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

Frederik Benzing [1]

Abstract

Second-order optimizers are thought to 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). Here, we combine tools from prior work to evaluate exact second-order updates with careful ablations to establish a surprising result: Due to its approximations, KFAC is not closely related to second-order updates, and in particular, it significantly outperforms true second-order updates. This challenges widely held believes and immediately raises the question why KFAC performs so well. We answer this question by showing that KFAC approximates a first-order algorithm, which performs gradient descent on neurons rather than weights. Finally, we show that this optimizer often 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. Originally proposed in the context of optimization (Martens and Grosse, 2015), where they have lead to many further developments (Grosse and Martens, 2016; Ba et al., 2016; Desjardins et al., 2015; Botev et al., 2017; George et al., 2018; Martens et al., 2018; Osawa et al., 2019; Bernacchia et al., 2019; Goldfarb et al., 2020), they have also proven influential in various other contexts like Bayesian inference, meta learning and continual learning (Ritter et al., 2018a;b; Dangel et al., 2020; Zhang et al., 2018a; Wu et al., 2017; Grant et al., 2018).

Here, we describe a surprising discovery: Despite its motivation, the KFAC optimizer does not rely on second-order information; in particular it significantly outperforms exact second-order optimizers. We establish these claims through a series of careful ablations and control experiments and build on prior work, which shows that exact second-order updates can be computed efficiently and exactly, if the dataset is small or when the curvature matrix is subsampled (Ren and Goldfarb, 2019; Agarwal et al., 2019).

Our finding that KFAC does not rely on second-order information immediately raises the question why it is nevertheless so effective. To answer this question, we show that KFAC approximates a different, first-order optimizer, which performs gradient descent in neuron- rather than weight space. We also show that this optimizer itself further improves upon KFAC, both in terms of computational cost as well as progress per parameter update.

Structure of the Paper. In Section 2 we provide background and define our terminology . 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.

[1]Department of Computer Science, ETH Zurich, 8092 Zurich, Switzerland. Correspondence to: Frederik Benzing <benzingfr@gmail.com>.

2. Background and Efficient Subsampled Natural Gradients

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 the (Generalised) Gauss Newton matrix 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 $\mathbf{F}$ is defined as

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

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 typically viewed as a second-order method (Martens and Grosse, 2015).

2.1. Subsampled, Exact Natural Gradients

If the dataset is moderately small, or if the Fisher is subsampled, i.e. evaluated on a mini-batch, then natural gradients can be computed exactly and efficiently as shown by (Ren and Goldfarb, 2019) with ideas described independently by (Agarwal et al., 2019). The key insight is to apply the Woodbury matrix inversion lemma and to realise that many intermediate quantities do not need to be stored or computed explicitly. We propose some modest theoretical as well as practical improvements to these techniques, which are deferred to Appendix I along with implementation details.

We also show that with an additional trick (Doucet, 2010; Hoffman and Ribak, 1991), one can sample efficiently and exactly from the Laplace posterior, see Appendix C.

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

2.2. 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 nonlinearity) 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). Note that with this definition, the total number of datapoints is the product of the number of inputs and the number of labels.

Following (Martens and Grosse, 2015), the Fisher will usually be approximated by sampling one label for each input. If 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**.

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 hyperparameter λ and $\frac{\lambda_A}{\lambda_E} = \frac{n \cdot \mathrm{Tr}(\mathbf{A}\mathbf{A}^T)}{m \cdot \mathrm{Tr}(\mathbf{E}_F \mathbf{E}_F^T)}$.

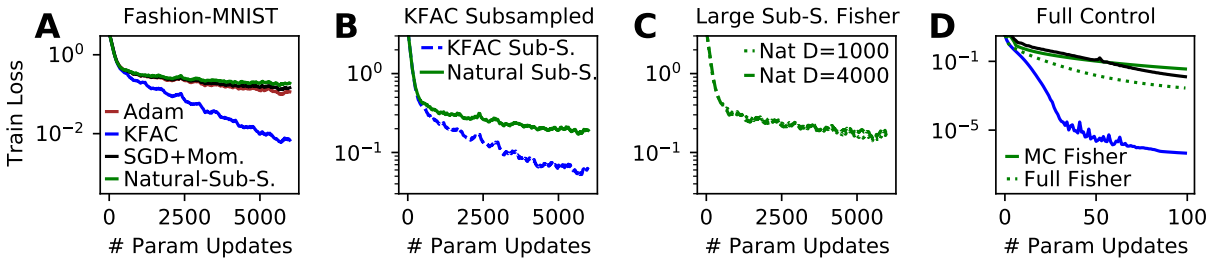

Figure 1. **Paradoxically, KFAC – an approximate second-order method – outperforms exact second-order udpates in standard as well as important control settings. (A)** Comparison between Subsampled Natural Gradients and KFAC. KFAC performs significantly better. Theoretically, its only advantage over the subsampled method is using more data to estimate the curvature. All methods use a batchsize of 100 and are trained for 10 epochs, with hyperparameters tuned individually for each method (here and in all other experiments). **(B)** Comparison between Subsampled Natural Gradients and Subsampled KFAC. Both algorithms use exactly the same amount of data to estimate the curvature. From a theoretical viewpoint, KFAC should be a strictly worse approximation of second-order updates than the exact subsampled method; nevertheless, it performs significantly better. **(C)** Additional control in which the subsampled Fisher is approximated on larger mini-batches. **(D)** Full control setting, in which the training set is restricted to 1000 images and gradients and curvature are computed on the entire batch. The dashed green line corresponds to exact natural gradients without any approximations. Consistent with prior literature, full second-order updates do outperform standard first-order updates (dashed green vs. black line). More importantly, and very surprisingly, KFAC significantly outperforms exact second-order updates. This is very strong evidence that KFAC is not closely related to Natural Gradients. **(A-D)** We repeated several key experiments with other datasets and architectures and results were consistent with the ones seen here, see main text and appendix.

Heuristic 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. To make this concrete, the two strategies use the following damped curvature matrices

$$\text{correct:} \quad \left(\mathbf{A}\mathbf{A}^T \otimes \mathbf{E}_F\mathbf{E}_F^T\right) + \lambda\mathbf{I} \tag{3}$$

$$\text{heuristic:} \quad \left(\mathbf{A}\mathbf{A}^T + \lambda_A\mathbf{I}\right) \otimes \left(\mathbf{E}_F\mathbf{E}_F^T + \lambda_E\mathbf{I}\right) \tag{4}$$

Heuristic damping adds undesired cross-terms $\lambda_E\mathbf{A}\mathbf{A}^T \otimes \mathbf{I}$ and $\lambda_A\mathbf{I} \otimes \mathbf{E}_F\mathbf{E}_F^T$ to the curvature, and we point out that these cross terms are typically much larger than the desired damping $\lambda\mathbf{I}$. While the difference in damping may nevertheless seem innocuous, Martens and Grosse (2015); Ba et al. (2016); George et al. (2018) all explicitly state that heuristic damping performs better than correct damping. From a theoretical perspective, this is a rather mysterious observation.

In practice, the Kronecker factors $\mathbf{A}\mathbf{A}^T$ and $\mathbf{E}_F\mathbf{E}_F^T$ are updated as exponential moving averages, so that they incorporate data from several recent mini-batches.

Subsampled Natural Gradients vs KFAC: There are two high level differences between KFAC and subsampled natural gradients. (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 additional 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 carried out on a fully connected network on Fashion MNIST (Xiao et al., 2017) and followed by results on a Wide ResNet (He et al., 2016) on CIFAR10 (Krizhevsky, 2009). We run several additional experiments, which are presented fully in the appendix, and will be refered to in the main text. These include repeating the first set of experiments on MNIST; results on CIFAR100; a VGG network (Simonyan and Zisserman, 2014) trained on SVHN (Netzer et al., 2011) and more traditional autoencoder experiments (Hinton and Salakhutdinov, 2006).

We emphasise that, while our results are surprising, they are certainly not caused by insufficient hyperparameter tuning or incorrect computations of second-order updates. In particular, we perform independent grid searches for each method and ablation and make sure that the grids are sufficiently wide and fine. Details are given in Appendix D and A and we describe part of our software validation in B. Code to validate and run the software is provided[1]. Moreover, as will be pointed out throughout the text, our results are consistent with many experiments from prior work.

To obtain easily interpretable results without unnecessary confounders, we choose a constant step size for all methods, and a constant damping term. This matches the setup of prior work (Desjardins et al., 2015; Zhang et al., 2018b; George et al., 2018; Goldfarb et al., 2020). We re-emphasise that these hyperparameters are optimized carefully and indepently for each method and experiment individually.

[1] https://github.com/freedbee/Neuron_Descent_and_KFAC

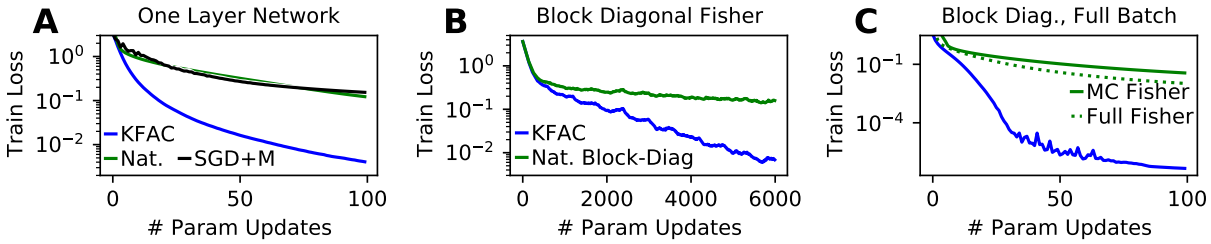

Figure 2. **Advantage of KFAC over exact, subsampled Natural Gradients is not due to block-diagonal structure. (A)** A one layer network (i.e. we perform logistic regression) is trained on 1000 images and full batch gradients are used. In particular, KFAC and the subsampled method use the same amount of data to estimate the curvature. 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. **(A-C)**Experiments on Fashion MNIST, results on MNIST are analogous, see appendix.

