ASDL: A Unified Interface for Gradient Preconditioning in PyTorch

Kazuki Osawa KAZUKI.OSAWA@INF.ETHZ.CH
ETH Zurich, Switzerland

Satoki Ishikawa RIVERSTONE@RIO.GSIC.TITECH.AC.JP
Rio Yokota RIOYOKOTA@GSIC.TITECH.AC.JP
Tokyo Institute of Technology, Japan

Shigang Li SHIGANGLI.CS@GMAIL.COM
Beijing University of Posts and Telecommunications, China

Torsten Hoefler HTOR@INF.ETHZ.CH
ETH Zurich, Switzerland

Abstract

Gradient preconditioning is a key technique to integrate the *second-order information* into gradients for improving and extending gradient-based learning algorithms. In deep learning, stochasticity, nonconvexity, and high dimensionality lead to a wide variety of gradient preconditioning methods, with implementation complexity and inconsistent performance and feasibility. We propose the Automatic *Second-order* Differentiation Library (ASDL), an extension library for PyTorch, which offers various implementations and a plug-and-play *unified interface* for gradient preconditioning. ASDL enables the study and structured comparison of a range of gradient preconditioning methods.

1. Introduction

Gradient preconditioning is a key technique for integrating *second-order* information such as *loss sharpness* (second-order derivatives) and *gradient covariance/second moment* (second-order statistics) into gradients. In deep learning in various domains such as vision [33], language [4, 38], graph [18], reinforcement learning [19], and quantum computing [43], gradient preconditioning has been reported to *improve* and *extend* gradient-based learning algorithms. The benefits of gradient preconditioning includes faster convergence of training [3, 29], more robust approximate Bayesian inference [20, 31, 48], regularization to avoid forgetting in continual learning [22, 35], identifying influential parameters and examples on model's output [16, 23], estimation of the mini-batch size with high data efficiency [30], and generic probabilistic prediction via gradient boosting [9].

To integrate the second-order information into the gradient $\boldsymbol{g}$, the gradient preconditioning applies the *preconditioning matrix* $\boldsymbol{P}$ to get the **preconditioned gradient** $\boldsymbol{Pg}$. In deep learning, where stochasticity, nonconvexity, and high dimensionality are inherent, there are a variety of choices for (i) the *curvature matrices* $\boldsymbol{C}$ containing various forms of second-order information (§2.1), (ii) the *representations* of $\boldsymbol{C}$ based on the neural network structures and matrix properties (§2.2), and (iii) the *solvers* for computing $\boldsymbol{Pg} \approx \boldsymbol{C}^{-1}\boldsymbol{g}$ (§2.3). This leads to a *diverse set* of gradient preconditioning methods (Figure 1,Table 1), each requiring *algorithm-specific* and *complex* implementations, making it challenging to incorporate them into existing training pipelines that usually use SGD-based gradient methods today. Furthermore, it is hard to switch between different methods in order to compare them. This implementation issue is critical because the *compute performance*, *pre-*

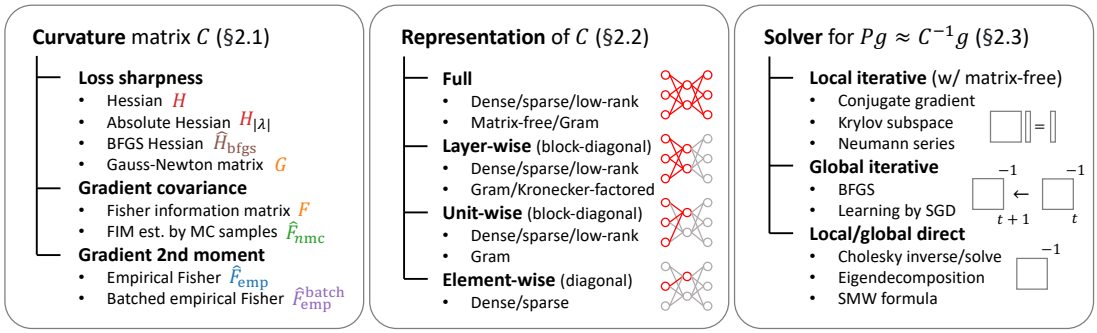

Figure 1: Three key components of gradient preconditioning in deep learning

diction accuracy, and *feasibility* (in terms of budget of time and memory) of methods are *highly dependent* on neural network architectures and specific training settings (§4).

To address this, we propose the Automatic *Second-order* Differentiation Library (**ASDL**), which extends PyTorch [37], an automatic-differentiation library, with a **unified interface** for gradient preconditioning using various curvature matrices, representations, and solvers (Figure 2) that is compatible with several types of training pipelines and neural network architectures.

2. Gradient Preconditioning in Deep Learning

Notations The *mini-batch empirical loss* $\mathcal{L}(\boldsymbol{\theta}) := \frac{1}{|\mathcal{B}|} \sum_{(\boldsymbol{x},\boldsymbol{t})\in\mathcal{B}} \ell(\boldsymbol{x},\boldsymbol{t};\boldsymbol{\theta}) = \langle \ell(\boldsymbol{x},\boldsymbol{t};\boldsymbol{\theta}) \rangle$ is the average of the per-example negative log-likelihood $\ell(\boldsymbol{x},\boldsymbol{t};\boldsymbol{\theta}) := -\log p_{\boldsymbol{\theta}}(\boldsymbol{t}|\boldsymbol{x}) =: h(f(\boldsymbol{x};\boldsymbol{\theta}),\boldsymbol{t})$ for each input-target pair $(\boldsymbol{x},\boldsymbol{t})$ in a mini-batch $\mathcal{B}$ sampled from the training set. $\boldsymbol{\theta} \in \mathbb{R}^P$ is the column vector containing the neural network parameters, $\langle\cdot\rangle$ represents the average over $\mathcal{B}$, $p_{\boldsymbol{\theta}}$ is model's predictive distribution, q is input distribution, and f is the neural network with K output neurons, $\boldsymbol{g} := \nabla\mathcal{L}(\boldsymbol{\theta}) \in \mathbb{R}^P$ is the *mini-batch gradient*, and $\boldsymbol{J}_f(\boldsymbol{x}) \in \mathbb{R}^{K\times P}$ is the Jacobian of f w.r.t. $\boldsymbol{\theta}$.

