OPT2021: 13th Annual Workshop on Optimization for Machine Learning

Kronecker-Factored Second-Order Optimizers Perform First-Order Descent on Neurons

Frederik Benzing BENZINGFR@GMAIL.COM
Department of Computer Science
ETH Zurich
8092 Zurich, Switzerland

Abstract

Second-order optimizers hold the potential to provide faster and more stable optimization than first-order methods. Due to the enormous size of the curvature matrix, it is widely believed that second-order methods are computationally intractable. Therefore, several approximate algorithms have been proposed. By far the most prominent and successful ones are Kronecker-Factored, block-diagonal approximations of the curvature matrix and they are considerably more data-efficient than standard first-order methods. However, so far it had not been investigated how accurate these approximations of the curvature matrix are. Our first contribution is to develop a tool to close this gap, namely a method to efficiently evaluate exact second-order updates based on a (potentially large) mini-batch estimate of the curvature matrix. We then combine this tool with a careful set of experiments to demonstrate that Kronecker-factored approximations of the curvature are rather imprecise, but nevertheless and very surprisingly offer faster optimization than exact second-order updates. Finally, to resolve this apparent contradiction, we explain how the Kronecker-factorisation induces similarity to a principled first-order method that performs gradient descent on neuron outputs rather than individual weights. We also show that this similarity is responsible for the success of Kronecker-factored methods. All in all, our contribution offers a valuable tool to analyse second-order methods, it provides a fundamentally different and improved understanding of Kronecker-factored methods and leads to more efficient optimization algorithms, both in terms of computational cost and progress per parameter update.

We refer the reader to an updated version of this paper "Gradient Descent on Neurons and its Link to Approximate Second-Order Optimisation" on arxiv. Code and link to the paper will be availabe on github https://github.com/freedbee/Neuron_Descent_and_KFAC

1. Introduction

While first-order optimization algorithms are the de facto standard for training neural networks, second-order methods hold the promise of allowing significant optimization speed-ups. However, they require estimating and inverting the curvature matrix, whose size scales as the square of the number of parameters. The well known natural gradient method [2] is often seen as a second-order optimization method, whose curvature matrix is given by the Fisher Information. To make its updates computable efficiently, [21] and [14] have proposed a Kronecker-factored, block-diagonal approximation of the Fisher Information and demonstrated strong empirical performance of the resulting algorithm [4, 14, 21].

Here, we first show that, when using a subsampled (or, in other words, mini-batch) estimate of the Fisher Information, it is not necessary to make approximations to compute natural gradients

exactly. When testing the resulting natural gradient method, we find that it performs significantly worse than KFAC. Very surprisingly, this observation also persists after controlling for the fact that KFAC uses an approximation of the full, rather than subsampled Fisher and after controlling for the block-diagonal structure of KFAC. This suggests that KFAC does not owe its optimization performance to approximating the curvature. Inspired by these insights and in order to understand KFAC's surprising effectiveness, we introduce a new first order optimizer. On top of its explantory power, this optimizer also has the advantage of outperforming KFAC in terms of computational cost and progress per parameter update date.

The paper is structured as follows: In a theoretical part we introduce our new-first order optimizer and describe how KFAC relates to it. In the experimental part, we first analyse why KFAC performs so well and then test the performance of our new optimizer. For related work and additional experiments we refer the reader to the appendix. For computing natural gradients efficiently, our insights compared to prior work [28] are mostly technical and we therefore postpone details to the appendix.

2. A New First Order Method (FOOF)

We introduce a new, simple first-order optimizer, called "Fast First-Order Optimizer" or "FOOF"[1]. Despite its simplicity and effectiveness, we are not aware of a description of it in prior literature.

Let us consider one layer of a neural network with weight matrix $\mathbf{W} \in \mathbb{R}^{n \times m}$. Denote the layer's input-activations (after the previous' layer non-linearity) by $\mathbf{A}$, which is a $m \times D$ matrix, where D is the number of datapoints. The network's output activations (before the non-linearity) are equal to $\mathbf{WA}$ and we denote backpropagated gradients (or errors) of the output units by $\mathbf{E} = \frac{\partial L}{\partial (\mathbf{WA})}$.

Typically, we compute the weights' gradients for each datapoint and average the results. Changing perspective, we can see the i-th datapoint as a linear constraint, that prescribes to change the weight so that the output of the layer changes in the direction of $\mathbf{E}_i$. We can then jointly optimize these constraints (rather than averaging gradients) to obtain a weight update. In other words, we want to find an update $\Delta\mathbf{W}$ to the parameters $\mathbf{W}$, so that the layer's output changes in the gradient direction, i.e. $(\Delta\mathbf{W})\mathbf{A} = \mathbf{E}$. More formally, we want to optimize

$$\min_{\Delta\mathbf{W} \in \mathbb{R}^{n \times m}} \|(\Delta\mathbf{W})\mathbf{A} - \mathbf{E}\|^2 \tag{1}$$

This is simply a linear regression problem (for each row of $\mathbf{W}$), whose explicit solution is

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

As a regulariser, one can use ridge regression, yielding updates of the form $(\mathbf{A}\mathbf{A}^T + \lambda\mathbf{I})^{-1}\mathbf{A}\mathbf{E}^T$. See Figure 5 for empirical results of this optimizer.

This form of udpate can also be seen as preconditioning by $\left(AA^T \otimes \mathbf{I}\right)^{-1}$ and we emphasise that this preconditioning matrix contains no first-order, let along second-order, information.

