Learning by solving differential equations

Benoit Dherin* DHERIN@GOOGLE.COM
Michael Munn* MUNN@GOOGLE.COM
Hanna Mazzawi MAZZAWI@GOOGLE.COM
Michael Wunder MWUNDER@GOOGLE.COM
Sourabh Medapati SMEDAPATI@GOOGLE.COM
Javier Gonzalvo XAVIGONZALVO@GOOGLE.COM

Abstract

Modern deep learning algorithms use variations of gradient descent as their main learning methods. Gradient descent can be understood as the simplest Ordinary Differential Equation (ODE) solver; namely, the Euler method applied to the gradient flow differential equation. Since Euler, many ODE solvers have been devised that follow this continuous dynamical system more precisely and more stably. Runge-Kutta (RK) methods provide a family of very powerful explicit and implicit high-order ODE solvers. However, these higher-order solvers have not found wide application in deep learning so far. In this work, we evaluate the performance of higher-order RK solvers when applied in deep learning, study their limitations, and propose ways to overcome these drawbacks. In particular, we explore how to improve their performance by incorporating key ingredients of modern neural network optimizers such as preconditioning, adaptive learning rates, and momentum.

1. Introduction

Optimization in deep learning is notoriously hard. One of the main difficulties is the rough nature of the optimization landscape which causes training instabilities or even convergence failures. Due to the high cost of training large-scale models, particularly modern LLMs, this has sparked a recent interest in optimization techniques which can mitigate these training instabilities (e.g., [39, 60, 63]).

Motivated by the theoretical potential for improved stability offered by higher-order ODE solvers, this work investigates the practical application and adaptation of such Runge-Kutta methods to deep learning, focusing on their performance, limitations, and effective modifications.

The nature of the problem can be traced back to the behaviour of the gradient field for losses coming from neural network optimization. Namely, the loss gradient tends to have extreme variations both in magnitude and direction during the course of training [37, 39, 44]. This makes simple training strategies like Stochastic Gradient Descent (SGD) - where the raw gradient evaluated at a single point over a batch of data is used to update the weight - often inefficient and unstable. The best modern deep-learning optimizers are variations of SGD that rely on essentially two important methods to modify the raw gradient in the SGD update in order to fix these unstable gradient issues. The first method consists of some form of averaging of the gradient over the training run yielding strategies like classical momentum [48, 53, 59]. The averaging helps with canceling out

*. These authors contributed equally to this work

the variations in gradient direction that are unhelpful. The second method, preconditioning, consists in multiplying the gradient by a matrix in order to lower gradient directions of high curvature to tame the gradient field. This has yielded methods like AdaGrad [18], Adadelta [66], RMSProp [61]. The most used generic method Adam [33] (as well as its variants like Nadam [17], AdamW [41], or NadamW [41]) use both schemes in conjunction, yielding optimizers that perform well on a wide range of settings. The most performant optimizers in the AlgoPerf leaderboard [11, 32] at the time of writing are methods using only a special form of preconditioning like Shampoo [2, 55], or methods that leverage some special form of averaging, like the schedule-free AdamW from [12].

Inspired by realizing the learning trajectory as discretizations of a continuous dynamical system (i.e. an ODE), this work introduces a gradient update modification (beyond averaging and preconditioning) to address training instabilities. Raw gradient updates (e.g., $\theta' = \theta - h\nabla L(\theta)$) deviate from the continuous loss minimization path with an $\mathcal{O}(h^2)$ error, causing training oscillations. Drawing from numerical methods of ODEs [28], we propose Runge-Kutta (RK) updates that track the gradient flow trajectory more precisely, leading to more stable optimization. In fact, many common deep learning optimizers can be viewed as first-order ODE solvers. For instance, gradient descent is Euler's method for the gradient flow ODE, and momentum methods can be derived from Hamiltonian systems [20, 23, 34, 45, 54]. Our research explores higher-order RK methods. These higher-order approaches promise more accurate tracking of the true ODE solutions (i.e., the gradient flow ODE which continuously minimizes the loss) compared to their first-order counterparts, aiming for improved stability and performance. Our main contributions are the following:

- We conduct a benchmark of the classical 4th-order RK method and discuss the benefits and limitations of higher-order RK updates in deep learning; see Sec. 2.
- We propose three modifications which aim to address certain limitations of vanilla RK methods: preconditioning, adaptive learning rates, momentum; see Sec. 3.
- We demonstrate experimentally that these modifications do indeed confer benefits when training with RK optimizers; see Figs. 7-13.

2. Benchmarking Runge-Kutta Methods in Deep Learning

In the context of deep learning, we will call a *vanilla RK update*, the gradient update obtained by applying a RK method (defined by the matrix $A = (a_{ij})$ and the vector $b = (b_i)$; see Appendix A for details) to the gradient flow ODE: $\dot{\theta} = -\nabla L(\theta)$, where L is the neural loss. We will denote the RK gradient by $g^*(\theta, h)$ defined by

$$g^*(\theta, h) = \sum_{i=1}^{s} b_i g(\theta_i) \quad \text{where} \quad \theta_i = \theta - h\sum_{j=1}^{s} a_{ij} g(\theta_j), \tag{1}$$

where $g(\theta) = \nabla L(\theta)$ denotes the gradient of the loss $L(\theta)$ and the neighborhood points θ_i where the gradients are evaluated are computed themselves as a solution of an implicit system of equations[1]. Here s denotes the number of "stages" of the RK method and indicates the number of neighboring points on which we "average" the gradient. An RK method of order k must have at least k stages

1. Note that if $a_{ij} = 0$ when $i \geq j$ then θ_i in Eq. (1) can be solved explicitly; in this case the method is called *explicit*, otherwise it is called *implicit*.

(i.e., $k \geq s$); thus giving a lower bound on the method order. The corresponding vanilla RK update with fixed learning rate h is then:

$$\theta' = \theta - hg^*(\theta, h). \tag{2}$$

Observe that the RK gradient $g^*(\theta, h)$ depends on the step-size. This dependence allows the update to be closer to the exact gradient flow $\theta(h)$ that continuously minimizes the loss without any instability. In theory, this is the main advantage of this type of update: they are more stable than raw gradient updates for higher learning rates, allowing for faster and stable convergence.

2.1. Empirical evaluation of vanilla Runge-Kutta

Benchmarking new optimizers is notoriously challenging [11, 32]. This is in part because many of the tricks and techniques needed to achieve SOTA performance are often optimizer-specific and potentially disadvantage a new optimizer. For instance, standard practices like weight decay or batch normalization can conflict with RK's underlying mechanics, if not specifically adapted. We therefore adopt the following strategy: from a collection of highly tuned strong baseline settings, taken from workloads within the init2winit framework [11, 24], we assess vanilla RK's competitiveness by substituting the original optimizer and tuning only RK's learning rate and batch size. This provides a deliberately challenging setup but allows for a naive "worst-case" comparison.

We see that vanilla RK updates can be competitive (even within this adverse setting) with the additional benefit of having only the learning rate train to tune and no accumulators, as shown in Table 1. Note that while vanilla RK excels on simple tasks, its competitiveness drops on more complex workloads. We suspect this is because common training tricks (e.g., batch norm, weight decay, cosine schedule), often necessary for such workloads, are detrimental to RK methods.

Table 1: Comparison of vanilla RK4 (see Appendix B) against strong baselines for various workloads. Accuracy represents mean best test accuracy during training over a fixed number of training steps and standard error over 5 independent trials. See Appendix D.5 for experiment details.