2.1. Curvature matrices

Loss sharpness The *Hessian* $\boldsymbol{H} := \nabla^2\mathcal{L} = \langle \nabla^2\ell(\boldsymbol{x},\boldsymbol{t};\boldsymbol{\theta}) \rangle \in \mathbb{R}^{P\times P}$ is the second-order derivative of $\mathcal{L}$ representing the *loss sharpness* [17], and the Newton direction is $\boldsymbol{Pg} = \boldsymbol{H}^{-1}\boldsymbol{g}$. The *absolute Hessian* $\boldsymbol{H}_{|\lambda|}$, which replaces the eigenvalues of $\boldsymbol{H}$ by their absolute values, is preferred in optimization of a nonconvex $\mathcal{L}$ to avoid saddle points [7, 8, 25]. The BFGS method estimates $\boldsymbol{H}$ (or $\boldsymbol{H}^{-1}$) with the *BFGS Hessian* $\hat{\boldsymbol{H}}_{\text{bfgs}}$ (or $\hat{\boldsymbol{H}}_{\text{bfgs}}{}^{-1}$), which is the accumulation of the changes in $\boldsymbol{g}$ (i.e., changes in the first-order derivatives) and $\boldsymbol{\theta}$ during iterative optimization of $\boldsymbol{\theta}$ with $\boldsymbol{Pg} = \hat{\boldsymbol{H}}_{\text{bfgs}}{}^{-1}\boldsymbol{g}$. The (generalized) *Gauss-Newton matrix* $\boldsymbol{G} := \left\langle \boldsymbol{J}_f(\boldsymbol{x})^\top \nabla^2_{\boldsymbol{y}} h(\boldsymbol{y},\boldsymbol{t})|_{\boldsymbol{y}=f(\boldsymbol{x})} \boldsymbol{J}_f(\boldsymbol{x}) \right\rangle$ [42], which ignores the second-order derivative of f w.r.t. $\boldsymbol{\theta}$ in $\boldsymbol{H}$ (i.e., views f as linear [13]) and is positive semi-definite, is also preferred in non-convex optimization [27].

Gradient covariance The *Fisher* $\boldsymbol{F} := \mathbb{E}_{q(\boldsymbol{x})}\left[\mathbb{E}_{p_{\boldsymbol{\theta}}(\boldsymbol{t}'|\boldsymbol{x})}\left[\nabla \log p_{\boldsymbol{\theta}}(\boldsymbol{t}'|\boldsymbol{x}) \nabla \log p_{\boldsymbol{\theta}}(\boldsymbol{t}'|\boldsymbol{x})^\top\right]\right] \in \mathbb{R}^{P\times P}$ is the *covariance* of gradient of log-likelihood $\nabla \log p_{\boldsymbol{\theta}}$. $\boldsymbol{F}$ is also the second-order derivative of the KL-divergence $D_{\text{KL}}(p_{\boldsymbol{\theta}}||p_{\boldsymbol{\theta}+\Delta\boldsymbol{\theta}})$ and is used as $\boldsymbol{C}$ in the natural gradient descent (NGD) [3]: $\boldsymbol{Pg} = \boldsymbol{F}^{-1}\boldsymbol{g}$. In practice, $\mathbb{E}_{q(\boldsymbol{x})}[\cdot]$ is estimated with $\langle\cdot\rangle$, and $\boldsymbol{F} = \boldsymbol{G}$ for cross-entropy and MSE loss [36], connecting the loss sharpness and gradient covariance perspectives in optimization [28].

Table 1: Representative gradient preconditioning methods in deep learning. "**KF**": Kronecker-factored. "**RR**": Rank reduction. "**SMW**": Sherman-Morrison-Woodbury formula. Methods analyzed in this study are underlined. See Table 3 for a more comprehensive list.

Method	**Curvature** matrix $\boldsymbol{C}$ (§2.1)		**Representation** of $\boldsymbol{C}$ (§2.2)		**Solver** for $\boldsymbol{Pg} \approx \boldsymbol{C}^{-1}\boldsymbol{g}$ (§2.3)	
	type	matrix	granularity	format	type	key operations
Hessian-free [27]	sharpness	$\boldsymbol{H}, \boldsymbol{G}$	full	matrix-free	local iterative	conjugate gradient
SMW-NG [39]	grad 2nd-m	$\hat{\boldsymbol{F}}_{\text{emp}}$	full	Gram, RR	local direct	SMW inverse
PSGD (KF) [25]	sharpness	$\boldsymbol{H}_{\lvert\lambda\rvert}$	layer	KF	global iterative	triangular solve, SGD
K-FAC [29]	grad cov, 2nd-m	$\hat{\boldsymbol{F}}_{1\text{mc}}, \hat{\boldsymbol{F}}_{\text{emp}}$	layer	KF	local/global direct	Cholesky inverse
Shampoo [15]	grad 2nd-m	$(\hat{\boldsymbol{F}}_{\text{emp}}^{\text{batch}})^{1/2}$	layer	KF	global direct	eigendecomp.
Adam [21]	grad 2nd-m	$(\hat{\boldsymbol{F}}_{\text{emp}}^{\text{batch}})^{1/2}$	element	dense	global direct	element-wise division

$\mathbb{E}_{p_{\boldsymbol{\theta}}(\boldsymbol{t}'|\boldsymbol{x})}[\cdot]$ involves K backward passes for $\nabla \log p_{\boldsymbol{\theta}}$ [6] (e.g., $K = 1000$ for ImageNet-1K), so $\boldsymbol{F}$ is often estimated with the *MC Fisher* $\hat{\boldsymbol{F}}_{n\text{mc}}$ with n Monte-Carlo (MC) samples of $\boldsymbol{t}_{\text{mc}} \sim p_{\boldsymbol{\theta}}(\boldsymbol{t}'|\boldsymbol{x})$.

Gradient second moment The *empirical Fisher* $\hat{\boldsymbol{F}}_{\text{emp}} := \left\langle \nabla \log p_{\boldsymbol{\theta}}(\boldsymbol{t}|\boldsymbol{x}) \nabla \log p_{\boldsymbol{\theta}}(\boldsymbol{t}|\boldsymbol{x})^\top \right\rangle = \left\langle \nabla \ell(\boldsymbol{x}, \boldsymbol{t}; \boldsymbol{\theta}) \nabla \ell(\boldsymbol{x}, \boldsymbol{t}; \boldsymbol{\theta})^\top \right\rangle \in \mathbb{R}^{P \times P}$ is the *second moment* of *per-example* empirical gradient. It can be computed during the backward pass for $\nabla \mathcal{L}$ and is preferred in large-scale settings [33, 38]. As $\hat{\boldsymbol{F}}_{\text{emp}}$ is no longer centered (i.e., $\langle \nabla \ell(\boldsymbol{x}, \boldsymbol{t}; \boldsymbol{\theta}) \rangle \neq \boldsymbol{0}$), it is claimed not to capture the useful second-order information for optimization [**?**] while it is empirically observed that NGD with $\hat{\boldsymbol{F}}_{\text{emp}}$ still achieves the fast convergence with smoothed $\boldsymbol{t}$ [34, 38]. Adaptive gradient methods such as Adam [21] and Shampoo [15] use the *batched empirical Fisher* $\hat{\boldsymbol{F}}_{\text{emp}}^{\text{batch}}(T) := \sum_{t=1}^{T} \alpha_t \boldsymbol{g}_t \boldsymbol{g}_t^\top$ ($0 \leq \alpha_t \leq 1$, $\boldsymbol{g}_t$ is for $\mathcal{B}_t$ at t-th training step), an online estimate of the *second moment* of *mini-batch* empirical gradient: $\boldsymbol{P}\boldsymbol{g}_T = (\hat{\boldsymbol{F}}_{\text{emp}}^{\text{batch}}(T))^{-1/2} \boldsymbol{g}_T$. $\hat{\boldsymbol{F}}_{\text{emp}}^{\text{batch}}$ looses the second-order information when the mini-batch size $|\mathcal{B}|$ is large [13], but it is also empirically observed that Shampoo achieves a faster convergence than first-order optimizers (SGD, LAMB [47]) in large-batch training [4][1].