2.1. Unbiased, Stochastic Version of FOOF and Amortisation of Matrix Inversion

The above algorithm implicitly uses the entire dataset to compute updates. If we want to apply it in a stochastic setting, we need to take some care to limit the bias of our updates. In particular, using

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

one mini-batch to estimate all quantities in equation (2) gives a biased approximation of the weight update. This is because we would approximate the product of the expectations of (non-independent) random variables as the expectation of the product; see also discussion in Appendix E.5.

For computational efficiency, rather than computing $\mathbf{A}\mathbf{A}^T$ on the full batch at each iteration, we keep an exponentially moving average of this quantity based on mini-batches and invert the resulting approximation. For the quantity $\mathbf{A}\mathbf{E}^T$ we used standard, unbiased mini-batch estimates.

The cost of inverting $\mathbf{A}\mathbf{A}^T$ can be amortised by only performing the inversion every T time steps for some suitable T. This could lead to a stale estimate of $\mathbf{A}\mathbf{A}^T$, but empirical results suggest that this is not a problem.

2.2. Relation of FOOF to (damped) KFAC

KFAC [14, 21] is a natural gradient method, which makes two approximations to the Fisher. First, only diagonal blocks of the Fisher, corresponding to individual layers of the network, are considered. Second, each block is approximated as a kronecker-product of two matrices. To describe KFAC's update rule, we use the same notation as above. In addition, we use $\mathbf{E}_F$ analogously to $\mathbf{E}$, but gradients are computed with respect to labels sampled from the model distribution. KFAC approximates each diagonal block of the Fisher as $(\mathbf{A}\mathbf{A}^T) \otimes (\mathbf{E}_F\mathbf{E}_F^T)$; while this approximation is exact for a single datapoint, it requires additional assumptions for more datapoints. For computational efficiency $\mathbf{A}\mathbf{A}^T$ as well as $\mathbf{E}_F\mathbf{E}_F^T$ are estimated as an exponentially moving average of mini-batch estimates. Using standard kronecker-product identities and noting that the gradient with respect to the weights is given by $\mathbf{A}\mathbf{E}^T$, this leads for the following update rule of KFAC:

$$(\Delta\mathbf{W})^T = \left(\mathbf{A}\mathbf{A}^T + \lambda_A\mathbf{I}\right)^{-1}\left(\mathbf{A}\mathbf{E}^T\right)\left(\mathbf{E}_F\mathbf{E}_F^T + \lambda_E\mathbf{I}\right)^{-1}, \tag{3}$$

where λ_A, λ_E are damping terms satisfying $\lambda_A \cdot \lambda_E = \lambda$ for a hyperparmater λ and $\frac{\lambda_A}{\lambda_E} = \frac{\mathrm{Tr}(\mathbf{A}\mathbf{A}^T)}{\mathrm{Tr}(\mathbf{E}\mathbf{E}^T}$ Recall that the update of our first order method FOOF is given by

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

where full batch $\mathbf{A}\mathbf{A}^T$ is also estimated as an exponentially moving average of mini-batch estimates. All that separates KFAC from our first-order algorithm is post-multiplication by $(\mathbf{E}_F\mathbf{E}_F^T + \lambda_E\mathbf{I})^{-1}$.

If the errors of different output neurons are uncorrelated, then $\mathbf{E}_F\mathbf{E}_F^T$ will be dominated by its diagonal. If further, the damping term is sufficiently large, $(\mathbf{E}_F\mathbf{E}_F^T + \lambda_E\mathbf{I})^{-1}$ will be close to the identity and the KFAC update will be close to FOOF. We will see below that this proximity, which is induced by damping, is linked to KFAC's performance.

3. Experiments

We carry out experiments on fully connected networks on MNIST [18] and Fashion MNIST [32] and average all results across three random seeds. A preliminary discussion of convolutional nets will be given below. We will investigate our subsampled natural gradient method, our first order method FOOF as well as KFAC [14, 21], SGD and Adam [17]. All methods use a constant learning rate, and, if applicable, a constant damping term λ.[2] Hyperparameters are optimized independently

2. For the natural gradient method we also experimented with automatic step size selection, adaptive damping and a form of momentum, all as described by [21], but found no notable performance gains.

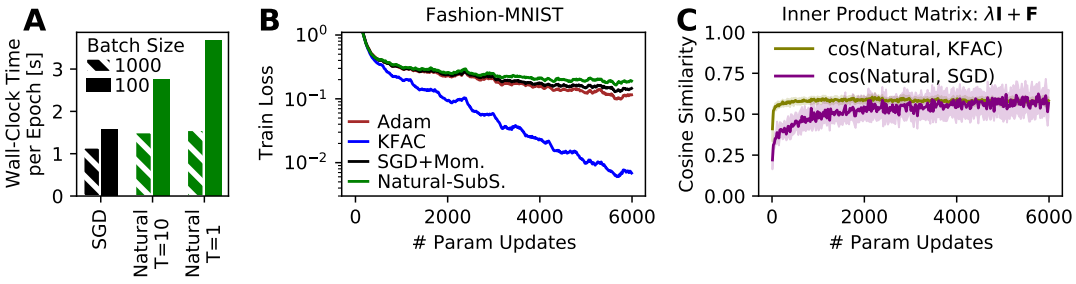

Figure 1: **KFAC outperforms exact, subsampled natural gradients.** **(A)** Wall-clock time comparison between SGD and Natural Gradients. T denotes how frequently the inverse Fisher is computed (implicitly). Implemented with PyTorch and run on a GPU (with optimized DataLoader). **(B, C)** Training loss of different algorithms trained for 10 epochs with batch size 100 on Fashion MNIST and MNIST. Perhaps surprisingly, KFAC, an approximate natural gradient method, reaches a loss that is approximately 10 times lower than the exact, subsampled natural gradients on both benchmarks.