Dataset	**Model**	**Baseline Optimizer**	**Baseline Accuracy (%)**	**RK4 Accuracy (%)**
MNIST	DNN	Adam	$96.64 \pm$ 7e-4	$96.88 \pm$ 6e-4
MNIST	Simple CNN	Momentum	$98.64 \pm$ 2e-4	$98.53 \pm$ 1e-4
Fashion-MNIST	DNN	Adam	$85.6 \pm$ 4e-4	$86.02 \pm$ 4e-4
Fashion-MNIST	Simple CNN	Momentum	$90.12 \pm$ 4e-4	$90.65 \pm$ 2e-3
CIFAR-10	WRN 28-10	Momentum	$97.10 \pm$ 1e-4	$96.78 \pm$ 4e-3
CIFAR-100	WRN 28-10	Momentum	$81.89 \pm$ 2e-3	$78.81 \pm$ 9e-4
ImageNet	ViT	NAdamW	$78.5 \pm$ 2e-3	$61.2 \pm$ 2e-3

2.2. Limitations

Although theoretically RK methods can provide benefits in training stability and simplicity in hyperparameter tuning, there are still practical challenges when naively applied to deep learning workloads. Most notably, these challenges are 1) the increase in compute per step, 2) convergence difficulties in case of gradient field stiffness, and 3) a generalization gap in the large batch setting. Below we describe how these factors can impact the overall efficiency and performance of RK methods.

Wall-clock time v.s. steps for vanilla RK. An order-k RK update requires at least k gradient evaluations per step. While seemingly costly, in fact RK's wall-clock time is comparable to Adam's provided all gradient data fits into device memory simultaneously. This can be seen with full-batch MNIST/Fashion MNIST training, where RK4 and Adam times are nearly identical; see Appendix D.1 Fig. 6. However, RK's wall-clock time sharply increases if gradient data exceeds memory capacity. For instance, when training a ResNet model on CIFAR-10, RK4's time is similar to Adam's for batch size of 1 but roughly doubles for batch size of 8192; see Appendix E.1 Fig. 16. Consequently, performance can become prohibitive for large workloads under current memory constraints. In short, RK updates are competitive in I/O-bound situations (where data fits in memory) but face significant wall-clock time challenges in memory-constrained scenarios.

Gradient flow stiffness. Runge-Kutta methods struggle with *stiff* ODEs $\dot{\theta} = f(\theta)$; i.e., where the vector field $f(\theta)$ exhibits large variations in reaction to small changes in θ, see [26]. In neural network optimization, such stiffness can arise from phenomena influencing the vector field at different scales such as interactions between smooth large-scale geometry and bumpy smaller-scale geometry of the loss surface [37]. This issue has been previously addressed in Physics-Informed Neural Networks (PINNs) using implicit first-order RK methods where stiffness has been shown to correlate with large Hessian eigenvalue gaps [38, 62]. While implicit RK methods generally manage stiffness better than explicit ones [26], the computational cost of solving their required non-linear systems given by (14) is often prohibitive for higher-order methods in neural networks. The following sections investigate modifications to the vanilla RK4 method which aim to mitigate this stiffness and help lessen the generalization gap we see in the large batch setting; cf. Fig. 1.

Generalization gap in large batch settings. In large batch settings, Runge-Kutta (RK) methods are stable because they closely follow the gradient flow. This, however, also prevents them from benefiting from the beneficial flatness bias induced by first-order discretization errors [5, 23, 56]. The absence of this implicit regularization leads to a generalization gap causing RK methods to underperform first-order methods like Adam [9], even on simple datasets such as MNIST and Fashion-MNIST in the large batch setting; see Fig. 1 (right column). On the other hand, for small batch settings, the implicit regularization due to the stochastic noise helps RK4 bridge the generalization gap existing in the large batch setting; see Fig. 1 (left column).

In the next section, we propose modifications which aim to combat ODE stiffness and bridge the generalization gap in the large batch setting observed in Fig. 1. Namely, we explore 1) modifying the gradient field via preconditioning, 2) adaptive learning rates, and 3) adding momentum.

3. Modifications to vanilla Runge-Kutta optimization

In this section we explore how three modern training techniques—preconditioning, adaptive learning rates, and momentum—can be integrated with vanilla RK methods to mitigate ODE stiffness and improve performance, allowing us to bridge the generalization gap observed in Fig. 1. For our experiments, we focus on "pure" MLP-based workloads, specifically MNIST and Fashion MNIST, since competitive performance is achievable without complex, optimizer-specific tricks.

3.1. Preconditioning Runge-Kutta by modifying the ODE

As discussed in Section 2.2, stiffness presents a significant challenge when applying RK methods to the gradient flow ODE of a neural loss, as it can cause the gradient field to change dramatically in response to even minor parameter variations. One way to mitigate this problematic behavior is through preconditioning; e.g., [1, 15, 18, 25, 43], see also [64] and [6]. One way to view preconditioning is that it offers the flexibility to transform the gradient flow ODE into one which alleviates this stiffness problem. That is, at each step k of the optimization process, we modify our differential equation $\dot{\theta} = f_k(\theta)$ and solve it with an RK method with step h and initial condition θ_k yielding a different RK update than the original RK update for step k. The only requirement is that the neural loss L decreases along the exact solutions of $\dot{\theta} = f_k(\theta)$ and that critical points of the vector field $f_k(\theta)$ remain the same as the critical points of the loss gradient. We can test this by computing the derivative of the loss w.r.t. time along the solutions of $\dot{\theta} = f_k(\theta)$ remains negative.[2]

Using this trick, we modify the gradient flow ODE to precondition the gradient. The following lemma outlines this step-wise modification using a matrix (potentially dependent on the step and parameters), which generalizes prior work [10, 19].

Lemma 1 (Preconditioning lemma) *Consider the preconditioned gradient flow $\dot{\theta}(t) = -A(\theta(t))g(\theta(t))$, where $g(\theta) = \nabla L(\theta)$ is the loss gradient and $A(\theta)$ is a positive definite symmetric matrix possibly depending on the parameter θ. Then, the loss L decreases on the exact solutions $\theta(t)$ of the preconditioned gradient flow equation at the following rate*

$$\frac{dL(\theta(t))}{dt} = -\|g(\theta(t))\|_A^2, \tag{3}$$

where $\|v\|_A^2 = v^T A v$ is the norm of v in the metric given by A.

Proof We obtain immediately that $\frac{dL(\theta(t))}{dt} = \nabla L(\theta(t))^T \dot{\theta}(t) = -g(\theta(t))^T A(\theta(t)) g(\theta(t))$. The positivity of $\|g(\theta(t))\|_A^2$ for all vectors is guaranteed by the fact that $A(\theta)$ is a Riemannian metric (i.e., $A(\theta)$ is symmetric and positive definite for all θ). ■

AdaGrad-like preconditioning for RK updates. AdaGrad [18] is one of the earliest forms of adaptive preconditioning and works by adapting the learning rate for each parameter based on the history of its gradients. More specifically, AdaGrad uses a matrix $G_n = \sum_{l=1}^n g(\theta_l)g(\theta_l)^T$ to accumulate past gradient outer products, capturing information about the loss-surface metric through the learning trajectory. Viewed through the lens of Lemma 1, for step n, Adagrad takes

$$A_n(\theta) = (\epsilon + \operatorname{diag} G_n)^{-1/2}, \tag{4}$$

where here ϵ is a small constant introduced only to improve numerical stability.

The insight that preconditioning smooths local geometry and reduces ODE stiffness suggests a natural choice for A_n. Specifically, a loss surface given by $L(\theta)$ possesses a natural geometry [5, 49], with its local distance metric characterized by the matrix

$$\mathcal{G}(\theta) = \mathbb{I} + g(\theta)g(\theta)^T, \quad \text{where} \quad g(\theta) = \nabla L(\theta). \tag{5}$$

2. In mathematical terms, this means that we can choose any differential equation at step k as long as the loss function remains a Lyapunov function for that differential equation [4].