2.2. Representations of matrices

It is infeasible to materialize $\boldsymbol{C} \in \mathbb{R}^{P \times P}$ and directly invert it ($\boldsymbol{C}^{-1}$) with the $\mathcal{O}(P^3)$ cost for deep neural networks with a massive number of parameters P (e.g., billions). To make practical use of (a portion of) the information in $\boldsymbol{C}$, there are various *matrix representations* using block-diagonal approximation, compact format, or both.

Full matrix Typical compact formats for exploiting the *full* $\boldsymbol{C}$ include matrix-vector products (*matrix-free*) (e.g., Hessian-free) and *Gram matrices* with rank reduction (e.g., SMW-NG).

Layer-/unit-/element-wise block-diagonal matrix Granularity of diagonal blocks are often per neural network *layer*, per *unit*, or per *element* of $\boldsymbol{\theta}$ (i.e., diagonal, e.g., Adam). *Kronecker-factored matrices* (e.g., PSGD, K-FAC, Shampoo) is one of the most common formats for layer-wise blocks.

1. See [13] for a more detailed description of these curvature matrices.

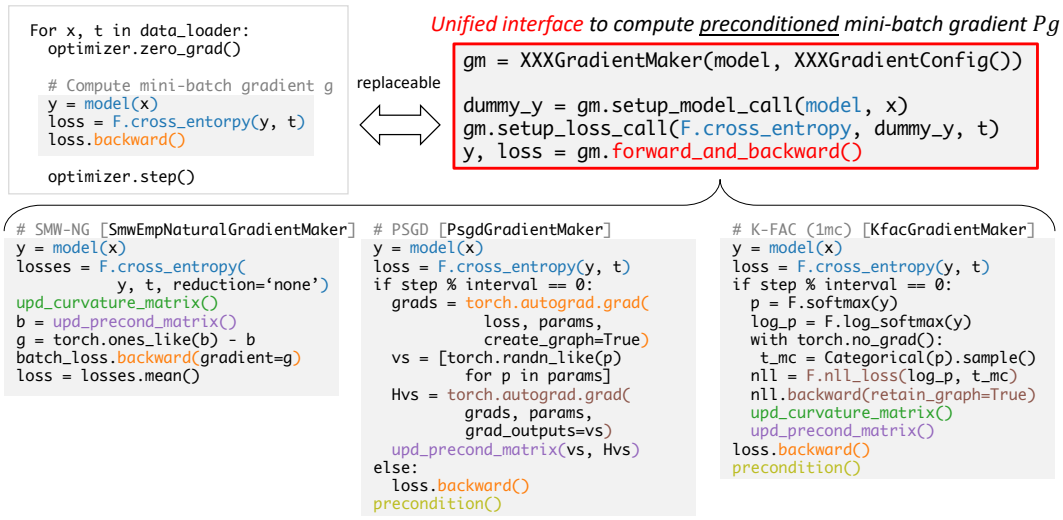

Figure 2: **Unified interface for gradient preconditioning in PyTorch.** `XXXGradientMaker` ("XXX": algorithm name), offered by ASDL, hides *algorithm-specific* and *complex* operations for $\boldsymbol{Pg}$ in a *unified* way. For training without gradient preconditioning, `GradientMaker` computes $\boldsymbol{g}$ with *the same interface* (i.e., no need to switch scripts). For ease of comparison, the color scheme for operations is consistent with Figure 3.

2.3. Solvers for preconditioning gradient

Local vs. global *Solvers* to compute $\boldsymbol{Pg} \approx \boldsymbol{C}^{-1}\boldsymbol{g}$ are first classified by the scope of information captured by $\boldsymbol{C}$, i.e., *local* information within one $\mathcal{B}$ vs. *global* information associated with multiple $\mathcal{B}$s observed through learning. By definition, solvers with $\hat{\boldsymbol{H}}_{\text{bfgs}}$ or $\hat{\boldsymbol{F}}^{\text{batch}}_{\text{emp}}$ are global solvers.

Iterative vs. direct Solvers are also classified by the type of linear solver for $\boldsymbol{C}_{repr}\boldsymbol{x} = \boldsymbol{g}$, i.e., *iterative* vs. *direct*, where $\boldsymbol{C}_{repr}$ is a certain representation (§2.2) of selected $\boldsymbol{C}$ (§2.1) containing local or global information. An iterative solver uses the matrix-free format for local while it materializes $\boldsymbol{C}_{repr}$ for global. A damping $\tau\boldsymbol{I}$ ($\tau > 0$) is often added to $\boldsymbol{C}_{repr}$ to improve numerical stability and/or guarantee positive definiteness ($(\boldsymbol{C}_{repr} + \tau\boldsymbol{I}) \succ 0$). This allows a fast direct solver using Cholesky decomposition (e.g., K-FAC) or SMW formula (e.g., SMW-NG) to be applied.

3. Unified Interface for Gradient Preconditioning in ASDL

Our Automatic *Second-order* Differentiation Library (**ASDL**)[2] implements gradient preconditioning methods listed in Table 1. Figure 2 shows a standard training pipeline in PyTorch with mini-batch gradients, the (simplified) operations in SMW-NG, PSGD, and K-FAC (with $\hat{\boldsymbol{F}}_{\text{1mc}}$) (see Appendix A for details), and the **unified interface**, `XXXGradientMaker`, which enables an easy integration of gradient preconditioning by hiding the *algorithm-specific* and *complex* operations. Each method exhibits compute performance, prediction accuracy, and feasibility depending *highly* on tasks (§4), so capability to flexibly switch/compare methods is important.