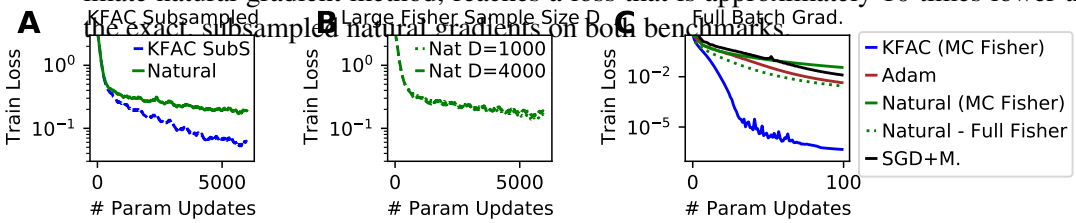

Figure 2: **Advantage of KFAC is not due to using more data to estimate the Fisher.** **(A)** Comparison between version of KFAC and Subsampled Natural Gradients, which use the same amount of data to estimate the Fisher. Very surprisingly, KFAC's advantage persists. **(B)** Versions of subsampled natural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

for each method and each experiment/ablation. As in KFAC, our subsampled Fisher is computed by drawing one label from the model distribution for each input image, also when full-batch gradients are computed.

Our first experiment measures the wall-clock time of our subsampled natural gradient algorithm, see Figure 1A. The amortised version, while performing equally to the unamortised version, requires less than double the time of vanilla SGD, showing that natural gradients can be computed efficiently in practice. The algorithm requires inverting a matrix of dimension $D \times D$, where D is the number of datapoints used to estimate the Fisher.[3] In practice, th cost of this inversion is dominated by other factors even for comparatively large batch sizes of 1000.

Next, we measure performance of subsampled natural gradients, see Figure 1B,C. Surprisingly, natural gradients are significantly outperformed by KFAC, which reaches an approximately 10 times lower loss on both benchmarks. This is despite KFAC being an approximation of natural gradients and our method computing exact natural gradients, except for subsampling the Fisher.

This directly leads to the hypothesis that KFAC's advantage over subsampled natural gradients is due to using more data for its approximation of the Fisher. To test this, we perform three experi-

3. By 'datapoint' we refer to a pair of input and label, where the latter is sampled from the model distribution.

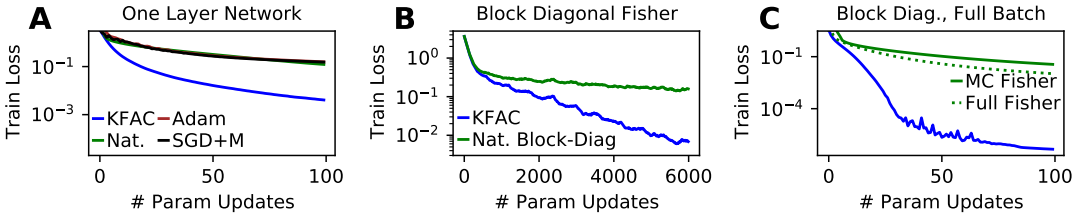

Figure 3: **Advantage of KFAC is not due to block-diagonal structure.** **(A)** A one layer network is trained on 1000 images and full batch gradients are used. In a one layer network the block-diagonal Fisher coincides with the full Fisher. **(B)** Comparison between KFAC and layerwise (i.e. block-diagonal) subsampled Natural Gradients. **(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.

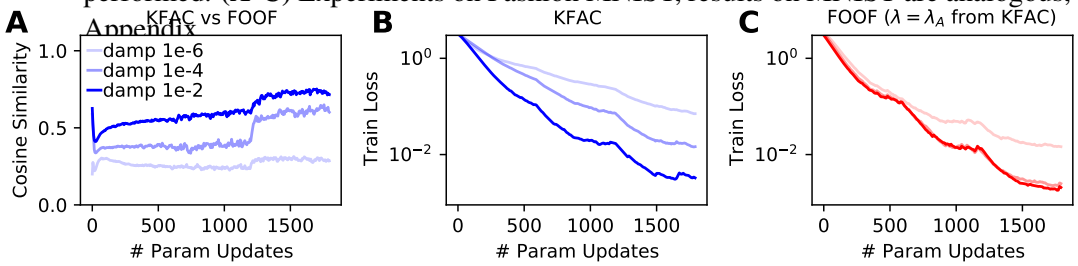

Figure 4: **Damping increases KFAC's performance as well as its similarity to our first order method FOOF.** **(A)** Damping increases similarity of KFAC to FOOF. **(A)** Performance of KFAC for different damping strengths. Performance increases with damping. (The learning rate was figures/ed for each damping strength individually.) **(C)** Performance of FOOF for different damping strengths. (A-C) and our theoretical analysis, this suggest that KFAC owes its performance to similarity to FOOF. **(A-C)** Experiments on MNIST, results on Fashion MNIST are analogous, see Appendix.

ments. (1) We explicitly restrict KFAC to use the same amount of data 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. The results are shown in Figure 2 and all lead to the same conclusion: The fact that KFAC uses more data than subsampled natural gradients does not explain its better performance.