Thus, preconditioning by $\mathcal{G}^{-1}(\theta)$ naturally scales the gradient field, thereby smoothing the local geometry. However, computing $\mathcal{G}^{-1}(\theta)$ is often impractical due to the matrix inversion. Inspired by AdaGrad, we instead use a diagonal approximation of $g(\theta)g(\theta)^T$ and (similar to AdaGrad) average over the trajectory which leads us to our *modified AdaGrad* preconditioner given by

$$A'_n(\theta) = (1 + \operatorname{diag} G_n)^{-1/2}. \tag{6}$$

This yields a preconditioner similar to AdaGrad's but which performs better than AdaGrad in our experiments and succeeds in decreasing the generalization gap with Adam in the large batch regime; as shown in Fig. 7. See Appendix C for more details around this preconditioning choice.

3.2. Making use of adaptive learning rates

Adaptive learning rates provide a promising tool for addressing the challenge of stiffness of the ODE. By automatically decreasing the learning rate in stiff regions and increasing the learning rate in more tame regions, we are able to more fully benefit from the precision offered by higher-order RK methods. The appropriate size for the learning rate is determined by measuring how the gradient varies in a neighborhood of a given point which, following [14], can be computed via the derivative of the gradient along the gradient flow:

$$\frac{dg(\theta)}{dt} = \nabla g(\theta)\dot{\theta} = -H(\theta)g(\theta), \tag{7}$$

where g and H denote the gradient and the Hessian of the loss, respectively.

Equation (7) implies that the gradient $g(\theta)$ along the gradient flow path changes slowly when the term $\|H(\theta)g(\theta)\|$ is small, and changes rapidly when this term is large. Therefore, an adaptive learning rate should be larger precisely when $\|H(\theta)g(\theta)\|$ is small (reflecting slow gradient changes) and smaller when it is large (reflecting rapid changes). In short, the learning rate should be inversely proportional to $\|H(\theta)g(\theta)\|$. To mitigate instabilities that may arise due to excessively large learning rates, we propose a rescaled version of the adaptive learning rate DAL-p in [51]. That is, we set

$$h_{\text{DALR}}(\theta) := \frac{c}{1 + \frac{c}{2}\left(\frac{\|H(\theta)g(\theta)\|}{\|g(\theta)\|}\right)^p}. \tag{8}$$

The new learning rate (8) is now capped by the (tunable) value c and substantially improves upon vanilla RK4, even beating tuned Adam on MNIST and Fashion-MNIST; as shown in Fig. 10.

3.3. Incorporating momentum with Runge-Kutta updates

Momentum [59] is a core component of many state-of-the-art optimizers like Adam [33] and its variants (e.g., Nadam [17], AdamW [41]). Traditionally, momentum based optimizers employ an exponential moving average (EMA) of past raw gradients, weighted by a hyperparameter β. We adapt this principle by applying the EMA to the RK gradients instead of the raw gradients, yielding the following RK momentum scheme

$$m_{n+1} = \beta m_n + g^*(\theta_n, h) \tag{9}$$
$$\theta_{n+1} = \theta_n - h m_{n+1}, \tag{10}$$

where the lookahead step uses a Runge–Kutta estimate of the gradient $g^*(\theta_n, h)$ in place of the standard Nesterov approximation [58], and where $m_0 = 0$. Accumulating these more precise gradients is beneficial and helps bridge the gap between vanilla RK and Adam, as shown in Fig. 13.

References

[1] Shun-ichi Amari. Natural gradient works efficiently in learning. *Neural computation*, 10(2):251–276, 1998. doi: 10.1162/089976698300017746. URL https://direct.mit.edu/neco/article/10/2/251/6143/Natural-Gradient-Works-Efficiently-in-Learning.

[2] Rohan Anil, Vineet Gupta, Tomer Koren, Kevin Regan, and Yoram Singer. Scalable second order optimization for deep learning. *arXiv preprint arXiv:2002.09018*, 2020. URL https://arxiv.org/abs/2002.09018.

[3] Imen Ayadi and Gabriel Turinici. Stochastic runge-kutta methods and adaptive sgd-g2 stochastic gradient descent. In *2020 25th International Conference on Pattern Recognition (ICPR)*, 2021.

[4] Andrea Bacciotti and Lionel Rosier. *Liapunov Functions and Stability in Control Theory*. Lecture Notes in Control and Information Sciences. Springer, London, 1 edition, 2001. ISBN 978-1-84628-817-3.

[5] David G.T. Barrett and Benoit Dherin. Implicit gradient regularization. In *ICLR*, 2021.

[6] Jeremy Bernstein and Laker Newhouse. Modular duality in deep learning. *arXiv preprint arXiv:2410.21265*, 2024.

[7] Michael Betancourt, Michael I Jordan, and Ashia C Wilson. On symplectic optimization. *arXiv preprint arXiv:1802.03653*, 2018. URL https://arxiv.org/abs/1802.03653.

[8] A. A. Brown and M. C. Bartholomew-Biggs. Some effective methods for unconstrained optimization based on the solution of systems of ordinary differential equations. *J. Optim. Theory Appl.*, 62(2):211–224, August 1989.

[9] Matias D Cattaneo, Jason M Klusowski, and Boris Shigida. On the implicit bias of adam. *arXiv:2309.00079*, 2023.

[10] J. Cortés. Finite-time convergent gradient flows with applications to network consensus. *Automatica*, 42(11):1993–2000, 2006.

[11] George E. Dahl, Frank Schneider, Peter Mattson, et al. Benchmarking neural network training algorithms. *arXiv preprint arXiv:2306.07179*, 2023. URL https://arxiv.org/abs/2306.07179.

[12] Aaron Defazio, Xingyu Yang, Harsh Mehta, Konstantin Mishchenko, Ahmed Khaled, and Ashok Cutkosky. The road less scheduled. *arXiv preprint arXiv:2405.15682*, 2024. URL https://arxiv.org/abs/2405.15682.

[13] Benoit Dherin and Mihaela Rosca. Corridor geometry in gradient-based optimization, 2024. URL https://arxiv.org/abs/2402.08818.

[14] Benoit Dherin, Michael Munn, Mihaela Rosca, and David Barrett. Why neural networks find simple solutions: The many regularizers of geometric complexity. In *NeurIPS*, 2022.

[15] Kingma Diederik. Adam: A method for stochastic optimization. *(No Title)*, 2014.

[16] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

[17] Timothy Dozat. Incorporating nesterov momentum into adam. In *ICLR Workshop*, 2016.

[18] 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.

[19] Carmel Fiscko, Aayushya Agarwal, Yihan Ruan, Soummya Kar, Larry Pileggi, and Bruno Sinopoli. Towards hyperparameter-agnostic dnn training via dynamical system insights. *arXiv preprint arXiv:2310.13901*, 2023.

[20] Guilherme França, Jeremias Sulam, Daniel Robinson, and René Vidal. Conformal symplectic and relativistic optimization. In *NeurIPS*, 2020.

[21] Guilherme França, Daniel P Robinson, and René Vidal. Admm and accelerated admm as continuous dynamical systems. In *International Conference on Machine Learning*, pages 1554–1562. PMLR, 2018. URL http://proceedings.mlr.press/v80/franca18a/franca18a.pdf.

[22] Guilherme França, Daniel P Robinson, and René Vidal. Gradient flows and proximal splitting methods: A unified view on accelerated and stochastic optimization. *Physical Review E*, 103(5), 2021. URL https://link.aps.org/doi/10.1103/PhysRevE.103.053304.

[23] Avrajit Ghosh, He Lyu, Xitong Zhang, and Rongrong Wang. Implicit regularization in heavy-ball momentum accelerated stochastic gradient descent. *ICLR*, 2023.

[24] Justin M. Gilmer, George E. Dahl, Zachary Nado, Priya Kasimbeg, and Sourabh Medapati. init2winit: a jax codebase for initialization, optimization, and tuning research. *github*, 2023. URL http://github.com/google/init2winit.