2. https://github.com/kazukiosawa/asdfghjkl

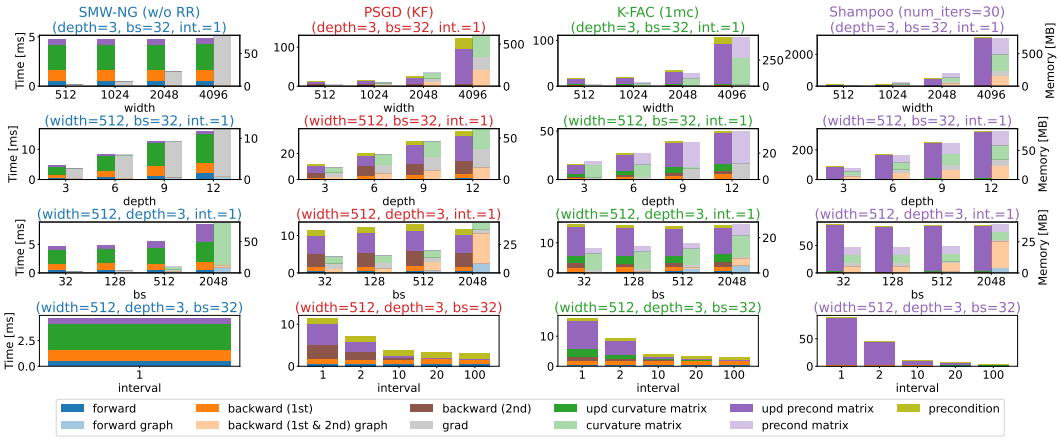

Figure 3: **One-step time and peak memory consumption** (left and right bar at each point in each box, respectively) for MLPs on MNIST. In the bottom row (scaling `interval`), the memory is not shown because it is constant. Times are the averages of 100 runs on an NVIDIA RTX 3090. An MLP has `depth` layers ($784-$`width`$-\cdots-$`width`-10) with ReLU non-linearities. `width`, `depth`, mini-batch size (`bs`), and matrix update interval (`interval`) are independently scaled from the base values (width=512, depth=3, bs=32, interval=1) for each method (except for SMW-NG, which can only take an interval=1).

4. Case Studies with ASDL

Using ASDL, we compare gradient preconditioning methods for optimization, i.e., adaptive gradient methods (with $\hat{\boldsymbol{F}}_{\text{emp}}^{\text{batch}}$) and second-order optimization methods (with other $\boldsymbol{C}$) in vision tasks.

Time and memory Figure 3 shows the time per step, peak memory consumption, and their breakdown in training MLPs on MNIST [24] using the representative methods summarized in Table 1. Each method behaves differently with respect to the scaling of the MLP's `width` (number of neurons) and `depth` (number of layers), the mini-batch size (`bs`), and the matrix update interval (`interval`). See Appendix A for a more detailed description of operations in each method.

Neural network training Table 2 shows the training results on MNIST and CIFAR-10 with various neural network architectures and methods.[3] For each task, the mini-batch size, learning rate, number of epochs, matrix update interval (if applicable), damping τ (if applicable), and number of power iterations (Shampoo) are tuned with Bayesian optimization under a predefined time budget.[4]

5. Discussion

By ASDL, we observe that no gradient preconditioning method is always superior (in the sense of compute performance, prediction accuracy, and feasibility) to another — it is critical to flexibly switch and compare methods. The extension of this work to distributed and mixed-precision training, where time and numerical stability bottlenecks change [4, 45], is an important future direction.

3. SMW-NG for ViT-tiny and MLP-Mixer-base, sequence models, are not supported in ASDL yet, but SMW-based methods are often infeasible because "bs" (in Figure 3) is the number of tokens = mini-batch size $\times$ sequence length.
4. See Appendix B for the details of experimental settings.

Table 2: **The test accuracy (and training time)** for models achieving the best validation accuracy. For each task, the best accuracy is **bolded** and the shortest time is underlined. "w": width.

Method	MNIST			CIFAR-10			
	MLP (w=128)	MLP (w=512)	MLP (w=2048)	ResNet18	WideResNet	ViT-tiny	MLP-Mixer-base
SGD	97.94 (48.8s)	**98.42** (101.5s)	98.46 (56.4s)	95.85 (33m)	96.97 (178m)	97.81 (15.7m)	96.71 (72m)
SMW-NG (w/o RR)	97.99 (160.2s)	97.74 (48.23s)	98.27 (164.7s)	94.27 (92m)	94.93 (454m)	-	-
PSGD (KF)	98.05 (42.7s)	98.33 (86.9s)	98.44 (26.5s)	96.07 (44m)	**96.99** (276m)	**97.95** (28.1m)	**97.33** (109m)
K-FAC (1mc)	97.94 (45.4s)	**98.42** (98.8s)	98.51 (15.0s)	95.97 (32m)	96.95 (167m)	97.68 (8.0m)	97.14 (70m)
Shampoo	**98.13** (739.6s)	98.35 (259.7s)	**98.55** (88.9s)	**96.38** (275m)	96.74 (113m)	97.93 (39.9m)	96.81 (111m)

Acknowledgments

K.O. is supported by an ETH Zurich Postdoctoral Fellowship. This work was supported by the PASC DaCeMI project, received EuroHPC-JU funding under grant MAELSTROM, No. 955513 as well as funding from the European Research Council under Project PSAP, No. 101002047, and we thank the Swiss National Supercomputing Center (CSCS) for supporting us with compute infrastructure. This work is supported by JST CREST Grant Number JPMJCR2112.

References

[1] Naman Agarwal, Brian Bullins, and Elad Hazan. Second-Order Stochastic Optimization for Machine Learning in Linear Time. *The Journal of Machine Learning Research*, 18(1):4148–4187, 2017.

[2] Naman Agarwal, Brian Bullins, Xinyi Chen, Elad Hazan, Karan Singh, Cyril Zhang, and Yi Zhang. Efficient Full-Matrix Adaptive Regularization. In *Proceedings of International Conference on Machine Learning (ICML)*, pages 102–110, 2019.

[3] Shun-ichi Amari. Natural Gradient Works Efficiently in Learning. *Neural Computation*, 10 (2):251–276, 1998.

[4] Rohan Anil, Vineet Gupta, Tomer Koren, Kevin Regan, and Yoram Singer. Scalable Second Order Optimization for Deep Learning. *arXiv preprint arXiv:2002.09018*, 2021. URL http://arxiv.org/abs/2002.09018.

[5] Aleksandar Botev, Hippolyt Ritter, and David Barber. Practical Gauss-Newton Optimisation for Deep Learning. In *Proceedings of International Conference on Machine Learning (ICML)*, pages 557–565, 2017.