This begs further investigation into why KFAC outperforms natural gradients. There are two approximations that KFAC makes to the Fisher. (1) It approximates the Fisher as block-diagonal. (2) It approximates each diagonal block as a kronecker product. To test whether (1) explains KFAC's performance, we perform two experiments. First, we train a one layer network on a subset of 1000 images with full-batch gradient descent. For a one layer network, the block-diagonal Fisher coincides with the Fisher. So, if the block-diagonal approximation were responsible for KFAC's performance, then in a one layer network, natural gradients should perform as well as KFAC. However, this is not the case as shown in Figure 3A. As an additional experiment, we consider a three layer network and calculate layer-wise natural gradients, ignoring inter-layer interactions, or equivalently, approximating the Fisher by its block-diagonal (but without approximating blocks as kronecker-products). Layer-wise natural gradients are computable efficiently exactly like full natural gradients

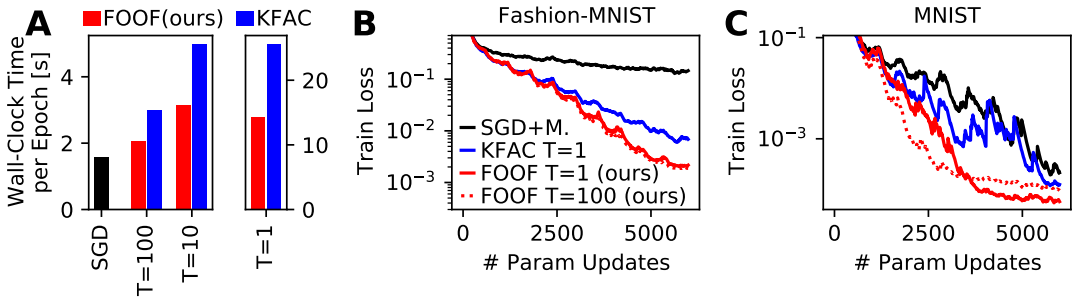

Figure 5: **FOOF outperforms KFAC in terms of both per-update progress and computation cost. (A)** Wall-clock time comparison between FOOF, KFAC and SGD. T denotes how frequently matrix inversions (see eq (2)) are performed. Implemented with PyTorch and run on a GPU (with optimized DataLoader). Increasing T above 100 does not notably improve runtime. **(B, C)** Training loss of different algorithms trained for 10 epochs with batch size 100 on Fashion MNIST and MNIST.

with our method. We run the layer-wise natural gradient algorithm in two settings: In a mini-batch setting, identical to the one shown in Figure 1 and in a full-batch setting, by restricting to a subset of 1000 training images. The results are shown in Figure 3B,C and confirm the finding from our first experiment: It is not the block-diagonal approximation that explains KFAC's performance.

All evidence points to one unexpected conclusion: KFAC owes its performance not to approximating natural gradients or layer-wise natural gradients, but – somehow – to using a kronecker-factored approximation of the diagonal blocks. We next turn to why this is the case.

We have already seen theoretically that KFAC is very similar to our first-order algorithm FOOF, raising the possibility that similarity to FOOF explains KFAC's performance. As discussed theoretically, similarity to FOOF likely requires the damping term to be sufficiently large. Indeed, Figure 4A confirms that increasing the damping strength increases similarity between KFAC and FOOF. For this experiment, to obtain a meaningful comparison, we slightly modify FOOF to always have the same damping strength λ_A as KFAC. Moreover, Figure 4B shows that the performance of KFAC increases with larger damping and becomes more similar to the performance of FOOF shown in Figure 4C. This supports our hypothesis that KFAC's performance is due to similarity to FOOF.

For an additional piece of evidence, note that KFAC damps each individual Kronecker-factor, rather than the entire product. This difference is pointed out in the original KFAC paper [21] along with the observation that damping factors leads to better performance, despite being theoretically less justified. Our view of KFAC explains this otherwise mysterious behaviour: Damping factors increases similarity to FOOF more than damping the product.

Finally, if our hypothesis holds, then we would expect FOOF to perform similarly to, if not better than, KFAC. This is indeed the case as shown in Figure 5B,C and further supports our hypothesis.

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 $\mathbf{E}\mathbf{E}^T$). These savings lead to a 1.5x speed-up in wall-clock time per-update for the amortised versions of KFAC and FOOF as shown in Figure 5A.

4. Limitations

Here, we measured each algorithms performance by its training loss and this was also the criterion by which hyperparameters were selected. It will also be interesting to investigate the generalisation properties of respective algorithms and to see how they interact with regularisation techniques. This will also require more complex benchmarks.

While we the above experiments are restricted to fully connected networks, preliminary results on convolutional nets are in line with all our main findings: (1) Natural Gradients are efficiently computable. (2) They are significantly outperformed by KFAC. (3) KFAC's Kronecker Factor $\mathbf{A}\mathbf{A}^T$ is identical to the premultiplier of FOOF in architectures with parameter sharing. (4) FOOF seems to perform as well as or better than KFAC.

5. Conclusion

We have shown that subsampled natural gradients are efficiently computable in large neural networks. The underlying techniques can also be used to compute subsampled Generalised Gauss-Newton updates efficiently. Moreover, they have potnetial applications in continual learning, meta learning and for bayesian laplace posterior approximations, as discussed in Appendix A.

Through a series of careful experiments, we used our subsampled gradient method to give a new, fundamentally different explanation of why KFAC is as effective as it is.

Finally, our analysis of KFAC lead us to develop a simple and effective first order optimizer, which is more efficient both in terms of computational cost and progress per parameter update.

References

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

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

[3] Shun-ichi Amari and Hiroshi Nagaoka. *Methods of information geometry*, volume 191. American Mathematical Soc., 2000.

[4] Jimmy Ba, Roger Grosse, and James Martens. Distributed second-order optimization using kronecker-factored approximations. 2016.

