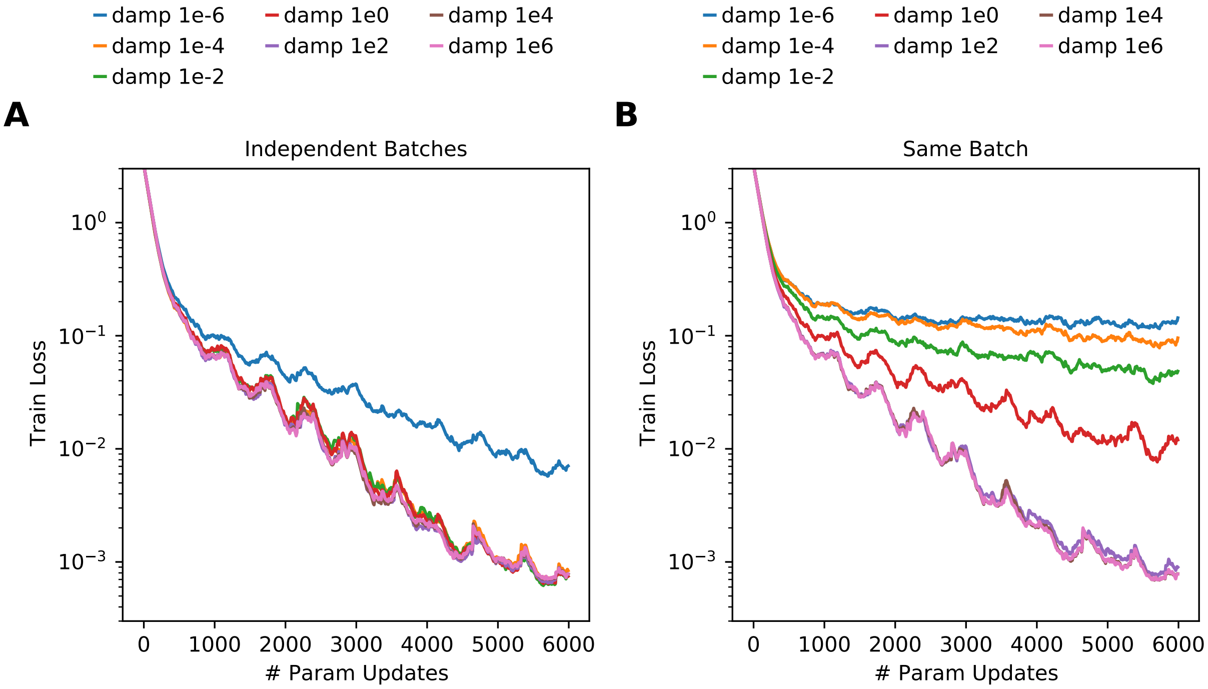

Figure J.14: Same as Figure J.13, but on MNIST.

Algorithm 1 Gradient Descent on Neurons (FOOF)

```
 1: **Hyperparameters:** learning rate $\eta$, damping strength $\lambda$, exponential decay factor $m$, inversion frequency
    $T$, number of updates for input covariance $S \leq T$
 2: **Initialise:** $t = 0$; For each layer $\ell$: Weights $\mathbf{W}_\ell$ (e.g. Kaiming-He init), exponential average $\mathbf{\Sigma}_\ell$ of $\mathbf{A}_\ell \mathbf{A}_\ell^T$
    and its damped inverse $\mathbf{P}_\ell = (\mathbf{\Sigma}_\ell + \lambda \mathbf{I})^{-1}$ (see Appendix K for details)

 3: while train do
 4:   Perform Standard Forward and Backward Pass For Current Mini-Batch With Loss $L$
 5:   for each layer $\ell$ do
 6:     $\mathbf{W}_\ell \leftarrow \mathbf{W}_\ell - \eta \mathbf{P}_\ell \nabla_{\mathbf{W}_\ell} L$ {Update Parameters as in Eq. 4}
 7:     if $(t \bmod T) == 0$ then
 8:       $\mathbf{P}_\ell \leftarrow (\mathbf{\Sigma}_\ell + \lambda \mathbf{I})^{-1}$ {Update Damped Inverse of Moving Average of $\mathbf{A}_\ell \mathbf{A}_\ell^T$ every $T$ steps}
 9:     end if
10:     if $((t + S) \bmod T) \in \{0, \ldots, S - 1\}$ then
11:       $\mathbf{\Sigma} \leftarrow m \cdot \mathbf{\Sigma}_\ell + (1 - m) \cdot \mathbf{A}_\ell \mathbf{A}_\ell^T$ {Update Moving Average of $\mathbf{A}_\ell \mathbf{A}_\ell^T$ beginning $S$ steps
                                             before inversion in line 8. $\mathbf{A}_\ell$ is defined as in Section 2.1.}
12:     end if
13:   end for
14:   $t \leftarrow t + 1$
15: end while
```

K Pseudocode, Implementation, Hyperparameters

Pseudocode for FOOF is given in Algorithm 1. Notation is anaologous to Section 2.1 and the amortisation described in Section D.