Following the default choice in the KFAC literature (Martens and Grosse, 2015), we usually use a Monte Carlo estimate of the Fisher, based on sampling one label per input. We will also carry out controls with the Full Fisher.

Performance: We first investigate the performance of KFAC and subsampled natural gradients, see Figure 1A. 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 over subsampled natural gradients 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 hypothesis, 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 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 (MC Fisher). In particular, we evaluate exact natural gradients (without any approximations: The gradient is exact, the Fisher is exact and the inversion is exact). The results are shown in Figure 1 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, also when the latter can be computed without any approximations.

This first finding is very surprising. Nevertheless, we point out that it is consistent with experimental results from prior work as well as commonly held beliefs. Firstly, it is widely believed that subsampling natural gradients leads to poor performance. This belief is partially evidenced by claims from (Martens et al., 2010) and often mentioned in informal discussions and reviews. It matches our findings and in particular Figure 1D, which shows that benefits of Natural Gradients over SGD only become notable when computing the Fisher fully.[2] Secondly, we have shown that KFAC performs well even when it is subsampled in the same way as we subsampled natural gradients. While this does seem to contradict the belief that natural gradient methods should not be subsampled, it is confirmed by experiments from (Botev et al., 2017; Bernacchia et al., 2019): See Figure 2 "per iteration curvature" in (Botev et al., 2017) and note that in (Bernacchia et al., 2019) the curvature is evaluated on individual minibatches.

Additional Experiments: We repeat the key experiments from Figure 1A,B in several additional settings: On a MLP on MNIST, on a ResNet with and without batch norm on CIFAR10 and for traditional autoencoder experiments. The findings are in line with the ones above, and solidify concerns whether KFAC is related to second-order information.

Controlling for Block-Diagonal Structure: This begs further investigation into why KFAC outperforms natural gradients. KFAC approximates the Fisher as block-diagonal. To test whether this explains KFAC's advantage, 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

[2]It also evidences the correctness of our implementation of natural gradients.

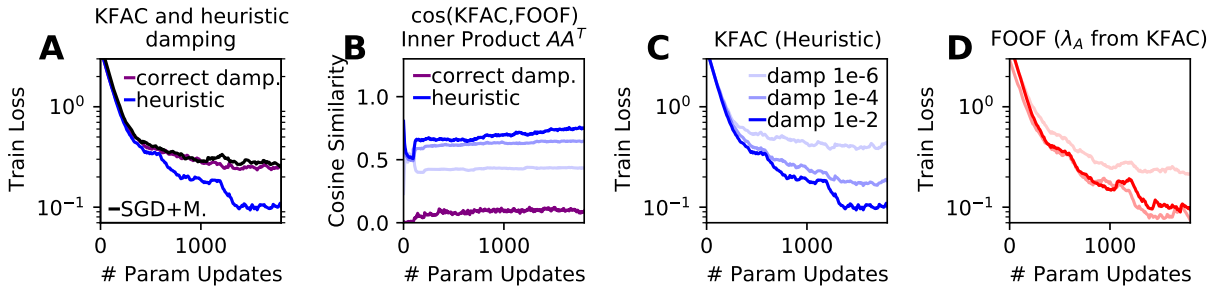

Figure 3. **Heuristic Damping increases KFAC's performance as well as its similarity to first-order method FOOF. (A)** Heuristic damping is strictly needed for performance of KFAC; with standard damping, KFAC performs similar to SGD. **(B)** Heuristic damping significantly increases similarity of KFAC to FOOF. For the inner product space, we use the "curvature" matrix of 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.8. Notably, FOOF already works well for lower damping terms than KFAC, suggesting that KFAC requires larger damping mainly to guarantee similarity to FOOF and limit the effect of its second Kronecker factor. **(A-D)** and our theoretical analysis suggest that KFAC owes its performance to similarity to the first-order method FOOF. Experiments are on Fashion-MNIST, results on MNIST are analogous, see appendix. We also re-run experiment (A) in several other settings and confirm that heuristic damping is crucial for performance, see Appendix. This is in line with reports from (Martens and Grosse, 2015; Ba et al., 2016; George et al., 2018).

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 2A. 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 the subsampled natural gradient method. We run the block-diagonal natural gradient algorithm in two settings: In a minibatch 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 2B,C confirm our previous 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.

Heuristic Damping. KFAC also deviates from exact second-order updates through its heuristic damping. To test whether this difference explains KFAC's performance, we implemented a version of KFAC with correct damping.[3] Figure 3A shows that KFAC owes essentially all of its performance to the damping heuristic. This finding is confirmed by experiments on CIFAR10 with a ResNet and on autoencoder experiments. We re-emphasise that KFAC outperforms exact natural gradients, and therefore the damping heuristic cannot be seen as giving a better approximation of second-order updates. Rather, heuristic damping causes performance benefits through some other effect.

Summary. We have seen that KFAC, despite its motivation as an approximate natural gradient method, behaves very differently from true natural gradients. In particular, and surprisingly, KFAC drastically outperforms natural gradients. Through a set of careful controls, we established that KFAC's advantage relies on a seemingly innocuous damping heuristic, which is unrelated to second-order information. We now turn to why this is the case.

4. First-order Descent on Neurons

We will first describe the optimizer "Fast First-Order Optimizer" or "FOOF"[4] and then explain KFAC's link to it.

FOOF's update rule is similar to some prior work (Desjardins et al., 2015; Frerix et al., 2017; Amid et al., 2021). The view on optimization which underlies FOOF is principled and new, and, among other differences, our insights and experiments linking KFAC to FOOF are new. For a more detailed discussion see Appendix H.2

Recall our notation for one layer of a neural network from Section 2.2, 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

$$\min_{\Delta\mathbf{W}\in\mathbb{R}^{n\times m}} \|(\Delta\mathbf{W})\mathbf{A} - \eta\mathbf{E}\|^2 + \frac{\lambda}{2}\|\Delta\mathbf{W}\|^2 \qquad (5)$$

where the second summand $\frac{\lambda}{2}\|\Delta\mathbf{W}\|^2$ is a proximity constraint limiting the update size. (5) is a linear regression

[3] This can be done with ideas from (George et al., 2018).

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

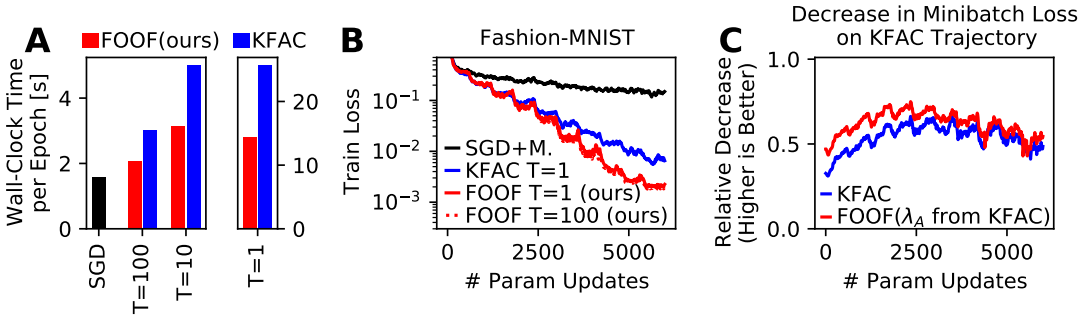

Figure 4. **FOOF outperforms KFAC in terms of both per-update progress and computation cost. (A)** Wall-clock time comparison between FOOF, KFAC and SGD. T denotes how frequently matrix inversions (see eq (6)) are performed. Implemented with PyTorch and run on a GPU. Increasing T above 100 does not notably improve runtime. FOOF is approximately 1.5x faster than KFAC. **(B)** Training loss on Fashion MNIST. FOOF is more data efficient and stable than KFAC. **(C)** Comparison of KFAC and a version of KFAC which drops the second Kronecker factor (equivalently, this corresponds to FOOF with damping term λ_A from KFAC). We follow the trajectory of KFAC, and at each point we compute the relative improvement of the loss on the current mini-batch that is achieved by the update of KFAC and FOOF, respectively. We use the learning rate and damping that is optimal for KFAC, and scale the FOOF update to have the same norm as the KFAC update at each layer. FOOF performs better, further suggesting that similarity to FOOF is responsible for KFAC's performance.

problem (for each row of $\Delta\mathbf{W}$) solved by

$$(\Delta\mathbf{W})^T = \eta \left(\lambda\mathbf{I} + \mathbf{A}\mathbf{A}^T\right)^{-1} \mathbf{A}\mathbf{E}^T \tag{6}$$

Pseudocode for the resulting optimizer FOOF is presented in Appendix K. Figures 4,5 show the empirical results of FOOF, which outperforms not only SGD and Adam, but also KFAC. An intuition for why "gradient descent on neurons" performs considerably better than "gradient descent on weights" 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 illustratitive toy example for this intuition.