[25] Vineet Gupta, Tomer Koren, and Yoram Singer. Shampoo: Preconditioned stochastic tensor optimization. In *International Conference on Machine Learning*, pages 1842–1850. PMLR, 2018.

[26] Ernst Hairer and Gerhard Wanner. *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems*, volume 14 of *Springer Series in Computational Mathematics*. Springer, Berlin, Heidelberg, 2 edition, 1996. ISBN 978-3540604525.

[27] Ernst Hairer, Syvert P Nørsett, and Gerhard Wanner. *Solving Ordinary Differential Equations I: Nonstiff Problems*, volume 8 of *Springer Series in Computational Mathematics*. Springer, 2 edition, 1993. URL https://link.springer.com/book/10.1007/978-3-662-12607-3.

[28] Ernst Hairer, Marlis Hochbruck, Arieh Iserles, and Christian Lubich. Geometric numerical integration. *Oberwolfach Reports*, 3(1):805–882, 2006.

[29] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016.

[30] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016.

[31] Matteo Hessel, David Budden, Fabio Viola, Mihaela Rosca, Eren Sezener, and Tom Hennigan. Optax: composable gradient transformation and optimisation, in jax!, 2020. URL http://github.com/deepmind/optax.

[32] Priya Kasimbeg, Frank Schneider, Runa Eschenhagen, Juhan Bae, Chandramouli Shama Sastry, Mark Saroufim, BOYUAN FENG, Less Wright, Edward Z. Yang, Zachary Nado, Sourabh Medapati, Philipp Hennig, Michael Rabbat, and George E. Dahl. Accelerating neural network training: An analysis of the algoperf competition. In *ICLR*, 2025.

[33] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *ICLR*, 2015.

[34] Nikola B Kovachki and Andrew M Stuart. Continuous time analysis of momentum methods. *Journal of Machine Learning Research*, 22(17):1–40, 2021. URL https://jmlr.org/papers/v22/19-466.html.

[35] Alex Krizhevsky. Learning multiple layers of features from tiny images. *https://www.cs.toronto.edu/ kriz/learning-features-2009-TR.pdf*, 2009.

[36] John M. Lee. *Introduction to Smooth Manifolds*, volume 218 of *Graduate Texts in Mathematics*. Springer, 2nd edition, 2012.

[37] Hao Li, Zheng Xu, Gavin Taylor, Christoph Studer, and Tom Goldstein. Visualizing the loss landscape of neural nets. In *NeurIPS*, 2018.

[38] Ye Li, Song-Can Chen, and Sheng-Jun Huang. Implicit stochastic gradient descent for training physics-informed neural networks. *arXiv preprint arXiv:2303.01767*, 2023. URL https://arxiv.org/abs/2303.01767.

[39] Liyuan Liu, Xiaodong Liu, Jianfeng Gao, Weizhu Chen, and Jiawei Han. Understanding the difficulty of training transformers. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP 2020)*, April 2020.

[40] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

[41] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *ICLR*, 2019.

[42] James Lucas, Shengyang Sun, Richard Zemel, and Roger Grosse. Aggregated momentum: Stability through passive damping. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=Syxt5oC5YQ.

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

[44] Sam McCandlish, Jared Kaplan, Dario Amodei, and OpenAI Dota Team. An empirical model of large-batch training. *arXiv preprint arXiv:1812.06162*, 2018.

[45] Michael Muehlebach and Michael I Jordan. A dynamical systems perspective on nesterov acceleration. In *International Conference on Machine Learning*, pages 4656–4662. PMLR, 2019. URL https://arxiv.org/abs/1905.07436.

[46] Antonia Orvieto, Simon Lacoste-Julien, and Nicolas Loizou. Dynamics of sgd with stochastic polyak stepsizes: Truly adaptive variants and convergence to exact solution. In *NeurIPS*, 2022.

[47] Antonio Orvieto and Lin Xiao. An Adaptive Stochastic Gradient Method with Non-negative Gauss-Newton Stepsizes. *arXiv preprint arXiv:2407.04358*, 2024.

[48] Boris T. Polyak. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1–17, 1964.

[49] Alison Pouplin, Hrittik Roy, Sidak Pal Singh, and Georgios Arvanitidis. On the curvature of the loss landscape. *arXiv preprint arXiv:2307.04719*, 2023.

[50] Chongli Qin, Yan Wu, Jost Tobias Springenberg, Andy Brock, Jeff Donahue, Timothy Lillicrap, and Pushmeet Kohli. Training generative adversarial networks by solving ordinary differential equations. In *Advances in Neural Information Processing Systems*, 2020. URL https://proceedings.neurips.cc/paper/2020/hash/3c8f9a173f749710d6377d3150cf90da-Abstract.html.

[51] Mihaela Rosca, Yan Wu, Benoit Dherin, and David G.T. Barrett. Discretization drift in two-player games. In *ICML*, 2021.

[52] Mihaela Rosca, Yan Wu, Chongli Qin, and Benoit Dherin. On a continuous time model of gradient descent dynamics and instability in deep learning. In *TMLR*, 2023.

[53] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.

[54] Bin Shi, Simon S Du, Weijie J Su, and Michael I Jordan. Acceleration via symplectic discretization of high-resolution differential equations. In *Advances in Neural Information Processing Systems*, pages 5744–5752, 2019. URL https://papers.nips.cc/paper/8811-acceleration-via-symplectic-discretization-of-high-resolution-different

[55] Hao-Jun Michael Shi, Tsung-Hsien Lee, Shintaro Iwasaki, Jose Gallego-Posada, Zhijing Li, Kaushik Rangadurai, Dheevatsa Mudigere, and Michael Rabbat. A distributed data-parallel pytorch implementation of the distributed shampoo optimizer for training neural networks at-scale. *arXiv preprint arXiv:2309.06497*, 2023. URL https://arxiv.org/abs/2309.06497.

[56] Samuel L Smith, Benoit Dherin, David G.T. Barrett, and Soham De. On the origin of implicit regularization in stochastic gradient descent. In *ICLR*, 2021.

[57] Dan Su, Qihai Jiang, Enhong Liu, and Mei Liu. Improving optimizers by runge-kutta method: A case study of sgd and adam. In *2024 12th International Conference on Intelligent Control and Information Processing (ICICIP)*, 2024.

[58] Weijie Su, Stephen Boyd, and Emmanuel Candès. A differential equation for modeling nesterov's accelerated gradient method: theory and insights. *Journal of Machine Learning Research*, 17:1–43, 2016. URL https://jmlr.org/papers/v17/15-084.html.

[59] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th International Conference on Machine Learning*, volume 28, pages 1139–1147. PMLR, 2013.

[60] Sho Takase, Shun Kiyono, Sosuke Kobayashi, and Jun Suzuki. Spike no more: Stabilizing the pre-training of large language models, 2025. URL https://openreview.net/forum?id=G84F1h2IiD.

[61] Tijmen Tieleman and Geoffrey Hinton. Rmsprop: Divide the gradient by a running average of its recent magnitude. COURSERA Neural Networks for Machine Learning, 2012.

[62] Sifan Wang, Yujun Teng, and Paris Perdikaris. Understanding and mitigating gradient flow pathologies in physics-informed neural networks. *SIAM Journal on Scientific Computing*, 43(5):A3055–A3081, 2021. doi: 10.1137/20M1318043. URL https://epubs.siam.org/doi/abs/10.1137/20M1318043.

[63] Mitchell Wortsman, Peter J Liu, Lechao Xiao, Katie E Everett, Alexander A Alemi, Ben Adlam, John D Co-Reyes, Izzeddin Gur, Abhishek Kumar, Roman Novak, Jeffrey Pennington, Jascha Sohl-Dickstein, Kelvin Xu, Jaehoon Lee, Justin Gilmer, and Simon Kornblith. Small-scale proxies for large-scale transformer training instabilities. In *ICLR*, 2024.