[6] Felix Dangel, Frederik Kunstner, and Philipp Hennig. BackPACK: Packing more into Backprop. In *International Conference on Learning Representations (ICLR)*, 2020. URL https://openreview.net/forum?id=BJlrF24twB.

[7] Yann N Dauphin, Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, Surya Ganguli, and Yoshua Bengio. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In *Advances in Neural Information Processing Systems*, volume 27, 2014.

[8] Yann N. Dauphin, Harm de Vries, and Yoshua Bengio. Equilibrated adaptive learning rates for non-convex optimization. In *Advances in Neural Information Processing Systems*, volume 28, August 2015.

[9] Tony Duan, Anand Avati, Daisy Yi Ding, Khanh K. Thai, Sanjay Basu, Andrew Y. Ng, and Alejandro Schuler. NGBoost: Natural Gradient Boosting for Probabilistic Prediction. In *International Conference on Machine Learning (ICML)*, pages 2690–2700, June 2020.

[10] John Duchi, Elad Hazan, and Yoram Singer. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *Journal of Machine Learning Research*, 12:2121–2159, 2011.

[11] Elias Frantar, Eldar Kurtic, and Dan Alistarh. Efficient Matrix-Free Approximations of Second-Order Information, with Applications to Pruning and Optimization. In *Advances in Neural Information Processing Systems*, volume 34, pages 14873–14886, July 2021.

[12] Thomas George, César Laurent, Xavier Bouthillier, Nicolas Ballas, and Pascal Vincent. Fast Approximate Natural Gradient Descent in a Kronecker Factored Eigenbasis. In *Advances in Neural Information Processing Systems*, pages 9550–9560, 2018.

[13] Roger Grosse. Chapter 3: Metrics, 2022. URL https://www.cs.toronto.edu/~rgrosse/courses/csc2541_2021/readings/L03_metrics.pdf.

[14] Roger B Grosse and Ruslan Salakhutdinov. Scaling Up Natural Gradient by Sparsely Factorizing the Inverse Fisher Matrix. In *International Conference on Machine Learning (ICML)*, pages 2304–2313, 2015.

[15] Vineet Gupta, Tomer Koren, and Yoram Singer. Shampoo: Preconditioned Stochastic Tensor Optimization. In *Proceedings of International Conference on Machine Learning (ICML)*, pages 1842–1850, 2018.

[16] Babak Hassibi and David G. Stork. Second order derivatives for network pruning: Optimal Brain Surgeon. In *Advances in Neural Information Processing Systems*, pages 164–171. 1993.

[17] Sepp Hochreiter and Jurgen Schmidhuber. Flat Minima. *Neural Computation*, 9(1):1–42, 1997.

[18] Mohammad Rasool Izadi, Yihao Fang, Robert Stevenson, and Lizhen Lin. Optimization of Graph Neural Networks with Natural Gradient Descent. In *IEEE International Conference on Big Data*, pages 171–179, 2020.

[19] Sham M Kakade. A Natural Policy Gradient. In T. G. Dietterich, S. Becker, and Z. Ghahramani, editors, *Advances in Neural Information Processing Systems*, pages 1531–1538, 2002.

[20] Mohammad Emtiyaz Khan, Didrik Nielsen, Voot Tangkaratt, Wu Lin, Yarin Gal, and Akash Srivastava. Fast and Scalable Bayesian Deep Learning by Weight-Perturbation in Adam. In *International Conference on Machine Learning (ICML)*, pages 2616–2625, 2018.

[21] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. In *International Conference on Learning Representations (ICLR)*, 2015.

[22] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A. Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, Demis Hassabis, Claudia Clopath, Dharshan Kumaran, and Raia Hadsell. Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences*, 114 (13):3521–3526, 2017.

[23] Pang Wei Koh and Percy Liang. Understanding Black-box Predictions via Influence Functions. In *Proceedings of International Conference on Machine Learning (ICML)*, pages 1885–1894, 2017.

[24] Yann LeCun, Corinna Cortes, and CJ Burges. The MNIST database of handwritten digits, 1998. URL http://yann.lecun.com/exdb/mnist.

[25] Xi-Lin Li. Preconditioned Stochastic Gradient Descent. *IEEE Transactions on Neural Networks and Learning Systems*, 29(5):1454–1466, 2018. ISSN 2162-237X, 2162-2388. doi: 10.1109/TNNLS.2017.2672978.

[26] Dong C. Liu and Jorge Nocedal. On the limited memory BFGS method for large scale optimization. *Mathematical Programming*, 45(1-3):503–528, August 1989. ISSN 0025-5610, 1436-4646. doi: 10.1007/BF01589116. URL http://link.springer.com/10.1007/BF01589116.

[27] James Martens. Deep learning via Hessian-free optimization. In *Proceedings of International Conference on Machine Learning (ICML)*, pages 735–742, 2010.

[28] James Martens. New Insights and Perspectives on the Natural Gradient Method. *Journal of Machine Learning Research*, 21(146):1–76, 2020.

[29] James Martens and Roger Grosse. Optimizing Neural Networks with Kronecker-factored Approximate Curvature. In *Proceedings of International Conference on Machine Learning (ICML)*, pages 2408–2417, 2015.

[30] Sam McCandlish, Jared Kaplan, Dario Amodei, and OpenAI Dota Team. An Empirical Model of Large-Batch Training. *arXiv preprint arXiv:1812.06162*, 2018.

[31] Zachary Nado, Jasper Snoek, Bowen Xu, Roger Grosse, David Duvenaud, and James Martens. Stochastic Gradient Langevin Dynamics That Exploit Neural Network Structure. In *International Conference on Learning Representations (ICLR) Workshop track*, 2018.

[32] Y. Ollivier. Riemannian metrics for neural networks I: feedforward networks. *Information and Inference*, 4(2):108–153, June 2015. ISSN 2049-8764, 2049-8772. doi: 10.1093/imaiai/iav006. URL https://academic.oup.com/imaiai/article-lookup/doi/10.1093/imaiai/iav006.

[33] Kazuki Osawa, Yohei Tsuji, Yuichiro Ueno, Akira Naruse, Rio Yokota, and Satoshi Matsuoka. Large-Scale Distributed Second-Order Optimization Using Kronecker-Factored Approximate Curvature for Deep Convolutional Neural Networks. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 12359–12367, 2019.

[34] Kazuki Osawa, Yohei Tsuji, Yuichiro Ueno, Akira Naruse, Chuan-Sheng Foo, and Rio Yokota. Scalable and Practical Natural Gradient for Large-Scale Deep Learning. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(1):404–415, 2022.