Initialisation: One detail omitted in the pseudocode is initialisation of $\mathbf{\Sigma}$ and $\mathbf{P}$. There is different ways to do this. We decided to perform Line 11 of Algorithm 1 for a number of minibatches (50) before training and then executing line Line 8 once. In addition, we make sure the exponentially moving average is normalised.

Amortisation Choices: Amortising the overhead of FOOF is achieved by choosing S, T suitably (large T and small S give the best runtimes). For fully connected layers updating $\mathbf{\Sigma}$ is cheap and we choose $S = T$ (i.e. $\mathbf{\Sigma}$ is updated at every step), we reported results for $T = 1$ and $T = 100$. For the ResNet, computing $\mathbf{A}\mathbf{A}^T$ is more expensive and we chose $T = 500$ (one inversion per epoch) and $S = 10$, see also Appendix D as well as below. Additional experiments (not shown) suggest that $\mathbf{\Sigma}$ can be estimated robustly on few datapoints and that it changes slowly during training.

Hyperparameter Choices: We chose $m = 0.95$ following [Martens and Grosse, 2015], brief experiments with $m = 0.999$ seemed to give very similar results. For damping λ and learning rate η, we performed grid searches. It may be interesting that a bayesian interpretation of FOOF (details omitted) suggests choosing λ as the precision used for standard weight initialisation schemes (e.g. Kaiming Normal initialisation) and seems to work well. If we choose this λ, we only need to tune the learning rate of FOOF, so that the required tuning is analogous to that of SGD.

Implementation and Convolutional Layers: A PyTorch implementation of FOOF using for- and backward hooks is simple. The implementation, in particular computing $\mathbf{A}\mathbf{A}^T$, is most straightforward for fully connected layers, but can be extended to layers with parameter sharing. For example in CNNs, we can interpret the convolution as a standard matrix multiplication by "extracting/unfolding" individual patches (see e.g. [Grosse and Martens, 2016]) and then proceed as before. This is what our implementation does. A more efficient technique avoiding explicitly extracting patches is presented in [Ober and Aitchison, 2021].

References

[Agarwal et al., 2019] Agarwal, N., Bullins, B., Chen, X., Hazan, E., Singh, K., Zhang, C., and Zhang, Y. (2019). Efficient full-matrix adaptive regularization. In *International Conference on Machine Learning*, pages 102–110. PMLR.

[Aitchison, 2018] Aitchison, L. (2018). Bayesian filtering unifies adaptive and non-adaptive neural network optimization methods. *arXiv preprint arXiv:1807.07540*.

[Amari, 1998] Amari, S.-I. (1998). Natural gradient works efficiently in learning. *Neural computation*, 10(2):251–276.

[Amari and Nagaoka, 2000] Amari, S.-i. and Nagaoka, H. (2000). *Methods of information geometry*, volume 191. American Mathematical Soc.

[Amid et al., 2021] Amid, E., Anil, R., and Warmuth, M. K. (2021). Locoprop: Enhancing backprop via local loss optimization. *arXiv preprint arXiv:2106.06199*.

[Ba et al., 2016] Ba, J., Grosse, R., and Martens, J. (2016). Distributed second-order optimization using kronecker-factored approximations.

[Bengio, 2000] Bengio, Y. (2000). Gradient-based optimization of hyperparameters. *Neural computation*, 12(8):1889–1900.

[Benzing, 2020] Benzing, F. (2020). Unifying regularisation methods for continual learning. *arXiv preprint arXiv:2006.06357*.

[Bernacchia et al., 2019] Bernacchia, A., Lengyel, M., and Hennequin, G. (2019). Exact natural gradient in deep linear networks and application to the nonlinear case. NIPS.

[Blundell et al., 2015] Blundell, C., Cornebise, J., Kavukcuoglu, K., and Wierstra, D. (2015). Weight uncertainty in neural network. In *International Conference on Machine Learning*, pages 1613–1622. PMLR.

[Botev et al., 2017] Botev, A., Ritter, H., and Barber, D. (2017). Practical gauss-newton optimisation for deep learning. In *International Conference on Machine Learning*, pages 557–565. PMLR.

[Dangel et al., 2020] Dangel, F., Harmeling, S., and Hennig, P. (2020). Modular block-diagonal curvature approximations for feedforward architectures. In *International Conference on Artificial Intelligence and Statistics*, pages 799–808. PMLR.

[Dangel et al., 2021] Dangel, F., Tatzel, L., and Hennig, P. (2021). Vivit: Curvature access through the generalized gauss-newton's low-rank structure. *arXiv preprint arXiv:2106.02624*.