[64] Shuo Xie, Tianhao Wang, Sashank Reddi, Sanjiv Kumar, and Zhiyuan Li. Structured preconditioners in adaptive optimization: A unified analysis. *arXiv preprint arXiv:2503.10537*, 2025.

[65] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. *arXiv preprint arXiv:1605.07146*, 2016.

[66] Matthew D. Zeiler. Adadelta: An adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

[67] Jingzhao Zhang, Aryan Mokhtari, Suvrit Sra, and Ali Jadbabaie. Direct runge-kutta discretization achieves acceleration. In *Advances in Neural Information Processing Systems*, 2018. URL https://papers.nips.cc/paper/7646-direct-runge-kutta-discretization-achieves-acceleration.

[68] Michael R Zhang, James Lucas, Geoffrey Hinton, and Jimmy Ba. Lookahead optimizer: k steps forward, 1 step back. In *Advances in neural information processing systems*, pages 9591–9601, 2019.

Appendix A. Background on RK methods

RK methods are a family of iterative methods that numerically approximate solutions to ODEs by calculating a weighted average of gradient estimates at several points within a single step (similarly to the lookahead optimizer from [68] or the aggregated momentum from [42]). These methods have an order associated to them, and the higher the order the more closely the numerical solution follows the exact solution. Neural network training can be seen as numerically solving the Gradient Flow (GF) differential equation. Since the exact solutions of GF are always stable, we expect that higher-order methods following these solutions more closely will benefit from increased stability in training loss. We explain this now in more detail.

Consider a general ODE of the form $\dot{\theta} = f(\theta)$, where $f : \mathbb{R}^k \to \mathbb{R}^k$ is a function called the ODE *vector field*. We use the notation $\dot{\theta}(t)$ to denote the derivative of the curve $\theta(t)$, which geometrically is the tangent vector to the curve at time t. A curve $\theta(t)$ is a solution of the differential equation $\dot{\theta} = f(\theta)$ if its tangent vector at t is exactly given by the vector field $f(\theta)$ evaluated at $\theta(t)$.

Gradient Flow. In the context of deep learning, the vector field is the negative gradient field: $f(\theta) := -\nabla L(\theta)$, where $L(\theta)$ is the loss, yielding the *gradient flow* ODE: $\dot{\theta} = -\nabla L(\theta)$. The exact solutions $\theta(t)$ of this differential equation are the continuous paths of steepest descent, along which the loss decreases continuously and stably at the rate prescribed by the gradient norm; namely, along a gradient-flow solution $\theta(t)$, the loss derivative w.r.t. time is negative:

$$\frac{dL(\theta(t))}{dt} = \nabla L(\theta(t))^T \dot{\theta}(t) = -\|\nabla L(\theta(t))\|^2 \leq 0$$

Following the solutions of this ODE closely, through numerical iterations, then ensures decreasing the loss in a way that is more stable (i.e., less loss oscillations) as we are closer to the gradient flow.

Order of an ODE solver. The simplest possible ODE solver is the Euler method, which in deep learning is called gradient descent:

$$\theta' = \theta + hf(\theta),$$

which approximates the exact solution $\theta(t)$ of the differential equation $\dot{\theta} = f(\theta)$ after a time $t = h$ starting from the point θ. Gradient descent is a *first-order* solver, since its error with respect to the solution of gradient flow after a step of size h is of order $\mathcal{O}(h^2)$. It is easy to see this by comparing the Taylor expansion of the exact solution of the ODE $\dot{\theta} = f(\theta)$ with the numerical method. The Taylor expansion of the exact solution $\theta(t)$ of $\dot{\theta} = f(\theta)$ is easy to compute:

$$\begin{aligned} \theta(h) &= \theta + \dot{\theta}(0)h + \frac{1}{2!}\ddot{\theta}(0)h^2 + \mathcal{O}(h^3) && (11) \\ &= \theta + hf(\theta) + \frac{h^2}{2} f'(\theta)f(\theta) + \mathcal{O}(h^3), && (12) \end{aligned}$$

where we obtained the second derivative of $\theta(t)$ by differentiating the ODE $\dot{\theta} = f(\theta)$ on both sides w.r.t. to time: $\ddot{\theta} = f'(\theta)\dot{\theta} = f'(\theta)f(\theta)$. From the exact solution expansion in (12), we see immediately that the Euler method is just a first order expansion, since it coincides with the first-order term in the learning rate only yielding an error of size $\mathcal{O}(h^2)$. More generally, we say that

Definition 2 *An iterative ODE solver as above is of order k if the error after one step is of order $\mathcal{O}(h^{k+1})$. In other words, for a method of order k, we have that*

$$\|\theta' - \theta(h)\| = \mathcal{O}(h^{k+1}).$$

where θ' is the method iterator after one step of size h from θ and $\theta(h)$ is the exact solution of the ODE starting at θ after a time h.

Runge-Kutta methods for solving ODEs. The Euler method is the simplest ODE solver from a vast family of ODE solvers. RK methods [26–28] generalize the Euler method in that each step is computed from a weighted average of gradients evaluated in a neighborhood of the previous iterate:

$$\theta' = \theta + h \sum_{i=1}^{s} b_i f(\theta_i(h)), \tag{13}$$

where the b_i's sum to one and the $\theta_i(h)$'s are points in a neighborhood of the previous iterate θ with $\theta_i(h) \to \theta$ as $h \to 0$, recovering the gradient flow for infinitesimally small learning rates. The neighborhood points $\theta_i(h)$ where the gradients are evaluated are computed themselves as a solution of an implicit system of equations[3] given by

$$\theta_i = \theta + h \sum_{j=1}^{s} a_{ij} f(\theta_j). \tag{14}$$

The main idea behind RK methods is that one can find special vectors $b = (b_i)$ and matrices $A = (a_{ij})$ such that the resulting ODE solver is of higher order (i.e., with lower error). This is done by expanding the solution of the differential equation as well as the RK method into a Taylor series and choosing the coefficients a_{ij} and b_i such that both series coincide up to the desired order. For instance, the Taylor series up to 2nd order of the RK method in (13) is (see Appendix B Eqs. (19)-(22) for details):

$$\theta' = \theta + h \left(\sum_{i=1}^{s} b_i \right) f(\theta) + h^2 \left(\sum_{ij=1}^{s} b_i a_{ij} \right) f'(\theta) f(\theta) + \mathcal{O}(h^3). \tag{15}$$

Comparing this with the exact solution Taylor expansion in (12), we obtain the conditions

$$\sum_{i=1}^{s} b_i = 1 \quad \text{and} \quad \sum_{ij=1}^{s} b_i a_{ij} = \frac{1}{2} \tag{16}$$

which ensure the method is of order 1 and 2, respectively. Finding RK weights that satisfy these and higher-order conditions is a difficult problem. We provide in Appendix B an illustration of the procedure by computing the weights for all possible second-order RK methods, and we give concrete coefficients for RK methods of order 3 and 4 as well. The number of gradient evaluations used in the RK method is called the number of *stages*. One key result is that for a method to be of order s we need at least (and possibly more) s stages. *This means that improving the method order requires increasing the number of gradient evaluations at each step [27].*

3. Note that if $a_{ij} = 0$ if $i \geq j$ then Eq. (14) can be solved explicitly; in this case the method is called *explicit*, otherwise it is called *implicit*.

Appendix B. Runge-Kutta updates

This section presents the formulas for Runge-Kutta (RK) updates up to the fourth order (RK2, RK3, and RK4). These methods provide approximations to the exact solutions of a differential equation $\dot{\theta} = f(\theta)$, where f is the differential-equation vector-field. For explicit second-order methods, it's possible to parameterize the entire family of these methods. We include this well-known computation [27] to illustrate the general approach for finding RK coefficients. In the main paper, we benchmarked RK4, which is the classical 4th order method, and it has an error of size $\mathcal{O}(h^5)$.