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

[6] Frederik Benzing. Unifying regularisation methods for continual learning. *arXiv preprint arXiv:2006.06357*, 2020.

[7] Alberto Bernacchia, Máté Lengyel, and Guillaume Hennequin. Exact natural gradient in deep linear networks and application to the nonlinear case. NIPS, 2019.

[8] Aleksandar Botev, Hippolyt Ritter, and David Barber. Practical gauss-newton optimisation for deep learning. In *International Conference on Machine Learning*, pages 557–565. PMLR, 2017.

[9] Erik Daxberger, Agustinus Kristiadi, Alexander Immer, Runa Eschenhagen, Matthias Bauer, and Philipp Hennig. Laplace redux–effortless bayesian deep learning. *arXiv preprint arXiv:2106.14806*, 2021.

[10] Guillaume Desjardins, Karen Simonyan, Razvan Pascanu, and Koray Kavukcuoglu. Natural neural networks. *arXiv preprint arXiv:1507.00210*, 2015.

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

[12] Thomas George, César Laurent, Xavier Bouthillier, Nicolas Ballas, and Pascal Vincent. Fast approximate natural gradient descent in a kronecker-factored eigenbasis. *arXiv preprint arXiv:1806.03884*, 2018.

[13] Donald Goldfarb, Yi Ren, and Achraf Bahamou. Practical quasi-newton methods for training deep neural networks. *arXiv preprint arXiv:2006.08877*, 2020.

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

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

[16] Yehuda Hoffman and Erez Ribak. Constrained realizations of gaussian fields-a simple algorithm. *The Astrophysical Journal*, 380:L5–L8, 1991.

[17] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

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

[19] Gaétan Marceau-Caron and Yann Ollivier. Practical riemannian neural networks. *arXiv preprint arXiv:1602.08007*, 2016.

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

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

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

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

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

[25] Yann Ollivier. True asymptotic natural gradient optimization. *arXiv preprint arXiv:1712.08449*, 2017.

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

[27] Razvan Pascanu and Yoshua Bengio. Revisiting natural gradient for deep networks. *arXiv preprint arXiv:1301.3584*, 2013.

[28] Yi Ren and Donald Goldfarb. Efficient subsampled gauss-newton and natural gradient methods for training neural networks. *arXiv preprint arXiv:1906.02353*, 2019.

[29] Hippolyt Ritter, Aleksandar Botev, and David Barber. Online structured laplace approximations for overcoming catastrophic forgetting. *arXiv preprint arXiv:1805.07810*, 2018.

[30] Hippolyt Ritter, Aleksandar Botev, and David Barber. 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, 2018.

[31] Nicolas Roux, Pierre-antoine Manzagol, and Yoshua Bengio. Topmoumoute online natural gradient algorithm. *Advances in Neural Information Processing Systems*, 20:849–856, 2007.

[32] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms, 2017.

[33] Guodong Zhang, James Martens, and Roger Grosse. Fast convergence of natural gradient descent for overparameterized neural networks. *arXiv preprint arXiv:1905.10961*, 2019.

Contents

A Further Applications of Implicit Fisher Computations 10

B Experimental Details 11

C Kronecker-Factored Curvature Approximations for Laplace Posteriors 12

D Related Work 13

E Details for Efficiently Computing $\mathbf{F}^{-1}$-vector products for a Subsampled Fisher 14

F Software Validation 16

G Additional Experiments 16

Appendix A. Further Applications of Implicit Fisher Computations

A.1. Bayesian Laplace Posterior Approximation

Laplace Approximations are a common approximation to posterior weight distributions and various techniques have been proposed to approximate them, see e.g. [9] for a recent overview and evaluation. 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.

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 [11], who credits [16].

A.1.1. SAMPLING FROM THE FULL COVARIANCE LAPLACE POSTERIOR FOR SUBSAMPLED FISHER INFORMATION

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. As for the natural gradients, we factorise $\mathbf{F} = \mathbf{G}\mathbf{G}^T$, where $\mathbf{G}$ is a $N \times D$ matrix (N is the number of parameters, D the number of datapoints). Using Woodbury's identity, we can write the posterior variance as

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

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

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

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

$$\tag{8}$$

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

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

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

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

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

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

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

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

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

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

A.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 E. The only modification is multiplication by $\mathbf{\Lambda}_{\text{prior}}^{-1}$ and for any diagonal prior, this can be solved easily.

A.2. Continual Learning

Closely related to bayesian posteriors is a number of continual learning algorithms []. For example, EWC 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 above, it is easy 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, scaling as the product of the number of datapoints and the number of neurons (rather than weights).

A.3. Meta Learning / Bilevel optimization

Some bilevel optimization algorithms rely on the Inverse Function Theorem to estimate outer-loop gradients after the inner loop has converged. They require evaluating a product of the form $(\lambda\mathbf{I} + \mathbf{H})^{-1}\mathbf{v}$, where $\mathbf{H}$ is the Hessian and $\mathbf{v}$ a vector, see e.g. [5]. If we approximate the Hessian by a subsampled Fisher, our methods are directly applicable to compute this product.

Appendix B. Experimental Details

Add details on KFAC momentum for AA^T and so on, including "adam" style normalisation
Add amortisation details for conv
Add initialisation details
cite PyTorch
add bias details
mention CIFAR data augmentation

B.1. Hyperparameter Tuning

Learning rates for all methods were tuned by a grid search, considering values of the form $1 \cdot 10^i, 3 \cdot 10^i$ for suitable (usually negative) integers i. The damping terms for KFAC and FOOF were determined by a grid searcher over $10^{-6}, 10^{-4}, 10^{-2}, 10^0, 10^2, 10^4, 10^6$. For SGD, momentum was grid-searched from $0.0, 0.9$. For Adam, we kept all hyperparameters except the learning rate fixed.