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

4.1. Stochastic Version of FOOF and Amortisation

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. 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 shown in Figure 4.

FOOF can be combined with momentum and decoupled weight decay.

4.2. KFAC as First-Order Descent on Neurons

Recall that the KFAC update is given by

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

Similarity of KFAC to FOOF and Damping: The update of KFAC differs from the FOOF update (eq (6)) only through the second factor $\left(\mathbf{E}_F\mathbf{E}_F^T + \lambda_E\mathbf{I}\right)^{-1}$. We emphasise that this similarity is induced mainly through the heuristic damping strategy. In particular, with correct damping, or without damping, the second Kronecker factor of KFAC could lead to updates that are essentially uncorrelated with FOOF. However, as we use heuristic damping and increase the damping strength λ_E, 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 3B 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 3C. These findings, in particular the necessity to use heuristic damping, already strongly suggest that similarity to FOOF is required for KFAC to perform well. Moreover, as shown in Figure 3D, FOOF requires lower damping than KFAC to perform well. This further suggests that damping in KFAC is strictly required to limit the effect of $\mathbf{E}_F\mathbf{E}_F^T$ on the update, thus increasing similarity to FOOF. All in all, these results directly support the claim 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

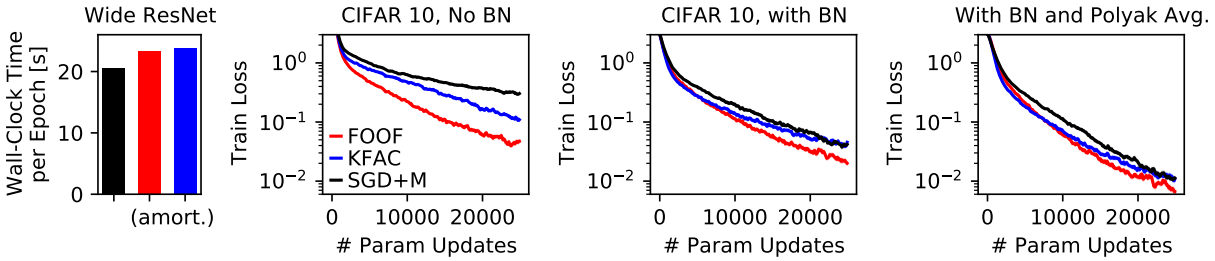

Figure 5. **FOOF outperforms KFAC in a Wide ResNet18 on CIFAR 10. (A)** Wall-clock time comparison between SGD and amortised versions of FOOF, KFAC. In convolutional architectures, FOOF and KFAC can be effectively amortised wihtout sacrificing performance. **(B, C, D)** Training loss in different settings. **(A-D)** Results on CIFAR100 and SVHN are analogous.

expect FOOF to perform better than or similarly to KFAC. We carry out two different tests of this hypothesis. First, we train a network using KFAC and at each iteration, we record the progress KFAC makes on the given mini-batch, measured as the relative decrease in loss. We compare this to the progress that KFAC would have made without its second Kronecker factor. We use learning rate and damping that are optimal for KFAC and, when dropping the second factor, we rescale the update to have the same norm as the original KFAC update for each layer. The results are shown in Figure 4C and show that without the second factor, KFAC makes equal or more progress, which supports our hypothesis. This observation is consistent across all different experimental setups we investigated, see appendix.
As a second test, we check whether FOOF outperforms KFAC, when both algorithms follow their own trajectory. This is indeed the case as shown in Figure 4B. The only case where the advantage described in Figure 4C does not translate to an overall advantage is the autoencoder setting, as analysed in the appendix. Results on a Wide ResNet18 demonstrate that our findings carry over to more complex settings and that FOOF outperforms KFAC, see 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 4A.

Cost in Convolutional Architectures: The only overhead of KFAC and FOOF which cannot be amortised is performing the matrix multiplications in eqs (6),(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 ($\sim$10% increase for FOOF, $\sim$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.[5]

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

5. 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 a different experimental setup (as demonstrated in (Zhang et al., 2018b)) and is left to future work.

We have restricted our investigation 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.

6. Discussion

The purpose of this discussion is twofold. On the one hand, we will show that, while being surprising and contradicting common, strongly held beliefs, much of our results are consistent with data from prior work. On the other hand, we will summarise in how far our fundamentally new explanation for KFAC's effectiveness improves upon prior knowledge and resolves several puzzling observations.

Natural Gradients vs KFAC: Our first key result is that KFAC outperforms exact natural gradients. We perform several controls and a particularly important set of experiments is comparing exact, subsampled natural gradients to subsampled KFAC in a range of settings. In these experiments, we

[5]Information reconstructed from Figure 3 in Ba et al. (2016) and Figure 4 in Desjardins et al. (2015).

find: (1) Subsampled natural gradients often do not perform much better than SGD with momentum. (2) Subsampled KFAC works very well. Finding (1) is consistent with rather common beliefs that subsampling the curvature is harmful. These beliefs are often uttered in informal discussions and are partially evidenced by claims from (Martens et al., 2010). Moreover, in one control (Fig 1D), we show that full (non-subsampled) natural gradients do outperform SGD with momentum, consistent for example with (Martens et al., 2010). Thus, finding (1) is in line with prior knowledge and results. Moreoever, finding (2) matches experiments from (Botev et al., 2017; Bernacchia et al., 2019) as described in Section 3 and thus also is in line with prior work.

Completely independent of our results, it is worth noting that the performance of subsampled KFAC reported in (Botev et al., 2017; Bernacchia et al., 2019) is hard to reconcile with the convictions that (1) KFAC is a natural gradient method and (2) subsampling the Fisher has detrimental effects.

Our explanation of KFACs performance resolves this contradiction. Even if one were to disagree with our explanation for KFAC's effectiveness, the above is an important insight, strengthened by our careful control experiments, and deserves further attention.

It is also worth noting that, a priori, the most natural way to check if our finding that KFAC outperforms Natural Gradients agrees with prior work would be to look for a direct comparison of KFAC with Hessian-Free optimization (HF). Given that both these algorithms have been well known for several years, one might expect there to be several direct comparisons. However, to the best of our knowledge, there is no meaningful comparison between these two algorithms in the literature.[6] If our results are to be reproduced independently, it will be interesting to see a thoroughly controlled, well tuned comparison between HF and KFAC.

Damping: A second cornerstone of our study is the effect of damping on KFAC. We found that employing a heuristic, rather than standard damping strategy is essential for performance. The result that heuristic damping improves KFAC's performance has been noted several times (qualitatively, rather than quantitatively), see the original KFAC paper (Martens and Grosse, 2015), its large-scale follow up (Ba et al., 2016), and even E-KFAC (George et al., 2018).

While choice of damping strategy may seem like a negligible detail at first, it is important to bear in mind that without heuristic damping KFAC performs like standard first-order optimizers like SGD or Adam. Thus, if we want to understand KFAC's effectiveness, we have to account for its damping strategy. This is achieved by our new explanation and even if one were to disagree with it, this finding deserves further attention and an explanation.

Ignoring the second Kronecker factor $\mathbf{E}_F\mathbf{E}_F^T$ of the approximate Fisher: A third important finding is that KFAC performs well, and usually better, without its second Kronecker factor. The fact that algorithms that are similar to KFAC without the second factor perform exceptionally well is consistent with prior work (Desjardins et al., 2015; Frerix et al., 2017; Amid et al., 2021). Moreover, (Desjardins et al., 2015) explicitly state that their algorithm performs more stably without the second Kronecker factor, which further confirms our findings. We emphasise that, without the second Kronecker factor, the preconditioning matrix of KFAC is a zeroth-order quantity, completely independent of the loss, and thus cannot be seen as a second-order method. From a second-order viewpoint, dropping dependence on the loss should have detrimental effects, inconsistent with results from the studies above as well as ours.

Architectures with Parameter Sharing: Finally, we point to another intriguing finding from (Grosse and Martens, 2016). For architectures with parameter sharing, like CNNs or RNNs, approximating the Fisher by a Kronecker product requires additional, sometimes complex assumptions, which are not always satisfied. (Grosse and Martens, 2016) explicitly investigate one such assumption for CNNs, pointing out that it is violated in architectures with average- rather than max-pooling. Nevertheless, KFAC performs very well in such architectures (Ba et al., 2016; George et al., 2018; Zhang et al., 2018b; Osawa et al., 2019). This suggests that KFAC works well independently of how closely it is related to the Fisher, which is a rather puzzling observation, when viewing KFAC as a natural gradient method. Again, our new explanation resolves this issue, since KFAC still performs gradient descent on neurons.[7]

In summary, we have shown that viewing KFAC as a second-order, natural gradient method is irreconcilable with a host of experimental results, from our as well as other studies. We then proposed a new, considerably improved explanation for KFAC's effectiveness. We also showed that the algorithm FOOF, which results from our explanation, gives further performance improvements compared to the state-of-the art optimizer KFAC.

References

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, L. (2018). Bayesian filtering unifies adaptive

[6] We are aware of only one comparison, which unfortunately has serious limitations, see Appendix H.1.1.

[7] In eq (5), we now impose one constraint per datapoint and per "location" at which the parameters are applied.

and non-adaptive neural network optimization methods. *arXiv preprint arXiv:1807.07540*.

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

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

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

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

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

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

Benzing, F., Gauy, M. M., Mujika, A., Martinsson, A., and Steger, A. (2019). Optimal kronecker-sum approximation of real time recurrent learning. In *International Conference on Machine Learning*, pages 604–613. PMLR.