Recall that a RK method is given by a matrix $A = (a_{ij})$ and a vector $b = (b_i)$. The corresponding RK update is of the form

$$\theta' = \theta + h\sum_{i=1}^{s} b_i f(\theta_i), \tag{17}$$

where the points θ_i are the solutions of the following system of equations

$$\theta_i = \theta + h\sum_{j=1}^{s} a_{ij} f(\theta_j). \tag{18}$$

The idea is to find A and b such that the Taylor expansion of the RK update in Eq. (17) coincides up to order $k+1$ with the Taylor series of the exact solution of the ODE, yielding a RK method of order k. The first step is to compute the Taylor expansion of the numerical method. This is easy for the first two orders, but rapidly becomes extremely complex and requires the introduction of sophisticated mathematics (see [27]). Let us compute this Taylor expansion here up to order 2, then we will illustrate the full process in the case of explicit second order methods:

$$\begin{aligned}
\theta' &= \theta + h\sum_{i=1}^{s} b_i f(\theta_i) && (19)\\
&= \theta + h\sum_{i=1}^{s} b_i f\Big(\theta + h\sum_{j=1}^{s} a_{ij} f(\theta_j)\Big) && (20)\\
&= \theta + h\sum_{i=1}^{s} b_i \left(f(\theta) + f'(\theta)\Big(h\sum_{j=1}^{s} a_{ij} f(\theta + \mathcal{O}(h))\Big) + \mathcal{O}(h^2)\right) && (21)\\
&= \theta + h\left(\sum_{i=1}^{s} b_i\right) f(\theta) + h^2\left(\sum_{ij=1}^{s} b_i a_{ij}\right) f'(\theta)f(\theta) + \mathcal{O}(h^3). && (22)
\end{aligned}$$

Therefore, the condition

$$\sum_{i=1}^{s} b_i = 1 \tag{23}$$

ensures that the method is at least a first order method, while the condition

$$\sum_{ij=1}^{s} b_i a_{ij} = \frac{1}{2} \tag{24}$$

ensures the method is at least of order 2.

The next paragraph applies this argument to explicit second order methods.

General second-order methods. For explicit second-order methods, we can compute all their possible weight, while also illustrating the general principle how to compute the RK weights $a = (a_{ij})$ and $b = (b_i)$. These methods have only two gradient evaluations (i.e. they have two stages), and a step of the method will have an error of order $\mathcal{O}(h^3)$. Since the method has two stages the vector b has two components only, and the matrix a is a lower-diagonal 2×2 matrix. This means that we can parameterize the probability vector b with a single number $\alpha \in [0,1]$, namely $b_1 = 1-\alpha$ and $b_2 = \alpha$. On the other hand, the lower-triangular 2×2 matrix a has a single non-zero entry $a_{21} = \beta$. Therefore the two points where we will evaluate the gradients will be $\theta_1 = \theta$ and $\theta_2 = \theta + h\beta f(\theta)$. This yields the gradient update rule:

$$\theta' = \theta + h\Bigg((1-\alpha)f(\theta) + \alpha f(\theta + h\beta f(\theta))\Bigg), \tag{25}$$

which parameterizes all the two-stages explicit RK methods. Now, the goal is to find α and β so that the error between one step of the method above and the exact solutions of $\dot\theta = f(\theta)$ is of order $\mathcal{O}(h^3)$. To figure that out, we can compute the Taylor expansion of the exact solution up to order $\mathcal{O}(h^3)$ as well as that of the update rule above and adjust the α and β for the two series to coincide up to order $\mathcal{O}(h^3)$. Let us do that. First of all, the Taylor's expansion of the solution $\theta(h)$ of the ODE $\dot\theta(t) = f(\theta(t))$ starting at $\theta(0) = \theta$ is given at the first orders by

$$\begin{aligned} \theta(h) &= \theta + \dot\theta(0)h + \frac{1}{2!}\ddot\theta(0)h^2 + \mathcal{O}(h^3) && (26)\\ &= \theta + hf(\theta) + \frac{h^2}{2}f'(\theta)f(\theta) + \mathcal{O}(h^3), && (27)\end{aligned}$$

where we obtained the second derivative of $\theta(t)$ by differentiating the ODE $\dot\theta = f(\theta)$ on both sides w.r.t. to time: $\ddot\theta = f'(\theta)\dot\theta = f'(\theta)f(\theta)$. On the other hand for the two-stage RK update rule (25) we have the following expansion in the learning rate:

$$\theta' = \theta + hf(\theta) + h^2\alpha\beta f'(\theta)f(\theta) + \mathcal{O}(h^3).$$

So, as long as $\alpha\beta = \frac{1}{2}$, that is $\beta = \frac{1}{2\alpha}$, the method will be of second order, yielding all possible two-stage second-order RK methods. They are parameterized by $\alpha \in (0,1]$ with updates:

$$\theta' = \theta + h\Bigg((1-\alpha)f(\theta) + \alpha f\big(\theta + \frac{h}{2\alpha}f(\theta)\big)\Bigg), \tag{28}$$

A popular choice is to take $\alpha = 1/2$ and $\beta = 1$, leading to the *improved Euler method* (also know as the Heun method) which is an order of magnitude more precise than the Euler method (the error is $\mathcal{O}(h^3)$ compared to $\mathcal{O}(h^2)$ for the Euler method) but it necessitates two gradient evaluations.

Below, we list popular 2nd, 3rd, and 4th order RK methods for reference.

RK2: Heun 2nd order method. As we saw, there all the second order RK methods are parameterized by a single coefficient $\alpha \in (0,1]$. A popular choice is $\alpha = 1/2$ leading to the Heun method or improved Euler method:

$$\begin{aligned} \theta_1 &= \theta && (29)\\ \theta_2 &= \theta + hf(\theta_1) && (30)\\ \theta' &= \theta + \frac{h}{2}\Big(f(\theta_1) + f(\theta_2)\Big) && (31)\end{aligned}$$

RK3: Kutta 3^{rd} order method. There are many more ways to produce 3rd order methods than second order ones. The computation idea is the same but it becomes much more intricate. For instance, here is a popular 3rd order method:

$$\begin{align}
\theta_1 &= \theta \tag{32}\\
\theta_2 &= \theta + \frac{h}{2} f(\theta_1) \tag{33}\\
\theta_3 &= \theta - h f(\theta_1) + 2h f(\theta_2) \tag{34}\\
\theta' &= \theta + \frac{h}{6}\Big(f(\theta_1) + 4 f(\theta_2) + f(\theta_3) \Big) \tag{35}
\end{align}$$

RK4: Classical 4^{th} order method. Similarly, there are many more ways to produce 4^{th} order methods than second order ones. Here is a very popular 4^{th} order method:

$$\begin{align}
\theta_1 &= \theta \tag{36}\\
\theta_2 &= \theta + \frac{h}{2} f(\theta_1) \tag{37}\\
\theta_3 &= \theta + \frac{h}{2} f(\theta_2) \tag{38}\\
\theta_4 &= \theta + h f(\theta_3) \tag{39}\\
\theta' &= \theta + h\frac{1}{6}\Big(f(\theta_1) + 2 f(\theta_2) + 2 f(\theta_3) + f(\theta_4) \Big) \tag{40}
\end{align}$$

Appendix C. Geometry of preconditioning

We discuss here the mathematics and intuition underlying the preconditioning of RK updates, giving some motivation for our particular preconditioning choice in Section 3.1.

Vectors and co-vectors. Let V be a vector space. A vector in v is usually represented in coordinates by a column vector. On the other hand, elements of the dual space V^*, which are linear maps from V to $\mathbb{R}$, are represented by row vectors and called *co-vectors*. The application of a co-vector $\mu \in V^*$ to a vector v yields the number $\mu \cdot v$, where $\cdot$ is the matrix product of the row vector μ by the column vector v.