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

We always chose the hyperparameters which gives best training loss at the end of training. Usually, these hyperparameters also outperform others in the early training stage.

B.2. Remaining details

Experiments were carried out on fully connected networks, with 3 hidden layers of size 1000. The only exception is the network in Figure 3A, which has no hidden layer. Unless noted otherwise, we trained networks for 10 epochs which batch size 100 on MNIST or Fashion MNIST, which were preprocessed to have zero mean and unit variance. 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.

Appendix C. Kronecker-Factored Curvature Approximations for Laplace Posteriors

A Kronecker-factored approximation of the curvature has also been used in the context of laplace posteriors [30] and this has been applied to continual learning [29]. 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 [29, 30] of Kronecker-factorisations of the curvature rely on proximity to the curvature matrix.

Here, we provide an alternative hypothesis. For simplicity of notation, we assume that the network has only one layer. Suppose $\mathbf{W}_0$ is a local optimum of the log-likelihood of the parameters. Further denote the approximation to the posterior covariance by $\mathbf{\Sigma}$. For an approximate Laplace posterior to be effective, we require that parameters which are assigned high liklihood by the laplace approximation actually do have high liklihood. In other words, when a weight pertubation $\mathbf{V}$ satisfies that $\mathbf{v}^T\mathbf{\Sigma}\mathbf{V}$ is small, then the parameter $\mathbf{W} + \mathbf{V}$ should have high likelihood.

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 curvature is approximately $\mathbf{A}\mathbf{A}^T \otimes \mathbf{I}$. Then, some calculations give

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

where $\mathbf{v}_i$ is the i-th row of $\mathbf{V}$, i.e. the set of weights connected to the i-th output neuron.[4] 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 $\mathbf{A}$ with small singular values). This means that the layer's output, and consequently the networks output, get perturbed very little. This in turn means, that $\mathbf{W} + \mathbf{V}$ 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.

In fact, this algorithm has already been developed and tested independently of and prior to our results [23]. It shows very strong performance, consistent with our prediction.

In addition, We note that a major drawback of the continual learning method [29] based on Kronecker-factors is that it needs to store kronecker-factors for each task, resulting in very large memory requirements as well as increased computational cost, especially as the number of tasks grows. Formally, individual Kronecker-factors from each task need to be stored, since the sum of two Kronecker-products (from different tasks) can not generally be written as a single product. If our hypothesis is true, this would theoretically prescribe ignoring the second factor of each product and moreover adding the first factors (from different tasks, for a fixed layer) directly. This would significantly reduce the memory footprint of the resulting method. In fact, this algorithm would simply be an application of [23] to continual learning.

Appendix D. Related Work

Perhaps closest to our newly developed first-order optimizer is [23], which is a Bayesian Posterior approximation and can be viewed as considering distributions over neuron activations rather than in weight space directly, similarly to how FOOF performs optimization steps on neuron activations rather than on weights directly. The findings of [23] also support our interpretation of [30] (see Appendix C).

Natural gradients were proposed by Amari and colleagues, see e.g. [2] and its original motivation stems from information geometry [3]. It is closely linked to classical second-order optimization through the link of the Fisher to the Hessian and the Generalised Gauss Newton matrix [20, 27]. Moreover, natural gradients can be seen as a special case of Kalman filtering [26]. Interestingly, different filtering equations can be used to justify Adam's [17] update rule [1].

There is a long history of approximating natural gradients and second order methods. For example, HF [22] 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 [10, 15, 19, 22, 24, 25, 31].

Kronecker-factored approximations [15, 21] have become the basis of several optimization algorithms [7, 8, 12, 13]. 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 [30], which can also be applied to continual learning [29]. A more detailed discussion of this may relate to our findngs can be found in Section C.

There also is a large body of work on theoretical convergence properties of Natural Gradients. [7] analyse the convergence of natural gradients in linear networks. Interestingly, they show that for

4. 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.

linear networks applied to regression problems (with homoscedastic noise), inverting a Kronecker-Factored approximation of the curvature results in exact natural gradients. Our results show that this link breaks down in the non-linear case. For non-linear, overparametrised networks, [33] recently gave a convergence analysis of both natural gradients and KFAC. We also refer to [33] for a more thorough review of theoretical results.

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

E.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

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

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

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

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

If we process a batch of data, we will use upper case letters for (pre-)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 $\mathbf{W}^{(k)}$.

E.2. Overview

The technique described here is similar to [28]. However, the implementation of [28] requires several for- and backward passes for each mini-batch, which is used to compute the Fisher. Another key difference, both in terms of computation time and update-direction quality is discussed in Section E.5.

We now outline how to efficiently compute exact natural gradients under the assumption that the Fisher is estimated from a mini-batch of moderate size (where 'moderate' can be on the order of

thousands without large difficulties). Let's assume we have B samples $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_B, y_B)$ from the model distribution and use this for a MC estimate of the Fisher, i.e.

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

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

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

We will not compute $\mathbf{G}$ explicitly. Rather, we will see that all needed quantities can be computed efficiently from the quantities obtained during a single standard for- and backward pass on the batch $\{(\mathbf{x}_i, y_i)\}_{i=1}^B$, namely the preactivations $\mathbf{A}^{(k)}$ and error $\mathbf{E}^{(k+1)}$.[5]

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.

E.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:

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

For (2), similar considerations give

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

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

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

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

5. 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.

E.4. Computational Complexity

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

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

E.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 [6].