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

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

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

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

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

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

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, 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, 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, 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.

Hinton, G. E. and Salakhutdinov, R. R. (2006). Reducing the dimensionality of data with neural networks. *science*, 313(5786):504–507.

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

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

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

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

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

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

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

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

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

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

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

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

Mujika, A., Meier, F., and Steger, A. (2018). Approximating real-time recurrent learning with random kronecker factors. *arXiv preprint arXiv:1805.10842*.

Netzer, Y., Wang, T., Coates, A., Bissacco, A., Wu, B., and Ng, A. Y. (2011). Reading digits in natural images with unsupervised feature learning.

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

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

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

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

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

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.

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

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

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

Simonyan, K. and Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

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.

Tallec, C. and Ollivier, Y. (2017). Unbiased online recurrent optimization. *arXiv preprint arXiv:1702.05043*.

Williams, R. J. and Zipser, D. (1995). Gradient-based learning algorithms for recurrent. *Backpropagation: Theory, architectures, and applications*, 433:17.

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

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

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, 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. For our Fashion MNIST and MNIST experiments, 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 large 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.

Moreover, the fact that natural gradients outperform SGD in Figure 1D (and some other settings) is very strong evidence that our implementation of natural gradients is correct.

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

$$\mathbf{\Lambda}^{-1} = \left(\mathbf{\Lambda}_{\text{prior}} + D\mathbf{G}\mathbf{G}^T\right)^{-1} = \mathbf{\Lambda}_{\text{prior}}^{-1} - D\mathbf{\Lambda}_{\text{prior}}^{-1}\mathbf{G}\left(\mathbf{I} + D\mathbf{G}^T\mathbf{\Lambda}_{\text{prior}}^{-1}\mathbf{G}\right)^{-1}\mathbf{G}^T\mathbf{\Lambda}_{\text{prior}}^{-1} \tag{7}$$

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

$$\mathbf{V} = \left(D^{1/2}\mathbf{\Lambda}_{\text{prior}}^{-1/2}\right)\mathbf{G} \tag{8}$$

$$\mathbf{U} = \mathbf{I} + \mathbf{V}^T\mathbf{V} = \mathbf{I} + D\mathbf{G}^T\mathbf{\Lambda}_{\text{prior}}^{-1}\mathbf{G} \tag{9}$$

$$\tag{10}$$

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

$$\mathbf{x} = \mathbf{y} - \mathbf{V}\mathbf{U}^{-1}\left(\mathbf{V}^T\mathbf{y} + \mathbf{z}\right) \tag{11}$$

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

$$\mathbb{E}\left[\mathbf{x}\mathbf{x}^T\right] = \mathbf{I} + \mathbf{V}\mathbf{U}^{-1}\left(\mathbf{V}^T\mathbf{V} + \mathbf{I}\right)\mathbf{U}^{-T}\mathbf{V}^T - 2\mathbf{V}\mathbf{V}^T \tag{12}$$

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

$$\mathbb{E}\left[\mathbf{x}\mathbf{x}^T\right] = \mathbf{I} - \mathbf{V}\mathbf{U}^{-1}\mathbf{V}^T \tag{13}$$

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

$$\mathbb{E}\left[\mathbf{x}\mathbf{x}^T\right] = \mathbf{I} - D\mathbf{\Lambda}_{\text{prior}}^{-1/2}\mathbf{G}\left(\mathbf{I} + D\mathbf{G}^T\mathbf{\Lambda}_{\text{prior}}^{-1}\mathbf{G}\right)^{-1}\mathbf{G}^T\mathbf{\Lambda}_{\text{prior}}^{-1/2} \tag{14}$$

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 Implicit 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, the method used here is directly applicable to compute this product.

D. Experimental Details

All experiments were implemented in PyTorch (Paszke et al., 2019).[8] All models use Kaiming-He initialisation (He et al., 2015; Glorot and Bengio, 2010). Moreover, we average all results across three different random seeds.

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.

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 Natural Gradients, KFAC, FOOF were determined by a grid searcher over $10^{-6}, 10^{-4}, 10^{-2}, 10^0, 10^2, 10^4, 10^6$ on Fashion MNIST and MNIST. We also confirmed that refining the grid to values of the form 10^i does not meaningfully change results. For all other experiments we used a grid with values of the form 10^i, but also there a coarser grid with values 10^{2i} would have given very similar results.

We extended grids if the performance at a boundary of the grid was optimal (or near optimal).

As already described in Appendix A, we did spend additional efforts to tune the natural gradient method, including hand-tuning, as well as using additional ingredients like a fancy, second-order form of momentum, automatic learning rates, adaptive damping, see Appendix A.

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 ($T = 1$). In particular, for plots like Figure 3C, the learning rate for each damping term was tuned individually.

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.

[8] So long and thanks for all the hooks.

- Step size: Tuning the stepsize of FOOF was as easy/difficult as tuning the step size 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.
- 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 2A, 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.

When experimenting with the ResNet with batchnorm, we found that KFAC was unstable with standard momentum. To fix this, we switched on the momentum term only after the first epoch was completed. This seemed helped all methods a bit, so we used it for all of them. We used the same setting for the ResNet without batchnorm.

When applicable, 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.

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. Autoencoder Experiments

We tried to replicate the setup of (Martens and Grosse, 2015) implemented in the tensorflow kfac repository[9]. Concretly, we use a fully connected network with layers of sizes (d, 1000, 500, 250, 30, 250, 500, 1000, d), where d is the number of input pixels of a single image (which depends on the dataset). All *hidden* layers, expect for the middle one (of size 30) are

[9] https://github.com/tensorflow/kfac/blob/master/kfac/examples/autoencoder_mnist.py

followed by a tanh-nonlinearity. We preprocessed each dataset to have zero mean and unit variance. We used a batch size of 1000, since (Martens and Grosse, 2015) state that small batch sizes lead to too much noise. For MNIST and CURVES we trained for 200 epochs, for FACES for 100 epochs. To amortise the runtime, we invert matrices every 10 steps.

D.7. SVHN

We used the same preprocessing for SVHN as for CIFAR10. To amortise runtime we used T=100, S=50 (see pseudocode 1). We confirmed that with T=S=10 results are essentially the same, suggesting that the amortisation did not harm performance. We used a VGG11 network (without batch norm).

D.8. Version of FOOF with λ_A from KFAC

In Figures 3,4, we use a version of FOOF which always uses exactly the same damping term λ_A for each layer 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 $\mathbf{A}\mathbf{A}^T$, while omitting the second factor. Note that λ_A varies during training, and typically increases so that this version of FOOF is slightly different from standard FOOF.

Moreover, omitting the second factor notably changes the update size. To correct for this effect, we made the following modifiaction.

In Figure 4C, we rescaled the resulting update (as also described in the figure caption) so that at each layer, the norm of the update had the same norm as the KFAC update.

In the experiment of Figure 3, which we did first, we used a slightly different strategy (but we do not think that this makes a difference). 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

Here, we re-derive KFAC (Martens and Grosse, 2015).

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:

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

In general, this approximation is imprecise: It is equivalent to approximating a rank $|I|$ matrix by a rank 1 matrix[10]. 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 convolutional networks this is even more pronounced: The full rank of the Fisher is multiplied by the number of locations at which the filer is applied, while the approximation remains at rank 1.

[10]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 $|I|$.

So in general, this approximation does not hold. In the literature it is usually justified by an independence assumption. Concretely, we view $\mathbf{a}_i\mathbf{a}_i^T, \mathbf{e}_i\mathbf{e}_i^T$ as random variables, where the randomness jointly depends on which datapoint we draw. We then assume that these random variables are independent. However, note that in non-degenerate neural networks we will usually be able to uniquely identify which datapoint was used, if we are given $\mathbf{e}_i\mathbf{e}_i^T$ (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 $\mathbf{e}_i$). Thus we can uniquely determine $\mathbf{a}_i\mathbf{a}_i^T$. This implies that the conditional entropy $H(\mathbf{a}_i\mathbf{a}_i^T \mid \mathbf{e}_i\mathbf{e}_i^T) = 0$, in particular $\mathbf{a}_i\mathbf{a}_i^T, \mathbf{e}_i\mathbf{e}_i^T$ are not independent.

We point out that none of our experiments directly checks how accurate this approximation of the Fisher Information is. Our experiments mainly show that heuristic damping breaks the link to the Fisher, but do not give data on the quality of the approximation before heuristic damping. Evaluating the quality of this approximation is an interesting question left to future work.

Relating this to findings of (Bernacchia et al., 2019), note that if we consider linear networks and regression problems with homoscedastic 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 a part of the insights from (Bernacchia et al., 2019) for the analysis of linear networks. The above reasoning also explains precisely where this breaks down for non-linear networks (or hetero-scedastic noise). In particular, in non-linear networks, the errors will be backpropagated through different non-linear functions (given by the activations from the forward pass) and the argument from the linear case breaks down.

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 on both datapoints simultaneously.

The 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 also supports the intuition that FOOFs advantage 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;a) of Kronecker-factorisations of the curvature rely on proximity to the curvature matrix.

We point out that the applications from (Ritter et al., 2018b;a) do not seem to rely on heuristic damping. Also in light of our findings, it remains plausible that without heuristic damping, KFAC is very similar to the Fisher. In other words, the below is a hypothesis, not a certainty. To evaluate it, it would be interesting to compare the performances of KFAC to Full Laplace as well as to the algorithm suggested below.