In machine learning, the loss gradient $\nabla L(\theta)$ is typically a co-vector (i.e., a row vector). This is the linear map that tells us how much the loss changes if we move from the parameter θ in the direction v, namely $L(\theta + v) \simeq L(\theta) + \nabla L(\theta) \cdot v$.

On the other hand, the velocity $\dot{\theta}(t)$ of a curve $\theta(t)$ in parameter space, is a *vector* (represented by a column vector) since by definition of the derivative we have

$$\dot{\theta}(t) = \lim_{\epsilon \to 0} \frac{\theta(t+\epsilon) - \theta(t)}{\epsilon}. \tag{41}$$

This is important for ODE's because on both sides of a differential equation

$$\dot{\theta}(t) = f(\theta(t)) \tag{42}$$

we need to be vectors (i.e. column vectors).

The gradient flow ODE. The standard gradient flow equation $\dot{\theta}(t) = \nabla L(\theta(t))$ suffers from the problem of equating a vector with a co-vector: the gradient flow written in this form is not *invariant under change of coordinates*. This means that the gradient flow equation does not assume the same form after a change of coordinates because vectors and co-vectors transform differently. This issue has already been noticed recently in [6] for optimizers like gradient descent that add a vector (the parameter) to a co-vector (the update), creating a type mismatch. Luckily, if we have a Riemannian metric around; i.e., a positive symmetric matrix $G(\theta)$ defined for each point θ of the parameter space, we can transform a co-vector into vector via what is sometimes called the *musical isomorphism* (see [36] for more details): $\mu \to G^{-1}\mu$. Therefore, the correct way to write the gradient flow ODE is with the help of an underlying metric tensor $G(\theta)$ on the parameter space given by

$$\dot{\theta}(t) = G(\theta(t))^{-1}\nabla L(\theta(t)). \tag{43}$$

In this form, the gradient flow ODE has the exact same form in every coordinate system. Therefore writing the gradient flow ODE in an invariant form coincides exactly with preconditioning the gradient field by the inverse metric tensor $A = G^{-1}$.

Natural metric on the loss surface. There is a natural metric on the loss surface, which has been identified in [5] Appendix A.2 and studied in [49]:

$$\mathcal{G}(\theta) = \mathbb{I} + g(\theta)g(\theta)^T, \quad \text{where} \quad g(\theta) = \nabla L(\theta). \tag{44}$$

Intuitively, this metric says that if you move by a small amount v from a point θ in parameter space, then you actually move a distance approximated by $\|v\|_{\mathcal{G}} = \sqrt{v^T\mathcal{G}(\theta)v}$ on the loss surface. It thus seems natural to precondition the gradient field by this metric tensor; i.e. $\mathcal{G}(\theta)^{-1}\nabla L(\theta)$ since it would scale back large gradients. Luckily, this is easy thanks to the next lemma:

Lemma 3 *The gradient $g(\theta)$ is an eigenvector of the matrix $\mathcal{G}(\theta) = \mathbb{I} + g(\theta)g(\theta)^T$ with eigenvalue $(1 + \|g(\theta)\|^2)$. This means that $G^\alpha(\theta)g(\theta) = (1 + \|g(\theta\|^2)^\alpha g(\theta)$.*

Proof First of all, we have that $\mathcal{G}(\theta)g(\theta) = g(\theta) + g(\theta)g(\theta)^T g(\theta) = (1 + \|g(\theta)\|^2)g(\theta)$, which proves the first part. The second part comes from the general fact that if a matrix G is invertible with eigenvalue λ for eigenvector v then v is also an eigenvector of $\mathcal{G}^\alpha$ but with eigenvalue λ^α ■

Adagrad-like conditioning. Unfortunately, the resulting preconditioner $\mathcal{G}^{-1}(\theta)\nabla L(\theta)$ was not helpful. It seems that integrating past information from the trajectory (as done in Adam or AdaGrad for instance) is important here. We decided to accumulate past information in a similar way as done in AdaGrad [18] through a diagonal approximation $\operatorname{diag} G_n$ where $G_n = \sum_{l=1}^n g(\theta_l)g(\theta_l)^T$. This yields the preconditioner

$$A_n'(\theta) = (1 + \operatorname{diag} G_n)^{-12}. \tag{45}$$

We chose the square root of the inverse above because it worked better in practice, and because it is also the choice made in Adam and AdaGrad. Note that our preconditioner differs only by the constant factor 1 which in AdaGrad is an hyperparameter ϵ that needs to be tune:

$$A_n(\theta) = (\epsilon + \operatorname{diag} G_n)^{-12}, \tag{46}$$

We tried both preconditioners in Appendix D.2 Fig. 8 and Fig. 9, and ours worked noticeably better than AdaGrad.

Appendix D. Experiments

We used two different TPU configurations:

- `SMALL-TPU`: a single Google Cloud TPU V2
- `COLAB-TPU`: a single Google Cloud TPU V3 accessed via a Google Colab.
- `LARGE-TPU`: a single Google Cloud TPU V4

D.1. Generalization gap for RK methods in the large batch regime (Figure 1)

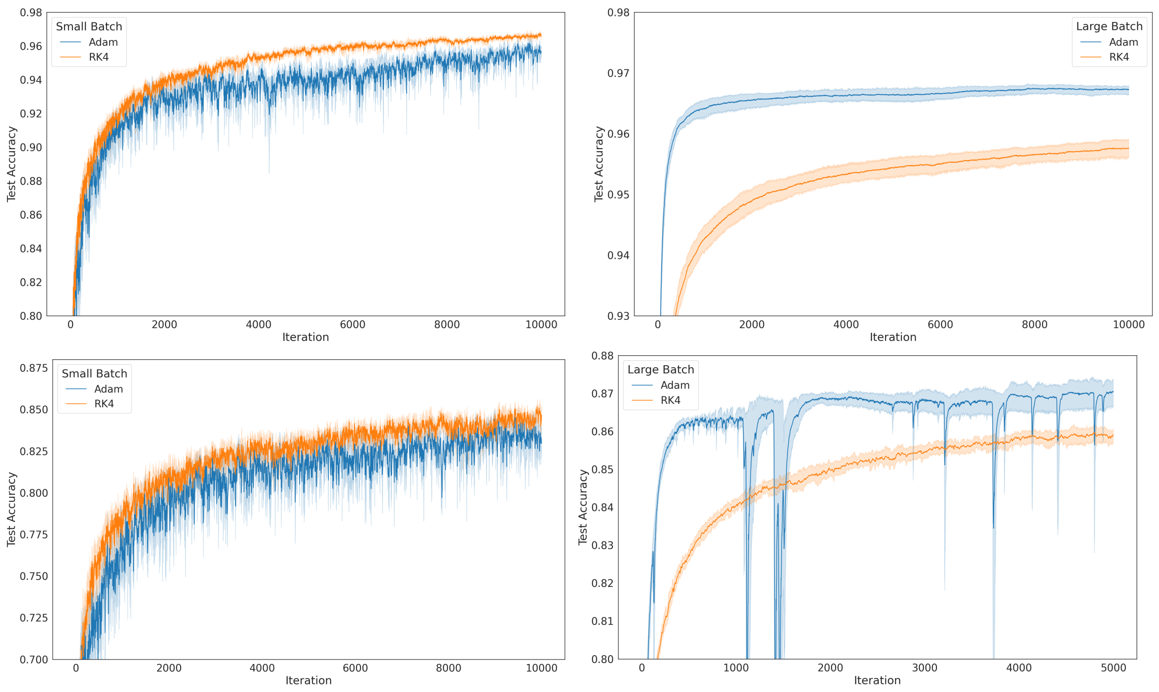

Figure 1: For MNIST (**top row**) and Fashion MNIST (**bottom row**) trained on MLP, vanilla RK4 achieves better test accuracy compared to Adam for small batches (**left column**) while we observe a generalization gap for large batches (**right column**). See Appendix D.1 for experiment details.