[Dauphin et al., 2014] Dauphin, Y., Pascanu, R., Gulcehre, C., Cho, K., Ganguli, S., and Bengio, Y. (2014). Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. *arXiv preprint arXiv:1406.2572*.

[Daxberger et al., 2021] Daxberger, E., Kristiadi, A., Immer, A., Eschenhagen, R., Bauer, M., and Hennig, P. (2021). Laplace redux–effortless bayesian deep learning. *arXiv preprint arXiv:2106.14806*.

[Desjardins et al., 2015] Desjardins, G., Simonyan, K., Pascanu, R., and Kavukcuoglu, K. (2015). Natural neural networks. *arXiv preprint arXiv:1507.00210*.

[Doucet, 2010] Doucet, A. (2010). A Note on Efficient Conditional Simulation of Gaussian Distributions. [Online; accessed 17-September-2021].

[Frerix et al., 2017] Frerix, T., Möllenhoff, T., Moeller, M., and Cremers, D. (2017). Proximal backpropagation. *arXiv preprint arXiv:1706.04638*.

[George et al., 2018] George, T., Laurent, C., Bouthillier, X., Ballas, N., and Vincent, P. (2018). Fast approximate natural gradient descent in a kronecker-factored eigenbasis. *arXiv preprint arXiv:1806.03884*.

[Glorot and Bengio, 2010] Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256. JMLR Workshop and Conference Proceedings.

[Goldfarb et al., 2020] Goldfarb, D., Ren, Y., and Bahamou, A. (2020). Practical quasi-newton methods for training deep neural networks. *arXiv preprint arXiv:2006.08877*.

[Grant et al., 2018] Grant, E., Finn, C., Levine, S., Darrell, T., and Griffiths, T. (2018). Recasting gradient-based meta-learning as hierarchical bayes. *arXiv preprint arXiv:1801.08930*.

[Graves, 2011] Graves, A. (2011). Practical variational inference for neural networks. *Advances in neural information processing systems*, 24.

[Grosse and Martens, 2016] Grosse, R. and Martens, J. (2016). A kronecker-factored approximate fisher matrix for convolution layers. In *International Conference on Machine Learning*, pages 573–582. PMLR.

[Grosse and Salakhudinov, 2015] Grosse, R. and Salakhudinov, R. (2015). Scaling up natural gradient by sparsely factorizing the inverse fisher matrix. In *International Conference on Machine Learning*, pages 2304–2313. PMLR.

[He et al., 2015] He, K., Zhang, X., Ren, S., and Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034.

[He et al., 2016] He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778.

[Hochreiter and Schmidhuber, 1997] Hochreiter, S. and Schmidhuber, J. (1997). Flat minima. *Neural computation*, 9(1):1–42.

[Hoffman and Ribak, 1991] Hoffman, Y. and Ribak, E. (1991). Constrained realizations of gaussian fields-a simple algorithm. *The Astrophysical Journal*, 380:L5–L8.

[Immer et al., 2021] Immer, A., Bauer, M., Fortuin, V., Rätsch, G., and Khan, M. E. (2021). Scalable marginal likelihood estimation for model selection in deep learning. *arXiv preprint arXiv:2104.04975*.

[Ioffe and Szegedy, 2015] Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. PMLR.

[Karakida et al., 2021] Karakida, R., Akaho, S., and Amari, S.-i. (2021). Pathological spectra of the fisher information metric and its variants in deep neural networks. *Neural Computation*, 33(8):2274–2307.

[Kingma and Ba, 2014] Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

[Kirkpatrick et al., 2017] Kirkpatrick, J., Pascanu, R., Rabinowitz, N., Veness, J., Desjardins, G., Rusu, A. A., Milan, K., Quan, J., Ramalho, T., Grabska-Barwinska, A., et al. (2017). Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences*, 114(13):3521–3526.

[Krizhevsky, 2009] Krizhevsky, A. (2009). Learning multiple layers of features from tiny images. Technical report.

[Kunstner et al., 2019] Kunstner, F., Balles, L., and Hennig, P. (2019). Limitations of the empirical fisher approximation for natural gradient descent. *arXiv preprint arXiv:1905.12558*.

[LeCun, 1998] LeCun, Y. (1998). The mnist database of handwritten digits. *http://yann. lecun. com/exdb/mnist/*.

[LeCun et al., 1990] LeCun, Y., Denker, J. S., and Solla, S. A. (1990). Optimal brain damage. In *Advances in neural information processing systems*, pages 598–605.

[MacKay, 1992] MacKay, D. J. (1992). A practical bayesian framework for backpropagation networks. *Neural computation*, 4(3):448–472.

[Marceau-Caron and Ollivier, 2016] Marceau-Caron, G. and Ollivier, Y. (2016). Practical riemannian neural networks. *arXiv preprint arXiv:1602.08007*.