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 9 and 10.

Appendix F. Software Validation

To validate that our algorithm of computing products between the damped, inverse Fisher and vectors 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 neural networks.

Appendix G. Additional Experiments

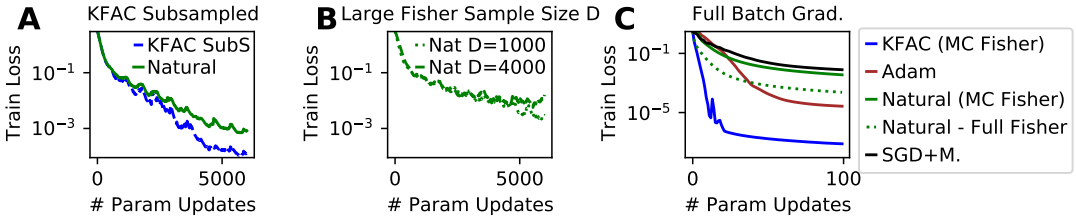

Figure 6: **Advantage of KFAC is not due to using more data to estimate the Fisher.** Same as Figure 2 but on MNIST rather than Fashion MNIST.

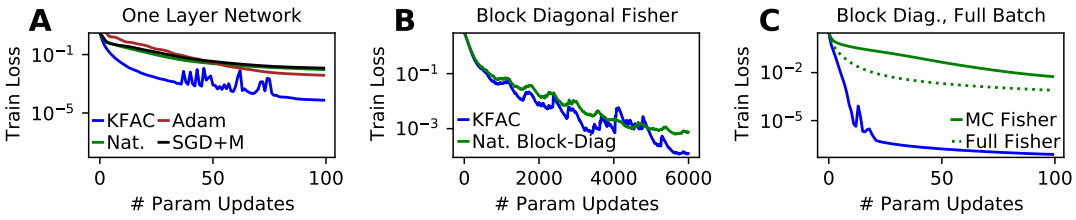

Figure 7: **Advantage of KFAC is not due to block-diagonal structure.** Same as Figure 3 but on MNIST rather than Fashion MNIST.

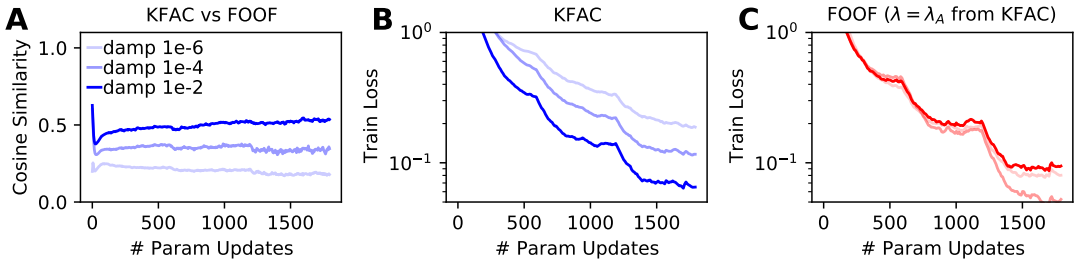

Figure 8: **Damping increases KFAC's performance as well as its similarity to our first order method FOOF.** Same as Figure 4 but on Fashion MNIST rather than MNIST

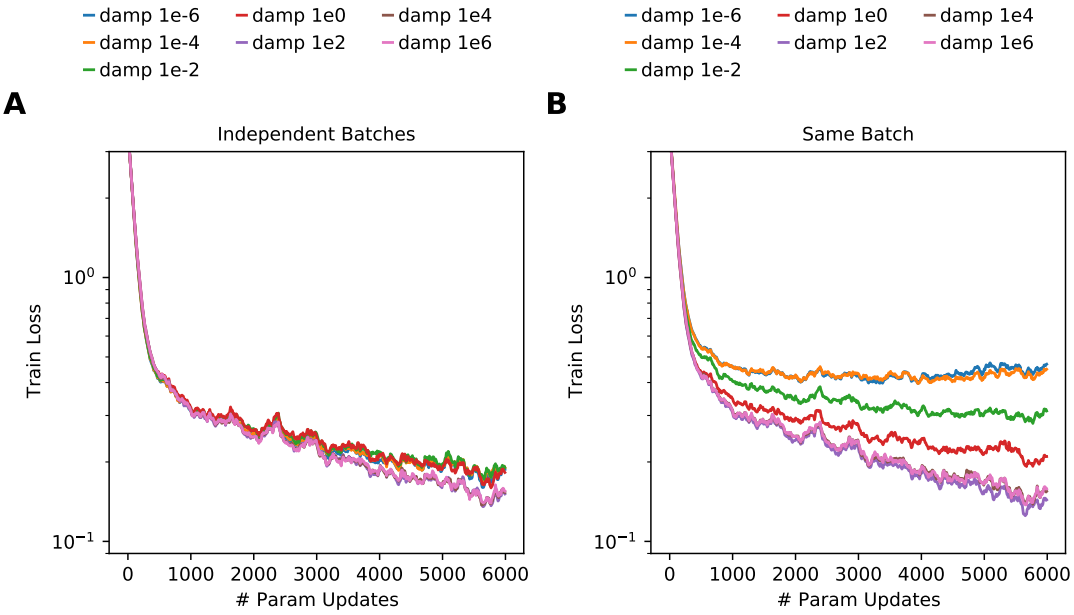

Figure 9: Comparison between computing Fisher Information and Gradient on independent or same mini-batches. In line with our usual procedure, the learning rate was re-optimised for each damping strength and each of the two options. This plot is on Fashion MNIST. For the same plot on MNIST see below.