[35] Pingbo Pan, Siddharth Swaroop, Alexander Immer, Runa Eschenhagen, Richard E. Turner, and Mohammad Emtiyaz Khan. Continual Deep Learning by Functional Regularisation of Memorable Past. In *Advances in Neural Information Processing Systems*, pages 4453–4464, 2020.

[36] Razvan Pascanu and Yoshua Bengio. Revisiting Natural Gradient for Deep Networks. In *International Conference on Learning Representations (ICLR)*, 2014. URL https://openreview.net/forum?id=vz8AumxkAfz5U.

[37] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 8026–8037, 2019.

[38] J. Gregory Pauloski, Lei Huang, Weijia Xu, Kyle Chard, Ian Foster, and Zhao Zhang. Deep Neural Network Training with Distributed K-FAC. *IEEE Transactions on Parallel and Distributed Systems*, pages 1–1, 2022. ISSN 1558-2183. doi: 10.1109/TPDS.2022.3161187. Conference Name: IEEE Transactions on Parallel and Distributed Systems.

[39] Yi Ren and Donald Goldfarb. Efficient Subsampled Gauss-Newton and Natural Gradient Methods for Training Neural Networks. *arXiv preprint arXiv:1906.02353*, 2019. URL http://arxiv.org/abs/1906.02353.

[40] Yi Ren and Donald Goldfarb. Tensor Normal Training for Deep Learning Models. In *Advances in Neural Information Processing Systems*, volume 34, pages 26040–26052, 2021.

[41] Nicolas L. Roux, Pierre-antoine Manzagol, and Yoshua Bengio. Topmoumoute Online Natural Gradient Algorithm. In J. C. Platt, D. Koller, Y. Singer, and S. T. Roweis, editors, *Advances in Neural Information Processing Systems 20*, pages 849–856. Curran Associates, Inc., 2008.

[42] Nicol N. Schraudolph. Fast Curvature Matrix-Vector Products for Second-Order Gradient Descent. *Neural Computation*, 14(7):1723–1738, July 2002. ISSN 0899-7667, 1530-888X. doi: 10.1162/08997660260028683. URL http://www.mitpressjournals.org/doi/10.1162/08997660260028683.

[43] James Stokes, Josh Izaac, Nathan Killoran, and Giuseppe Carleo. Quantum Natural Gradient. *Quantum*, 4:269, May 2020. ISSN 2521-327X. doi: 10.22331/q-2020-05-25-269.

[44] Zedong Tang, Fenlong Jiang, Maoguo Gong, Hao Li, Yue Wu, Fan Yu, Zidong Wang, and Min Wang. SKFAC: Training Neural Networks With Faster Kronecker-Factored Approximate Curvature. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 13479–13487, 2021.

[45] Yuichiro Ueno, Kazuki Osawa, Yohei Tsuji, Akira Naruse, and Rio Yokota. Rich Information is Affordable: A Systematic Performance Analysis of Second-order Optimization Using K-FAC. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2145–2153, August 2020. ISBN 978-1-4503-7998-4. doi: 10.1145/3394486.3403265. URL https://dl.acm.org/doi/10.1145/3394486.3403265.

[46] Minghan Yang, Dong Xu, Zaiwen Wen, Mengyun Chen, and Pengxiang Xu. Sketch-Based Empirical Natural Gradient Methods for Deep Learning. *Journal of Scientific Computing*, 92 (3):94, September 2022. ISSN 0885-7474, 1573-7691. doi: 10.1007/s10915-022-01911-x. URL https://link.springer.com/10.1007/s10915-022-01911-x.

[47] Yang You, Igor Gitman, and Boris Ginsburg. Large Batch Training of Convolutional Networks. *arXiv preprint arXiv:1708.03888*, 2017. URL http://arxiv.org/abs/1708.03888.

[48] Guodong Zhang, Shengyang Sun, David Duvenaud, and Roger Grosse. Noisy Natural Gradient as Variational Inference. In *Proceedings of International Conference on Machine Learning (ICML)*, pages 5852–5861, 2018.

Table 3: **Gradient preconditioning methods in deep learning**. "**KF-io**": input-output Kronecker-factored. "**KF-dim**": dimension-wise Kronecker-factored. "**RR**": rank reduction. `SMW`: Sherman-Morrison-Woodbury formula. "**L**": local "**G**": global "**iter**": iterative. "**NN ind.**": how to calculate $\boldsymbol{Pg}$ is independent of the neural network architecture. If the matrix $\boldsymbol{C}$ is "*full*" granularity, it can be applied to any granularity (e.g., PSGD (KF), TONGA (unit) introduced by the authors), but some methods require additional derivation, computation and memory costs.