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 $\mathrm{vec}\,(\mathbf{V})^T\,\mathbf{\Sigma}\mathrm{vec}\,(\mathbf{V})$ 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

$$\mathrm{vec}\,(\mathbf{V})^T\,\mathbf{\Sigma}\mathrm{vec}\,(\mathbf{V}) \approx \mathrm{vec}\,(\mathbf{V})^T\,(\mathbf{A}\mathbf{A}^T \otimes \mathbf{I})\mathrm{vec}\,(\mathbf{V}) = \sum_i \mathbf{v}_i^T(\mathbf{A}\mathbf{A}^T)\mathbf{v}_i \tag{15}$$

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.[11] 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, indirectly supporting our hypothesis.

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; 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 (empirical) 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).

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.

[11] 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.

KFAC faces the problem of approximating a sum of kronecker-products by a single kronecker-product. This problem also occurs when approximating real time recurrent learning of recurrent networks (Williams and Zipser, 1995; Tallec and Ollivier, 2017; Mujika et al., 2018; Benzing et al., 2019) and in this context (Benzing et al., 2019) show how to obtain optimal biased and unbiased approximations. Our results suggest that it is not promising to apply these techniques to approximate natural gradients more accurately.

H.1.1. Comparison between HF and KFAC

The subsampled natural gradient method upon which many of our results rely was first described in (Ren and Goldfarb, 2019). On top of their useful, important theoretical results, they also provide an empirical evaluation of their method, and – to the best of our knowledge – the only published comparison of KFAC and HF.

Unfortunately, there is very strong evidence that all methods considered there are heavily undertuned or that there is another issue. To see this, note that in (Ren and Goldfarb, 2019) a network trained on MNIST with one hidden layer of 500 neurons achieves a training loss of around 0.3 and a test accuracy of less than 95% for all considered optimizers. This is much worse than standard results and clearly not representative of normal neural network training. We quickly verified that in exactly the same setting, with KFAC we are able to obtain a loss which is more than 100x smaller and a test accuracy of 98%.[12]

H.1.2. 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. We point out that the empirical results in non-linear networks from (Bernacchia et al., 2019) essentially amount to a re-discovery of KFAC, as such they do not contradict our results.

For non-linear, strongly 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 (Zhang et al., 2019) do not establish similarity between KFAC and Natural Gradients but rather give two separate convergence proofs.

Both these theoretical results (Bernacchia et al., 2019; Zhang et al., 2018a) do not account for any form of damping, so they have to be seen as independent of the empirically well-performing version of KFAC and our investigation.

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 often not notably outperform SGD.

H.1.3. 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.

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

[12] We used the same network, same activation function and same number of epochs. Weight initialisation scheme, batch size and preprocessing were not described in the original experiments, so we used batch size 100 and the same initialisation and preprocessing as in our other MNIST experiments (which has no data augmentation).

information. In this sense, and bearing in mind our previous experiments, NNN should not be seen as a second-order optimizer and our results offer an explanation 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. In particular, the derivation of (Frerix et al., 2017) gives an alternative perspective on the update equation of FOOF.

Among other differences, (Frerix et al., 2017; Amid et al., 2021) (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.

Note also that the framework from (Ollivier, 2018) can be applied to interpret FOOF as applying Kalman filtering to each layer individually. Thus, in some vague sense, FOOF is bayes-optimal and some readers may find this an enticing explanation for FOOF's strong empirical performance.

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 $\mathbf{W} = \left(\mathbf{W}^0, \dots, \mathbf{W}^{(\ell-1)}\right)$, 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

$$s^{(k)} = \mathbf{W}^{(k-1)}\mathbf{a}^{(k-1)} \qquad \text{for } k = 1, \dots, \ell \tag{16}$$

$$\mathbf{a}^{(k)} = \sigma(\mathbf{s}^{(k)}) \qquad \text{for } k = 1, \dots, \ell - 1 \tag{17}$$

$$f(\mathbf{x}) = f(\mathbf{x}; \mathbf{W}) = \operatorname{softmax}(a^{(\ell)}) \tag{18}$$

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 $(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_B, y_B)$ from the model distribution and use this for a MC estimate of the Fisher, i.e.

$$\mathbf{F} = \frac{1}{B}\sum_{i=1}^{B} \mathbf{g}_i \mathbf{g}_i^T = \mathbf{G}\mathbf{G}^T \tag{19}$$

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

$$(\lambda \mathbf{I} + \mathbf{F})^{-1}\mathbf{u} = (\lambda \mathbf{I} + \mathbf{G}\mathbf{G}^T)^{-1} = \lambda^{-1}\mathbf{I}\mathbf{u} - \lambda^{-2}\mathbf{G}(\mathbf{I} + \frac{1}{\lambda}\mathbf{G}^T\mathbf{G})^{-1}\mathbf{G}^T\mathbf{u} \tag{20}$$

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 $\mathbf{E}^{(k+1)}$.[13]

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 $\mathrm{mat}\left(\mathbf{g}_i^{(k)}\right) = \mathbf{a}_i^{(k)}\mathbf{e}_i^{(k)T}$ is a rank one matrix, so that $\mathbf{g}_i^{(k)} \cdot \mathbf{u}^{(k)} = \mathbf{a}_i^{(k)T}\mathrm{mat}\left(\mathbf{u}^{(k)}\right)\mathbf{e}^{(k)}$. A sufficiently efficient way to vectorise these computations is the following:

$$\mathbf{v} = \mathbf{G}^T\mathbf{u} = \sum_{k=0}^{\ell-1} \mathrm{diag}\left(\mathbf{A}^{(k),T}\mathrm{mat}\left(\mathbf{u}^{(k)}\right)\mathbf{E}^{(k+1)}\right) \tag{21}$$

For (2), similar considerations give

$$\mathbf{G}^T\mathbf{G} = \sum_{k=0}^{\ell-1} (\mathbf{A}^{(k)T}\mathbf{A}^{(k)}) \odot (\mathbf{E}^{(k+1)T}\mathbf{E}^{(k+1)}) \tag{22}$$

Finally, for (3), note that $\mathbf{G}\mathbf{w}$ 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

$$\mathrm{mat}\left((\mathbf{G}\mathbf{w})^{(k)}\right) = \mathbf{A}^{(k)T}\left(\mathbf{E} \odot (\mathbf{w}\mathbf{1}^T)\right) \tag{23}$$

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.

[13] 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.

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 (21) takes time $O(B\sum_k n_k^2))$, (22) takes time $O(B\sum_k n_k^2)$, (23) 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 I.6 and I.7.

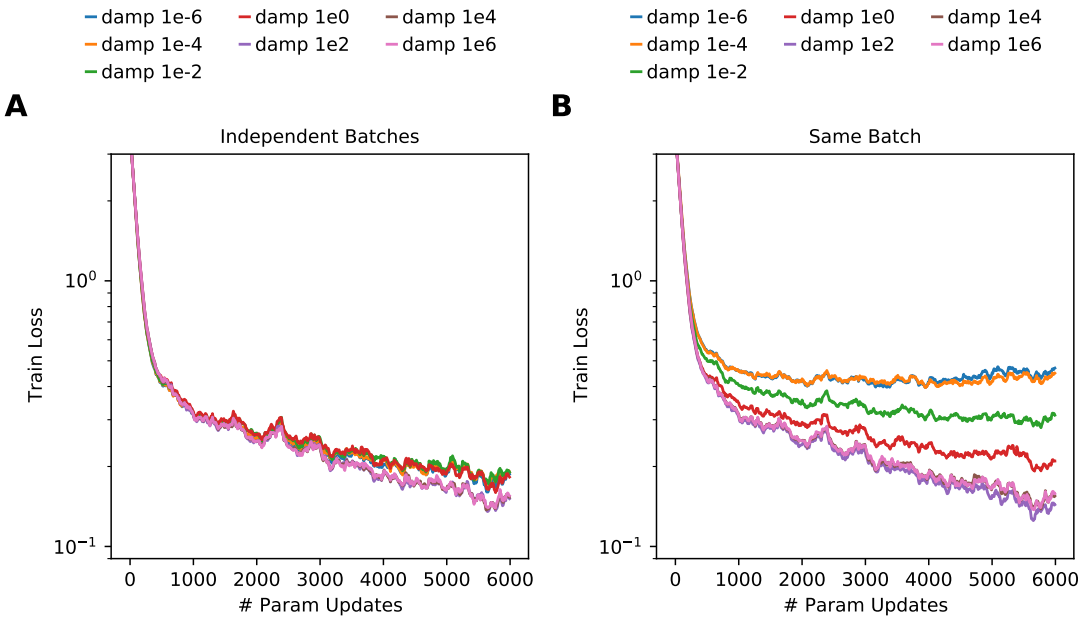

Figure I.6. Investigating the effect of evaluating Fisher and gradient on different respectively identical minibatches.