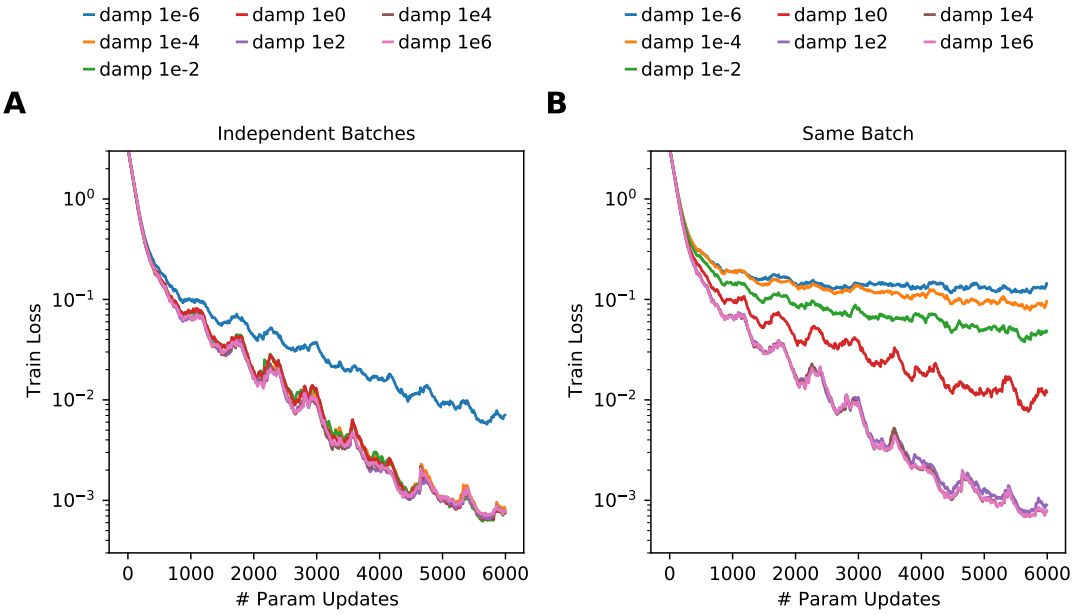

Figure 10: Same as Figure 9, but on MNIST.

References

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

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

[3] Shun-ichi Amari and Hiroshi Nagaoka. *Methods of information geometry*, volume 191. American Mathematical Soc., 2000.

[4] Jimmy Ba, Roger Grosse, and James Martens. Distributed second-order optimization using kronecker-factored approximations. 2016.

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

[6] Frederik Benzing. Unifying regularisation methods for continual learning. *arXiv preprint arXiv:2006.06357*, 2020.

[7] Alberto Bernacchia, Máté Lengyel, and Guillaume Hennequin. Exact natural gradient in deep linear networks and application to the nonlinear case. NIPS, 2019.

[8] Aleksandar Botev, Hippolyt Ritter, and David Barber. Practical gauss-newton optimisation for deep learning. In *International Conference on Machine Learning*, pages 557–565. PMLR, 2017.

[9] Erik Daxberger, Agustinus Kristiadi, Alexander Immer, Runa Eschenhagen, Matthias Bauer, and Philipp Hennig. Laplace redux–effortless bayesian deep learning. *arXiv preprint arXiv:2106.14806*, 2021.

[10] Guillaume Desjardins, Karen Simonyan, Razvan Pascanu, and Koray Kavukcuoglu. Natural neural networks. *arXiv preprint arXiv:1507.00210*, 2015.

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

[12] Thomas George, César Laurent, Xavier Bouthillier, Nicolas Ballas, and Pascal Vincent. Fast approximate natural gradient descent in a kronecker-factored eigenbasis. *arXiv preprint arXiv:1806.03884*, 2018.

[13] Donald Goldfarb, Yi Ren, and Achraf Bahamou. Practical quasi-newton methods for training deep neural networks. *arXiv preprint arXiv:2006.08877*, 2020.

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

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

[16] Yehuda Hoffman and Erez Ribak. Constrained realizations of gaussian fields-a simple algorithm. *The Astrophysical Journal*, 380:L5–L8, 1991.

[17] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

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

[19] Gaétan Marceau-Caron and Yann Ollivier. Practical riemannian neural networks. *arXiv preprint arXiv:1602.08007*, 2016.

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

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

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

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

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

[25] Yann Ollivier. True asymptotic natural gradient optimization. *arXiv preprint arXiv:1712.08449*, 2017.

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

[27] Razvan Pascanu and Yoshua Bengio. Revisiting natural gradient for deep networks. *arXiv preprint arXiv:1301.3584*, 2013.

[28] Yi Ren and Donald Goldfarb. Efficient subsampled gauss-newton and natural gradient methods for training neural networks. *arXiv preprint arXiv:1906.02353*, 2019.

[29] Hippolyt Ritter, Aleksandar Botev, and David Barber. Online structured laplace approximations for overcoming catastrophic forgetting. *arXiv preprint arXiv:1805.07810*, 2018.

[30] Hippolyt Ritter, Aleksandar Botev, and David Barber. 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, 2018.

[31] Nicolas Roux, Pierre-antoine Manzagol, and Yoshua Bengio. Topmoumoute online natural gradient algorithm. *Advances in Neural Information Processing Systems*, 20:849–856, 2007.

[32] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms, 2017.

[33] Guodong Zhang, James Martens, and Roger Grosse. Fast convergence of natural gradient descent for overparameterized neural networks. *arXiv preprint arXiv:1905.10961*, 2019.