Method	**Curvature** matrix $\boldsymbol{C}$ (§2.1)		**Representation** of $\boldsymbol{C}$ (§2.2)		**Solver** for $\boldsymbol{Pg} \approx \boldsymbol{C}^{-1}\boldsymbol{g}$ (§2.3)		NN ind.
	type	matrix	granularity	format	type	key operations	
LiSSA [1]	sharpness	$\boldsymbol{H}$	full	dense	G iter	Neumann series	✓
PSGD [25]	sharpness	$\boldsymbol{H}_{\lvert\lambda\rvert}$	full	dense	G iter	triangular solve & SGD	✓
Neumann optimizer [**?**]	sharpness	$\boldsymbol{H}$	full	matrix-free	L iter	Neumann series	✓
Hessian-free [27]	sharpness	$\boldsymbol{H}, \boldsymbol{G}$	full	matrix-free	L iter	conjugate gradient	✓
KSD [**?**]	sharpness	$\boldsymbol{H}, \boldsymbol{G}$	full	matrix-free	L iter	Krylov subspace method	✓
L-BFGS [26]	sharpness	$\hat{\boldsymbol{H}}_{\text{bfgs}}$	full	matrix-free	G iter	approx. BFGS	✓
SMW-GN [39]	sharpness	$\boldsymbol{G}$	full	Gram, RR	L direct	SMW inverse	✗
SMW-NG [39]	grad 2$^{\text{nd}}$m	$\hat{\boldsymbol{F}}_{\text{emp}}$	full	Gram, RR	L direct	SMW inverse	✗
TONGA [41]	grad 2$^{\text{nd}}$m	$\hat{\boldsymbol{F}}_{\text{emp}}$	full	Gram, RR	G direct	SMW solve & eigendecomp.	✓
M-FAC [11]	grad 2$^{\text{nd}}$m	$\hat{\boldsymbol{F}}^{\text{batch}}_{\text{emp}}$	full	Gram, RR	G direct	SMW solve	✓
GGT [2]	grad 2$^{\text{nd}}$m	$(\hat{\boldsymbol{F}}^{\text{batch}}_{\text{emp}})^{1/2}$	full	Gram, RR	G direct	SMW solve	✓
FANG [14]	grad cov	$\hat{\boldsymbol{F}}_{n\text{mc}}$	full	sparse	L/G direct	incomplete Cholesky	✓
PSGD (KF) [25]	sharpness	$\boldsymbol{H}_{\lvert\lambda\rvert}$	layer	KF-io	G iter	triangular solve & SGD	✗
K-BFGS [**?**]	sharpness	$\hat{\boldsymbol{H}}_{\text{bfgs}}$	layer	KF-io	G iter	BFGS	✗
K-FAC [29]	grad cov, 2$^{\text{nd}}$m	$\hat{\boldsymbol{F}}_{n\text{mc}}, \hat{\boldsymbol{F}}_{\text{emp}}$	layer	KF-io	L/G direct	Cholesky inverse	✗
KFLR [5]	grad cov	$\boldsymbol{F}$	layer	KF-io	L/G direct	Cholesky inverse	✗
KFRA [5]	grad cov, 2$^{\text{nd}}$m	$\hat{\boldsymbol{F}}_{n\text{mc}}, \hat{\boldsymbol{F}}_{\text{emp}}$	layer	KF-io	L/G direct	Cholesky inverse & recursion	✗
EKFAC [12]	grad cov, 2$^{\text{nd}}$m	$\hat{\boldsymbol{F}}_{\text{emp}}$	layer	KF-io	L/G direct	eigendecomp. (or SVD)	✗
SKFAC [44]	grad cov, 2$^{\text{nd}}$m	$\hat{\boldsymbol{F}}_{1\text{mc}}, \hat{\boldsymbol{F}}_{\text{emp}}$	layer	KF-io, RR	L direct	SMW inverse & reduction	✗
SENG [46]	grad 2$^{\text{nd}}$m	$\hat{\boldsymbol{F}}_{\text{emp}}$	layer	Gram, RR	L/G direct	SMW inverse & sketching	✗
TNT [40]	grad cov, 2$^{\text{nd}}$m	$\hat{\boldsymbol{F}}_{n\text{mc}}, \hat{\boldsymbol{F}}_{\text{emp}}$	layer	KF-dim	L direct	Cholesky inverse	✓
Shampoo [15]	grad 2$^{\text{nd}}$m	$(\hat{\boldsymbol{F}}^{\text{batch}}_{\text{emp}})^{1/2}$	layer	KF-dim	G direct	eigendecomp.	✓
unit-wise NG [32]	grad cov, 2$^{\text{nd}}$m	$\hat{\boldsymbol{F}}_{n\text{mc}}, \hat{\boldsymbol{F}}_{\text{emp}}$	unit	dense	L/G direct	Cholesky inverse	✗
TONGA (unit) [41]	grad 2$^{\text{nd}}$m	$\hat{\boldsymbol{F}}_{\text{emp}}$	unit	Gram, RR	G direct	SMW solve & eigendecomp.	✗
AdaHessian [**?**]	sharpness	$\boldsymbol{H}$	element	dense	G direct	element-wise division	✓
SFN [7]	sharpness	$\boldsymbol{H}_{\lvert\lambda\rvert}$	element	dense	L/G direct	element-wise division	✓
Equilibrated SGD [8]	sharpness	$\boldsymbol{H}_{\lvert\lambda\rvert}$	element	dense	L/G direct	element-wise division	✓
AdaGrad [10]	grad 2$^{\text{nd}}$m	$(\hat{\boldsymbol{F}}^{\text{batch}}_{\text{emp}})^{1/2}$	element	dense	G direct	element-wise division	✓
Adam [21]	grad 2$^{\text{nd}}$m	$(\hat{\boldsymbol{F}}^{\text{batch}}_{\text{emp}})^{1/2}$	element	dense	G direct	element-wise division	✓

Appendix A. Target gradient preconditioning methods

We describe PyTorch-style pseudo codes for SMW-NG [39] (algorithm 1), PSGD [25] (algorithm 2), K-FAC (with $\hat{\boldsymbol{F}}_{1\text{mc}}$) [29] (algorithm 3), and Shampoo [15] (algorithm 4). The color scheme for operations are consistent with that used in Figure 3

Figure 4 shows the throughput (image/s) of gradient preconditioning methods while varying mini-batch size, matrix update interval, and number of power iterations.

Algorithm 1 SMW-NG

procedure SMWEMPNATURALGRADIENTMAKER(the old parameters $\theta_{[old]}$)
    y = model(x)
    losses = F.cross_entropy(y, t, reduction='none')
    G = 0
    torch.autograd.grad(losses.sum(), model.parameters(), retain_graph=True)
    # for each layer during torch.autograd.grad()
        G += torch.mm(act, act$^\top$).mul(torch.mm(err, err$^\top$))
    v = G.sum(dim=1)
    b = cholesky_solve(G, v, damping)
    ones = torch.ones_like(b)
    batch_loss.backward(gradient=(ones - b) / damping)
    update = [p.grad for p in model.parameters()]
    $\theta_{[new]}$ = $\theta_{[old]}$ - μ*update
end procedure

Algorithm 2 PSGD

procedure PSGDGRADIENTMAKER(inputs: the old preconditioner $Q_{[old]}$, the old parameters $\theta_{[old]}$)
    y = model(x)
    loss = F.cross_entropy(y, t)
    grad = torch.autograd.grad(loss, model.parameters(), create_graph=True)
    vs = [torch.randn_like(p) for p in model.parameters()]
    Hvs = torch.autograd.grad(grads, params, grad_outputs=vs)
    a = torch.mm($Q_{[old]}$, Hvs)
    b = torch.linalg.solve_triangular($Q_{[old]}^\top$, vs, upper=False)
    $\nabla\epsilon$ = 2*torch.tril(aa$^\top$-bb$^\top$)
    $Q_{[new]}$ = $Q_{[old]}$ - scalar*torch.mm($\nabla\epsilon$, $Q_{[old]}$)
    update = torch.linalg.multi_dot([$Q_{[new]}^\top$, $Q_{[new]}$, grad])
    $\theta_{[new]}$ = $\theta_{[old]}$ - μ*update
end procedure

Algorithm 3 K-FAC (1mc)

procedure KFACGRADIENTMAKER(inputs: the old parameters $\theta_{[old]}$)
    y = model(x)
    loss = F.cross_entropy(y, t)
    p = F.softmax(y)
    log_p = F.log_softmax(y)
    with torch.no_grad():
        t_mc = torch.distribution.Categorical(p).sample()
    nll = F.nll_loss(log_p, t_mc)
    nll.backward(retain_graph=True)
    # for each layer during .backward()
        A = torch.mm(act$^\top$, act)
        B = torch.mm(err$^\top$, err)
        Ainv = torch.cholesky_inverse(torch.linalg.cholesky(A))
        Binv = torch.cholesky_inverse(torch.linalg.cholesky(B))
    loss.backward()
    # for each layer
        grad = [param.grad for param in layer]
        update = torch.linalg.multi_dot([Binv, grad, Ainv])
    $\theta_{[new]}$ = $\theta_{[old]}$ - μ*update
end procedure