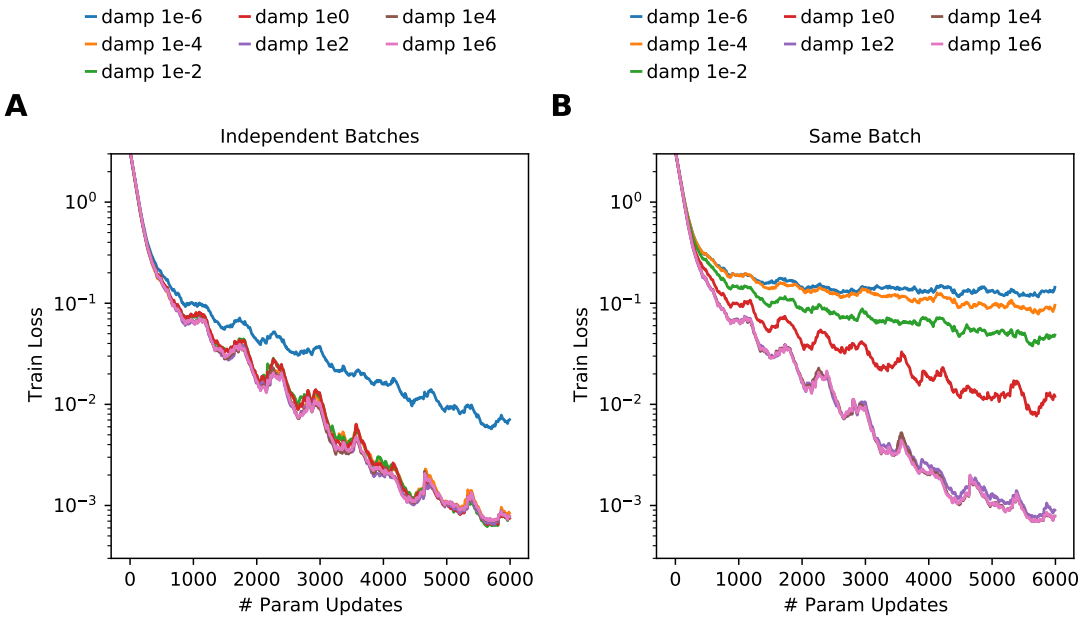

Figure I.7. Same as Figure I.6, but on MNIST.

J. Additional Experiments

In Section J.1, there are experiments analogous to Figures 1-4 from the main paper, but on MNIST rather than Fashion-MNIST. Results are in line with the ones on Fashion-MNIST, but effects are usually smaller, presumably due to MNIST-classification being a very simple task for MLPs.

In Section J.2, there are comparisons of subsampled natural gradients to subsampled KFAC for a ResNet on CIFAR10 and for autoencoder experiments. The fact that KFAC performs considerably better than natural gradients strongly suggests that also in these settings KFAC cannot be seen as a natural gradient method.

In Section J.3, we show that heuristic damping is crucial for performance of KFAC for a ResNet on CIFAR10 and for autoencoder experiments. This further supports the claim that KFAC should not be seen as a natural gradient method and suggests that similarity to FOOF is important for KFAC.

In Section J.4, we show performance comparisons between KFAC and FOOF for different benchmarks and architectures. Figure J.16 contains performance of a Wide ResNet18 on CIFAR 100 (rather than CIFAR 10 in the main paper). Figures J.17, J.18 contain training data for a VGG11 network on SVHN and additionally show how different algortihms are affected by different batch sizes. Figures J.19-J.21 contain autoencoder experiments on MNIST, Curves and Faces. Here, we make a somewhat puzzling observation: When we follow the KFAC training trajectory, FOOF makes more progress per parameter udpate than KFAC. Nevertheless, when we use FOOF for training, we obtain slightly worse results than for KFAC. This may suggest that in the autoencoder experiments KFAC chooses a different trajectory that is easier to optimize than FOOF. We emphasise that similarity to FOOF remains a significantly better explanation for KFAC's performance than similarity to natural gradients (recall Figure J.13). As an additional experiment we run KFAC with the empirical Fisher, rather than an MC approximation. Despite its name, the empirical Fisher is usually argued to be a poor approximation of the Fisher (Martens, 2014; Kunstner et al., 2019). Nevertheless we find that KFAC works equally well with the empirical Fisher, see Figure J.22, also supporting the view that KFAC's effectiveness is not directly linked to the Fisher.

J.1. Experiments anaologous to main paper but on MNIST rather than Fashion MNIST

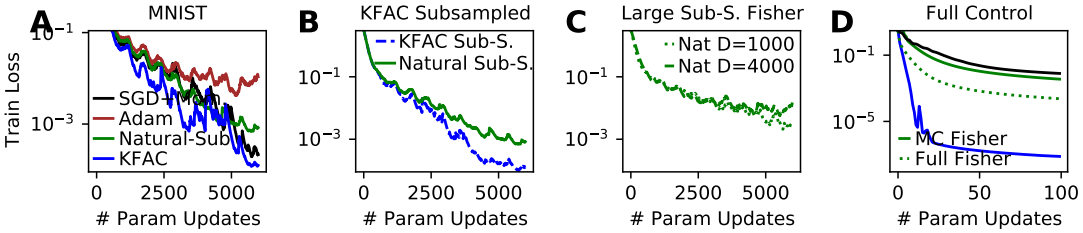

Figure J.8. **Paradoxically, KFAC – an approximate second-order method – outperforms exact second-order udpates in standard as well as important control settings.** Same as Figure 1 but on MNIST rather than Fashion MNIST.

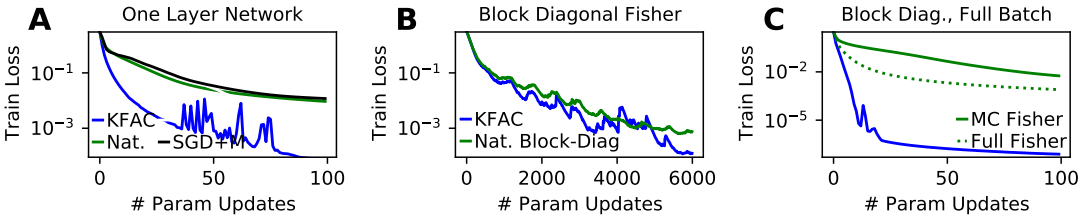

Figure J.9. **Advantage of KFAC over exact, subsampled Natural Gradients is not due to block-diagonal structure.** Same as Figure 2 but on MNIST rather than Fashion MNIST.

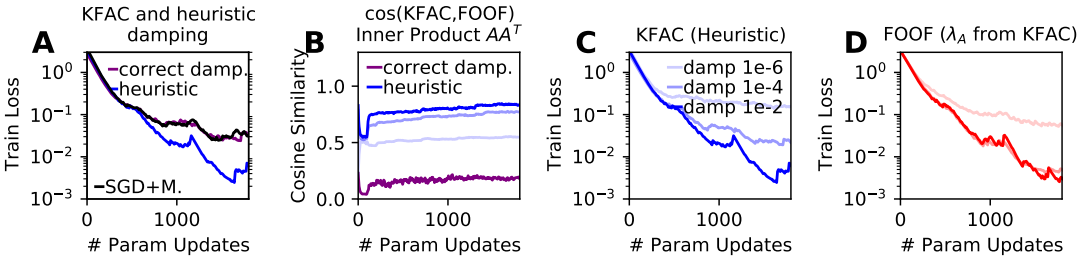

Figure J.10. **Heuristic Damping increases KFAC's performance as well as its similarity to first-order method FOOF.** Same as Figure 3 but on MNIST rather than Fashion MNIST.

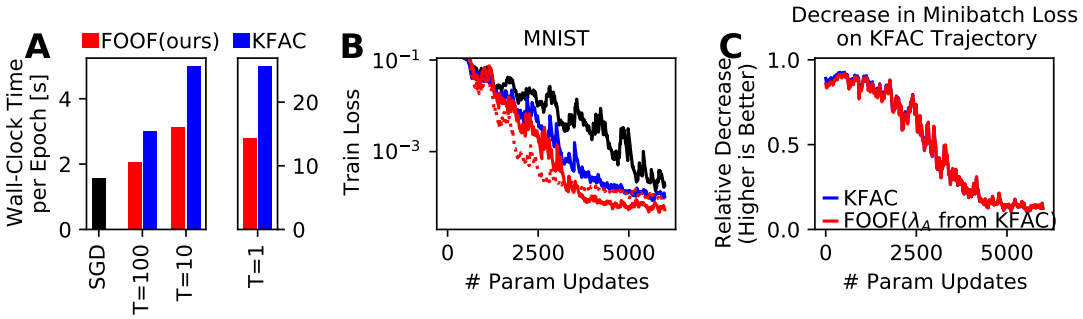

Figure J.11. **FOOF outperforms KFAC in terms of both per-update progress and computation cost.** Same as Figure 4 but on MNIST rather than Fashion MNIST.