[Martens, 2014] Martens, J. (2014). New insights and perspectives on the natural gradient method. *arXiv preprint arXiv:1412.1193*.

[Martens et al., 2018] Martens, J., Ba, J., and Johnson, M. (2018). Kronecker-factored curvature approximations for recurrent neural networks. In *International Conference on Learning Representations*.

[Martens et al., 2021] Martens, J., Ballard, A., Desjardins, G., Swirszcz, G., Dalibard, V., Sohl-Dickstein, J., and Schoenholz, S. S. (2021). Rapid training of deep neural networks without skip connections or normalization layers using deep kernel shaping. *arXiv preprint arXiv:2110.01765*.

[Martens et al., 2010] Martens, J. et al. (2010). Deep learning via hessian-free optimization. In *ICML*, volume 27, pages 735–742.

[Martens and Grosse, 2015] Martens, J. and Grosse, R. (2015). Optimizing neural networks with kronecker-factored approximate curvature. In *International conference on machine learning*, pages 2408–2417. PMLR.

[Nguyen et al., 2017] Nguyen, C. V., Li, Y., Bui, T. D., and Turner, R. E. (2017). Variational continual learning. *arXiv preprint arXiv:1710.10628*.

[Ober and Aitchison, 2021] Ober, S. W. and Aitchison, L. (2021). Global inducing point variational posteriors for bayesian neural networks and deep gaussian processes. In *International Conference on Machine Learning*, pages 8248–8259. PMLR.

[Ollivier, 2015] Ollivier, Y. (2015). Riemannian metrics for neural networks i: feedforward networks. *Information and Inference: A Journal of the IMA*, 4(2):108–153.

[Ollivier, 2017] Ollivier, Y. (2017). True asymptotic natural gradient optimization. *arXiv preprint arXiv:1712.08449*.

[Ollivier, 2018] Ollivier, Y. (2018). Online natural gradient as a kalman filter. *Electronic Journal of Statistics*, 12(2):2930–2961.

[Osawa et al., 2019] Osawa, K., Tsuji, Y., Ueno, Y., Naruse, A., Yokota, R., and Matsuoka, S. (2019). Large-scale distributed second-order optimization using kronecker-factored approximate curvature for deep convolutional neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 12359–12367.

[Pascanu and Bengio, 2013] Pascanu, R. and Bengio, Y. (2013). Revisiting natural gradient for deep networks. *arXiv preprint arXiv:1301.3584*.

[Paszke et al., 2019] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. (2019). Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32:8026–8037.

[Polyak and Juditsky, 1992] Polyak, B. T. and Juditsky, A. B. (1992). Acceleration of stochastic approximation by averaging. *SIAM journal on control and optimization*, 30(4):838–855.

[Ren and Goldfarb, 2019] Ren, Y. and Goldfarb, D. (2019). Efficient subsampled gauss-newton and natural gradient methods for training neural networks. *arXiv preprint arXiv:1906.02353*.

[Ritter et al., 2018a] Ritter, H., Botev, A., and Barber, D. (2018a). Online structured laplace approximations for overcoming catastrophic forgetting. *arXiv preprint arXiv:1805.07810*.

[Ritter et al., 2018b] Ritter, H., Botev, A., and Barber, D. (2018b). A scalable laplace approximation for neural networks. In *6th International Conference on Learning Representations, ICLR 2018-Conference Track Proceedings*, volume 6. International Conference on Representation Learning.

[Roux et al., 2007] Roux, N., Manzagol, P.-a., and Bengio, Y. (2007). Topmoumoute online natural gradient algorithm. *Advances in Neural Information Processing Systems*, 20:849–856.

[Sutskever et al., 2013] Sutskever, I., Martens, J., Dahl, G., and Hinton, G. (2013). On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pages 1139–1147. PMLR.

[Wu et al., 2017] Wu, Y., Mansimov, E., Grosse, R. B., Liao, S., and Ba, J. (2017). Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation. *Advances in neural information processing systems*, 30:5279–5288.

[Xiao et al., 2017] Xiao, H., Rasul, K., and Vollgraf, R. (2017). Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms.

[Zhang et al., 2019] Zhang, G., Martens, J., and Grosse, R. (2019). Fast convergence of natural gradient descent for overparameterized neural networks. *arXiv preprint arXiv:1905.10961*.

[Zhang et al., 2018a] Zhang, G., Sun, S., Duvenaud, D., and Grosse, R. (2018a). Noisy natural gradient as variational inference. In *International Conference on Machine Learning*, pages 5852–5861. PMLR.

[Zhang et al., 2018b] Zhang, G., Wang, C., Xu, B., and Grosse, R. (2018b). Three mechanisms of weight decay regularization. *arXiv preprint arXiv:1810.12281*.