Algorithm 4 Shampoo

procedure SHAMPOOOPTIMIZER(inputs: the old left-hand-side preconditioner $L_{[old]}$, the old right-hand-side preconditioner $R_{[old]}$, the old parameters $\theta_{[old]}$)
    y = model(x)
    loss = F.cross_entropy(y, t)
    loss.backward()
    # for each layer
        grad = [param.grad for param in layer]
        $L_{[new]}$ = $L_{[old]}$ + torch.mm(grad, grad$^\top$)
        $R_{[new]}$ = $R_{[old]}$ + torch.mm(grad$^\top$, grad)
        Compute $L_{[new]}^{-1/4}$ from the eigendecomposition by `num_iters` power iterations
        Compute $R_{[new]}^{-1/4}$ from the eigendecomposition by `num_iters` power iterations
        update = torch.linalg.multi_dot([$L_{[new]}^{-1/4}$, grad, $R_{[new]}^{-1/4}$])
    $\theta_{[new]}$ = $\theta_{[old]}$ - μ*update
end procedure

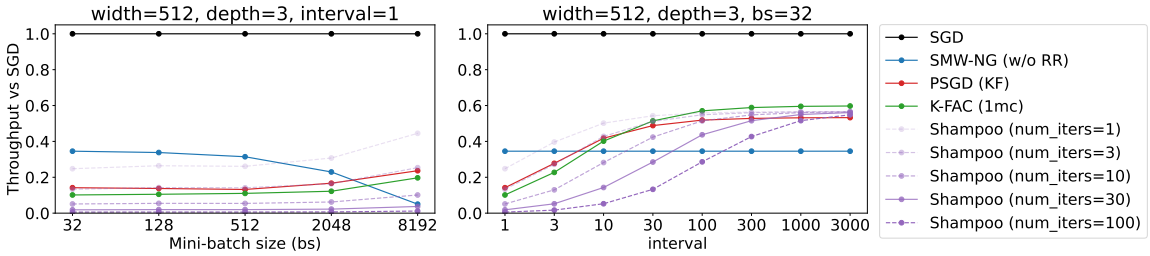

Figure 4: Throughput (image/s) of gradient preconditioning methods compared to SGD with various mini-batch sizes and matrix update intervals in the same setting as Figure 3. For Shampoo we observe the effect on throughput of varying the number of power iterations (`num_iters`) for the eigendecomposition of the Kronecker factors.

Appendix B. Experimental settings

We split the training set of MNIST (60,000 images) into 49,152 and 10,848 images for training and validation, respectively, and evaluate the test accuracy using the testing set (10,000 images). Similarly, we split the training set of CIFAR-10 (50,000 images) into 45,056 and 4,944 images for training and validation, respectively, and evaluate the test accuracy using the testing set (10,000 images). For each task, we tune the mini-batch size, initial learning rate, number of epochs, matrix update interval (for PSGD, K-FAC, and Shampoo), damping τ (for SMW-NG, PSGD, and K-FAC), and number of power iterations (for Shampoo for the eigendecomposition of the Kronecker factors) using the Bayesian optimization under a predefined time budget within the searching spaces described below. The count of combinations explored during the Bayesian optimization is summarized in Table 4. The learning rate is schedule by the cosine annealing decay so that it becomes 0 at the end of training (i.e., the number of epochs affects the decaying speed of learning rate). We apply gradient clipping with the maximum norm of 1. For each task and method, we report the test accuracy and training time of the model checkpoint (in every epoch) achieving the best validation accuracy in Table 2. As a baseline, we also train models with SGD with momentum of 0.9.

B.1. MLP on MNIST

- Mini-batch size : {64,128,256,512,1024,2048}
- Initial learning rate : {3e-1,1e-1,3e-2,1e-2,3e-3,1e-3}
- Number of epochs : {5,10,20}
- Matrix update interval (PSGD, K-FAC, and Shampoo) : {1,3,10,30,100,300}
- Damping τ (SMW-NG, PSGD, and K-FAC) : {1,1e-3,1e-5}
- Number of power iterations (Shampoo) : {10,25,50}

We use a weight decay of 5e-4 and apply no data augmentation.

B.2. ResNet18 and WideResNet on CIFAR-10

We use WideResNet with a depth of 28. We use the existing implementation[5] for defining these architectures. For training WideResNet we adopt dropout(droprate=0.3).

- Mini-batch size : {64,128,256,512,1024,2048}
- Initial learning rate : {3e-1,1e-1,3e-2,1e-2,3e-3,1e-3}
- Number of epochs : {100,200}
- Matrix update interval (for PSGD, K-FAC, and Shampoo) : {3,10,30,100,300}
- Damping τ (for SMW-NG, PSGD, and K-FAC) : {1,1e-3,1e-5}
- Number of power iterations (for Shampoo) : {10,25,50}

We use a weight decay of 5e-4. We apply RandomCrop, RandomHorizontalFlip and Cutout as data augmentation.

B.3. ViT-tiny and MLP-Mixer-base on CIFAR-10

We fine-tune ViT-T/16 and Mixer-B/16 models pretrained on ImageNet-1K.

- Mini-batch size : {64,128,256}
- Initial learning rates : {3e-1,1e-1,3e-2,1e-2,3e-3,1e-3}
- Number of epochs : {10,20}
- Matrix update interval (for PSGD, K-FAC, and Shampoo) : {3,10,30,100,300}
- Damping τ (for PSGD and K-FAC) : {1,1e-3,1e-5}
- Number of power iteration (for Shampoo) : {10,25,50}

We do not use a weight decay. We apply RandomCrop, RandomHorizontalFlip and Cutout as data augmentation.

5. https://github.com/uoguelph-mlrg/Cutout

Table 4: The count of combinations explored during the Bayesian optimization for each setting. Relatively lower counts are due to the lower average throughput (image/s).

Method	MNIST			CIFAR-10			
	MLP (w=128)	MLP (w=512)	MLP (w=2048)	ResNet18	WideResNet	ViT-tiny	MLP-Mixer-base
SGD	344	332	384	60	32	65	45
SMW-NG (w/o RR)	353	408	303	47	25	-	-
PSGD (KF)	392	369	288	50	79	71	61
K-FAC (1mc)	255	317	420	121	79	71	43
Shampoo	150	83	43	29	73	83	52