J.2. Subsampled Natural Gradients vs Subsampled KFAC

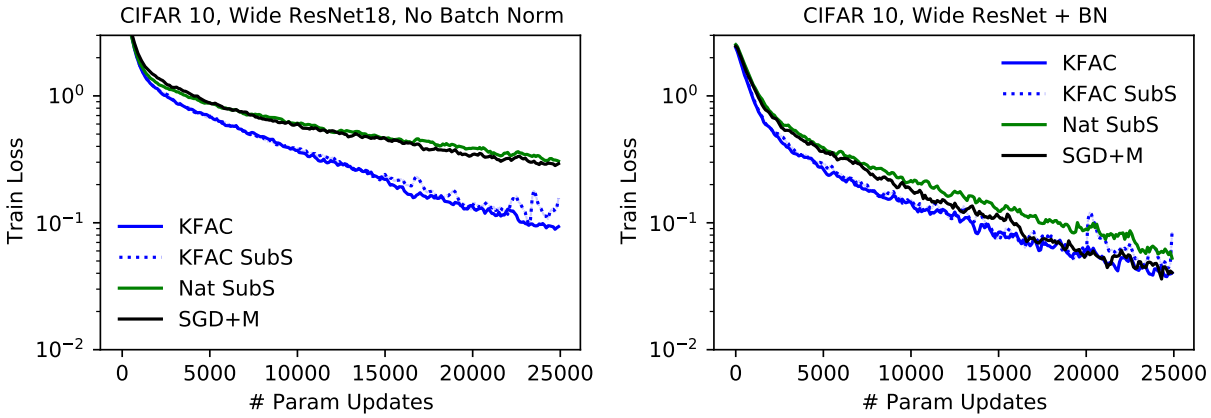

Figure J.12. KFAC outperforms Subsampled Natural Gradients, also when KFAC is subsampled and uses exactly the same amount of data as Natural gradients to estimate the curvature. This is analogous to Figure 1B, but on ConvNets and with a more complicated dataset. It confirms our claim that KFAC does not rely on second-order information. Note that with large damping, natural gradients becomes approximately equal to SGD – thus the difference seen between SGD+M and natural gradients is due to momentum.

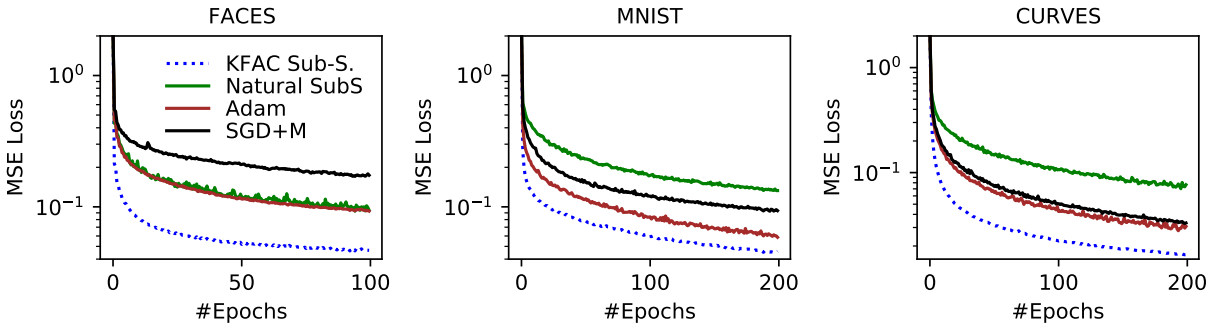

Figure J.13. KFAC outperforms Subsampled Natural Gradients, also when KFAC is subsampled and uses exactly the same amount of data as Natural gradients to estimate the curvature. Autoencoder Experiments. We confirmed that Natural Gradients do not perform worse than SGD without momentum. In other words the advantage of SGD+M vs Natural Gradients on MNIST and Curves is due to using momentum.

J.3. Effect of Heuristic Damping on KFAC

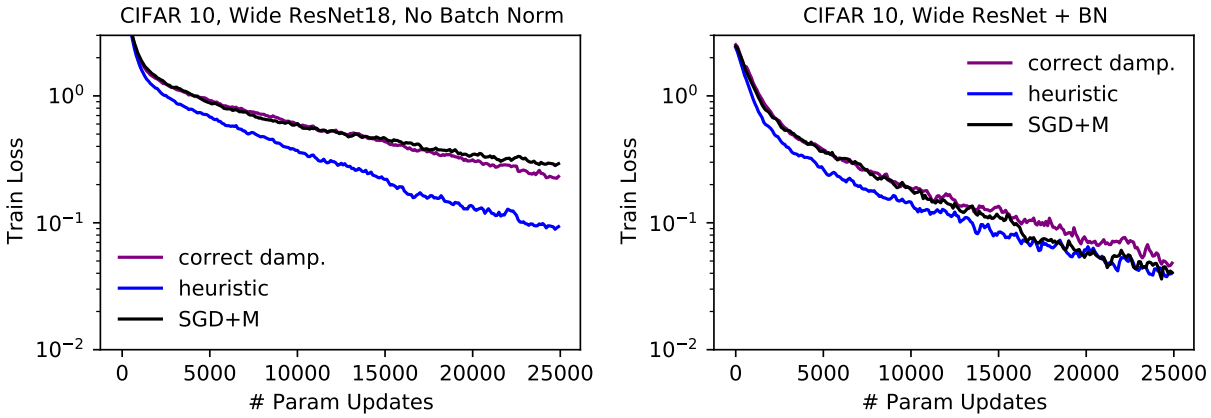

Figure J.14. Effect of Heuristic damping on KFAC on CIFAR10 with a ResNet. Analogously to Figure 3A, we find that heuristic damping is essential for KFAC's performance.

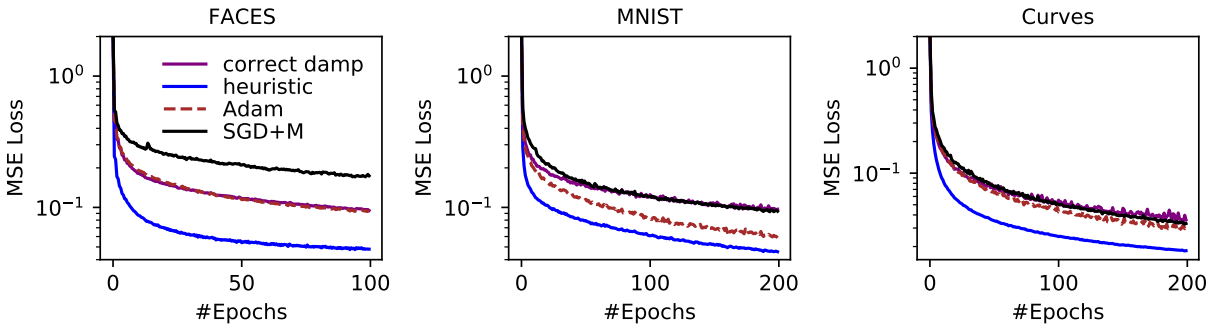

Figure J.15. Effect of Heuristic damping on KFAC in autoencoder experiments. Analogously to Figure 3A, we find that heuristic damping is essential for KFAC's performance.

J.4. Performance Comparisons

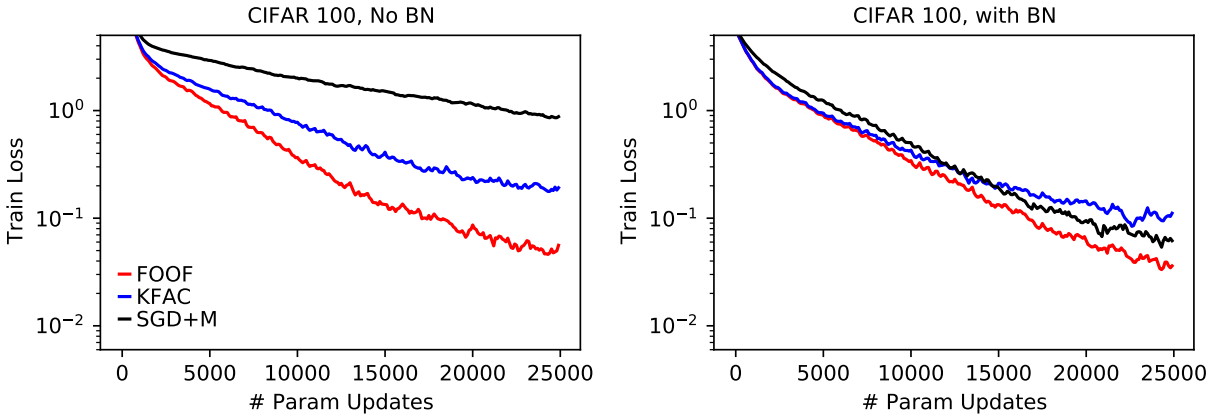

Figure J.16. Performance comparison on CIFAR 100 with a Wide ResNet18.

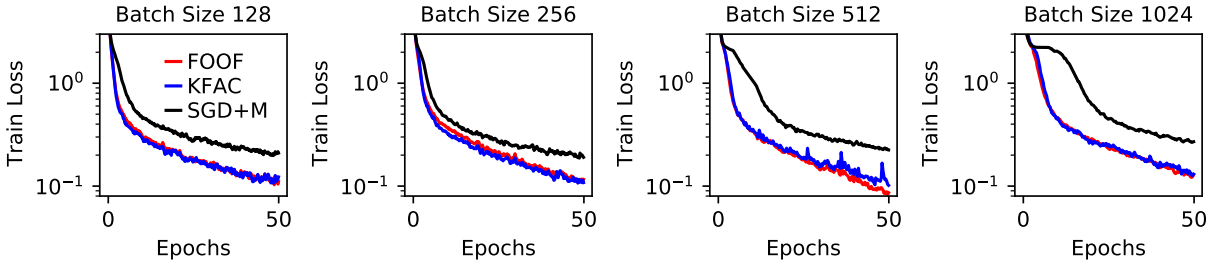

Figure J.17. Performance comparison on SVHN with a VGG11 network and different batch sizes. See also Figure J.18 for same data portrayed differently.

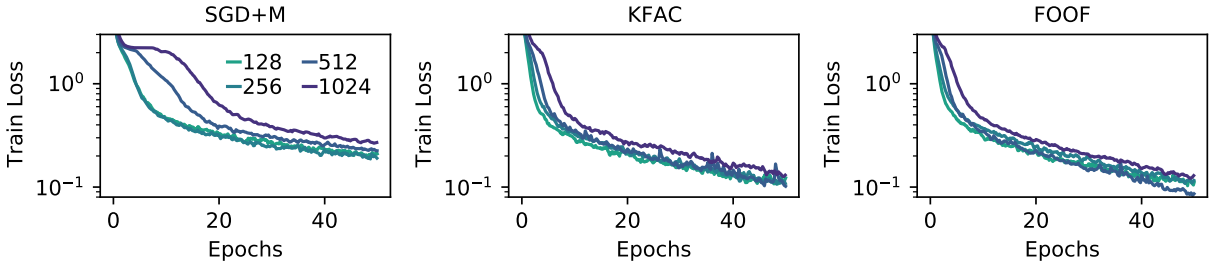

Figure J.18. Performance on SVHN with a VGG11 network across different batch sizes for different algorithms. See also Figure J.17 for same data portrayed differently. Note that color coding differs from remaining plots in the paper.

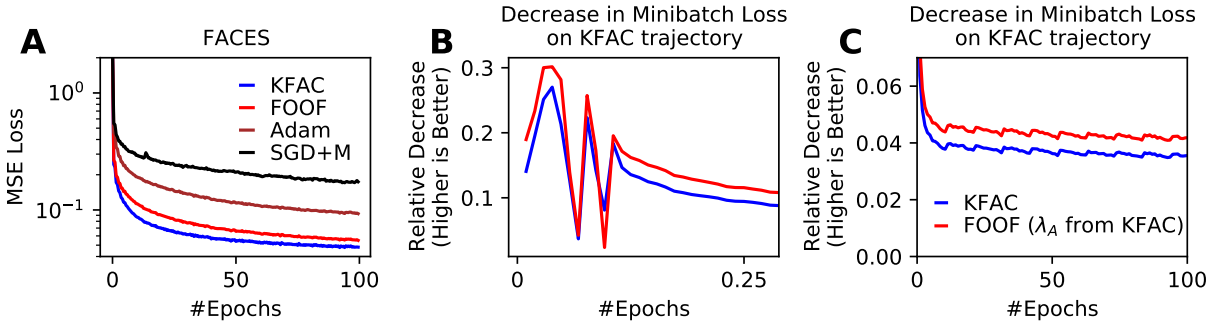

Figure J.19. Performance for FACES autoencoder experiment. KFAC slightly outperforms FOOF, but when FOOF is on KFAC trajectory it typically makes more progress per udpate. This may suggest that the advantage of KFAC is due to choosing a different optimization trajectory. (B) shows same data as (C) with a different axes zoom.

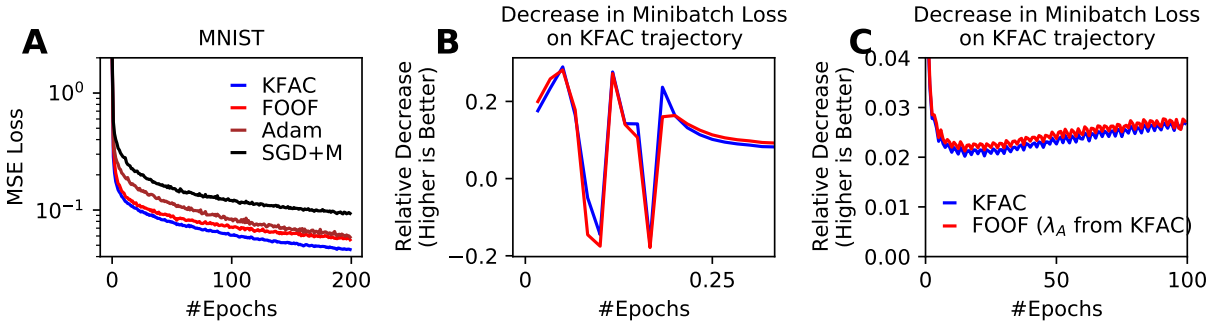

Figure J.20. Performance for MNIST autoencoder experiment. KFAC slightly outperforms FOOF, but when FOOF is on KFAC trajectory it typically makes more progress per udpate. This may suggest that the advantage of KFAC is due to choosing a different optimization trajectory. (B) shows same data as (C) with a different axes zoom.

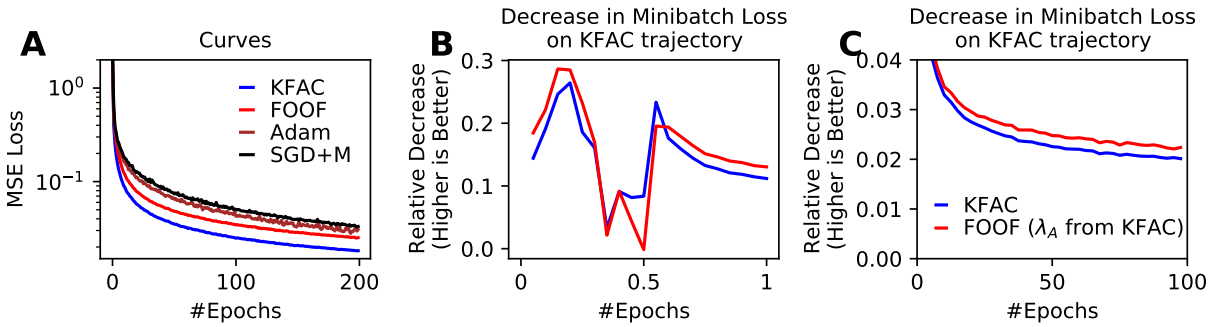

Figure J.21. Performance for Curves autoencoder experiment. KFAC slightly outperforms FOOF, but when FOOF is on KFAC trajectory it typically makes more progress per udpate. This may suggest that the advantage of KFAC is due to choosing a different optimization trajectory. (B) shows same data as (C) with a different axes zoom.

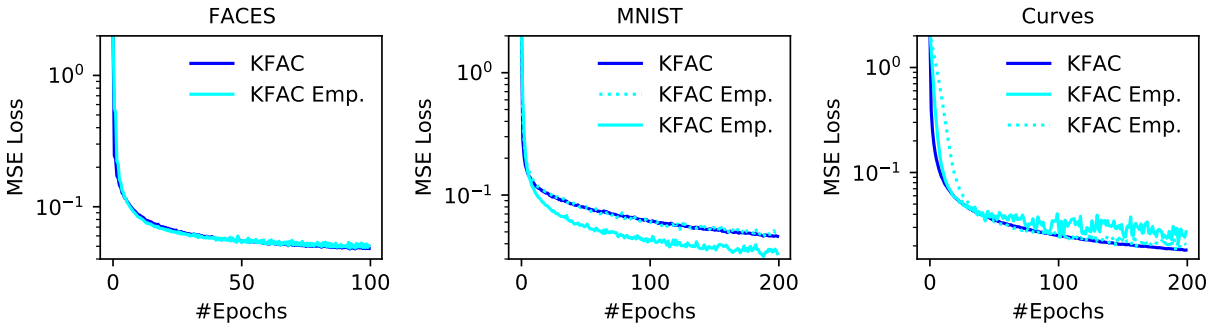

Figure J.22. Performance of standard KFAC (using an MC sample to estimate the Fisher) and a version of KFAC using the empirical Fisher. Solid an dashed cyan lines show different hyperparametrisations of the same algorithm. The advantage fo the empirical Fisher on MNIST seems to be due to allowing different hyperparametrisations to be stable.

Algorithm 1 Gradient Descent on Neurons (FOOF)

Hyperparameters: learning rate η, damping strength λ, exponential decay factor m, inversion period T, number of updates for input covariance $S \leq T$
Initialise: $t = 0$; For each layer ℓ: 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)

while train **do**
  Perform Standard Forward and Backward Pass For Current Mini-Batch With Loss L
  for each layer ℓ **do**
    $\mathbf{W}_\ell \leftarrow \mathbf{W}_\ell - \eta \mathbf{P}_\ell \nabla_{\mathbf{W}_\ell} L$ {Update Parameters as in Eq. 6}
    if $(t \bmod T) == 0$ **then**
      $\mathbf{P}_\ell \leftarrow (\mathbf{\Sigma}_\ell + \lambda \mathbf{I})^{-1}$ {Update Damped Inverse of Moving Average of $\mathbf{A}_\ell \mathbf{A}_\ell^T$ every T steps}
    end if
    if $((t + S) \bmod T) \in \{0, \dots, S-1\}$ **then**
      $\mathbf{\Sigma} \leftarrow m \cdot \mathbf{\Sigma}_\ell + (1 - m) \cdot \mathbf{A}_\ell \mathbf{A}_\ell^T$ {Update Moving Average of $\mathbf{A}_\ell \mathbf{A}_\ell^T$ beginning S steps before inversion in line 1. $\mathbf{A}_\ell$ is defined as in Section 2.2.}
    end if
  end for
  $t \leftarrow t + 1$
end while

K. Pseudocode, Implementation, Hyperparameters

Pseudocode for FOOF is given in Algorithm 1. Notation is anaologous to Section 2.2 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 1 of Algorithm 1 for a number of minibatches (50) before training and then executing line Line 1 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: Note that a discussion of hyperparameter robustness is also provided in Section D. 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. Alternatively, we typically fount $\lambda = 100$ to work well, but the exact magnitude may depend on implementation details (in particular, how factors are scaled and how they depend on the batch size).

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 (at the cost of making small approximations) is presented in (Ober and Aitchison, 2021).