Left column (small batch regime): RK4 has an advantage over Adam in the small batch regime (batch size is 16) for both MNIST (top) and Fashion MNIST (bottom). In both cases, we trained a 3 hidden layer MLP with 500 neurons in each layer trained for 10000 steps on 5 seeds with metrics reported at each iteration. **Top (MNIST):** Adam learning rate was tuned to 0.001 while the decay parameters were set to Optax defaults [31], and RK4 learning rate was tuned to 0.003. The learning curves for this experiment are in Fig. 2. **Bottom (Fashion MNIST):** Adam learning rate was tuned to 0.001 while the decay parameters were set to Optax defaults [31], and RK4 learning rate was tuned to 0.003. The learning curves for this experiment are in Fig. 3. The 5 random seeds for each experiment sweep took roughly 10 minutes of training on the `COLAB-TPU` configuration for each workload.

Right column (large batch regime): RK4 suffers from a generalization gap over Adam. In the large batch regime (batchsize=60000) for both MNIST (top) and Fashion MNIST (bottom). In both cases, we trained a 3 hidden layer MLP with 500 neurons in each layer trained for 10000 steps on 5 seeds with metrics reported at each iteration. **Top (MNIST):** Adam learning rate was tuned to 0.001 while the decay parameters were set to Optax defaults [31], and RK4 learning rate was tuned to 0.004. The learning curves for this experiment are in Fig. 4. **Bottom (Fashion MNIST):** Adam learning rate was tuned to 0.001 while the decay parameters were set to Optax defaults [31], and RK4 learning rate was tuned to 0.003. The learning curves for this experiment are in Fig. 5. The 5 random seeds for each experiment took roughly 4 hours of training on a `COLAB-TPU` configuration for each workload.

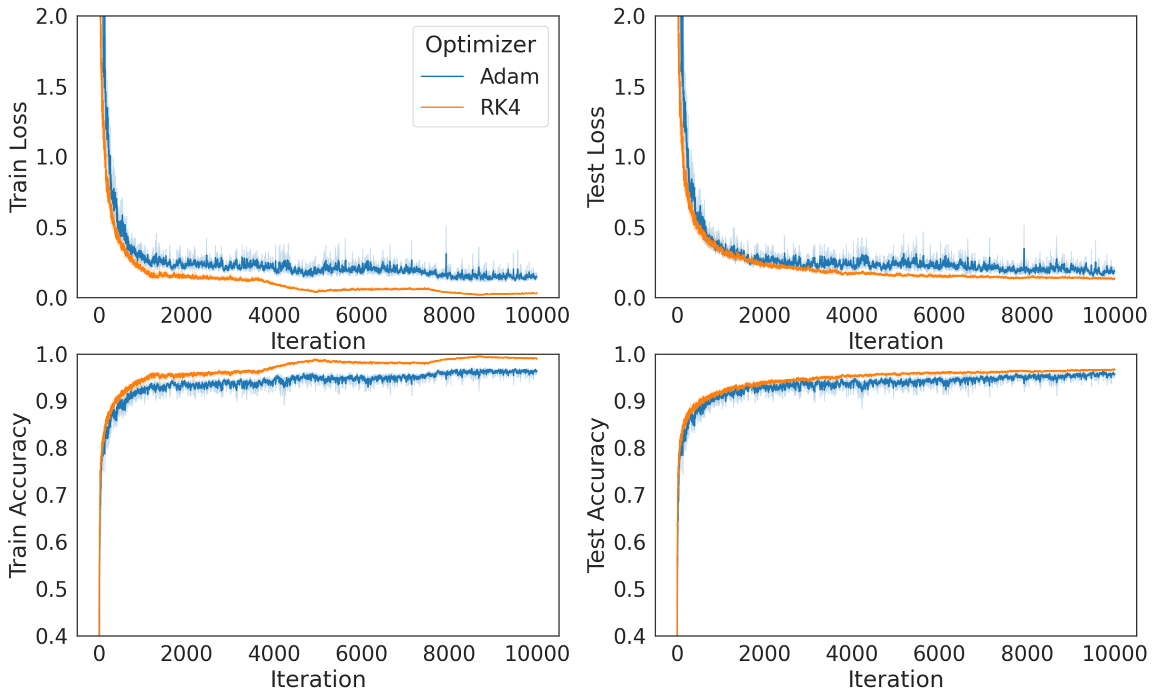

Figure 2: RK4 is competitive with Adam on MNIST with a 3 hidden layer MLP of 500 neurons each when trained with small batches of size 16 for 10000 steps. Adam's learning rate was tuned to 0.001 while the decay parameters were set to Optax defaults [31], and RK4 learning rate was tuned to 0.003; 5 seeds.

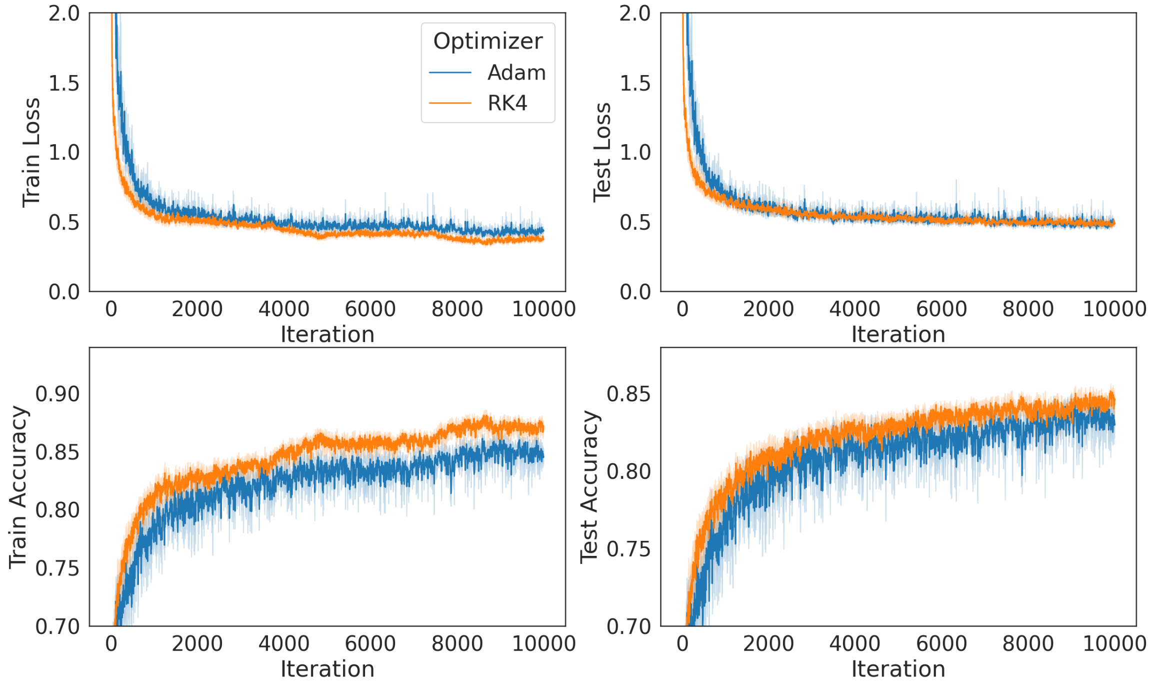

Figure 3: RK4 is competitive with Adam on Fashion MNIST with a 3 hidden layer MLP of 500 neurons each when trained with small batches of size 16 for 10000 steps. Adam's learning rate was tuned to 0.001 while the decay parameters were set to Optax defaults [31], and RK4 learning rate was tuned to 0.003; 5 seeds.

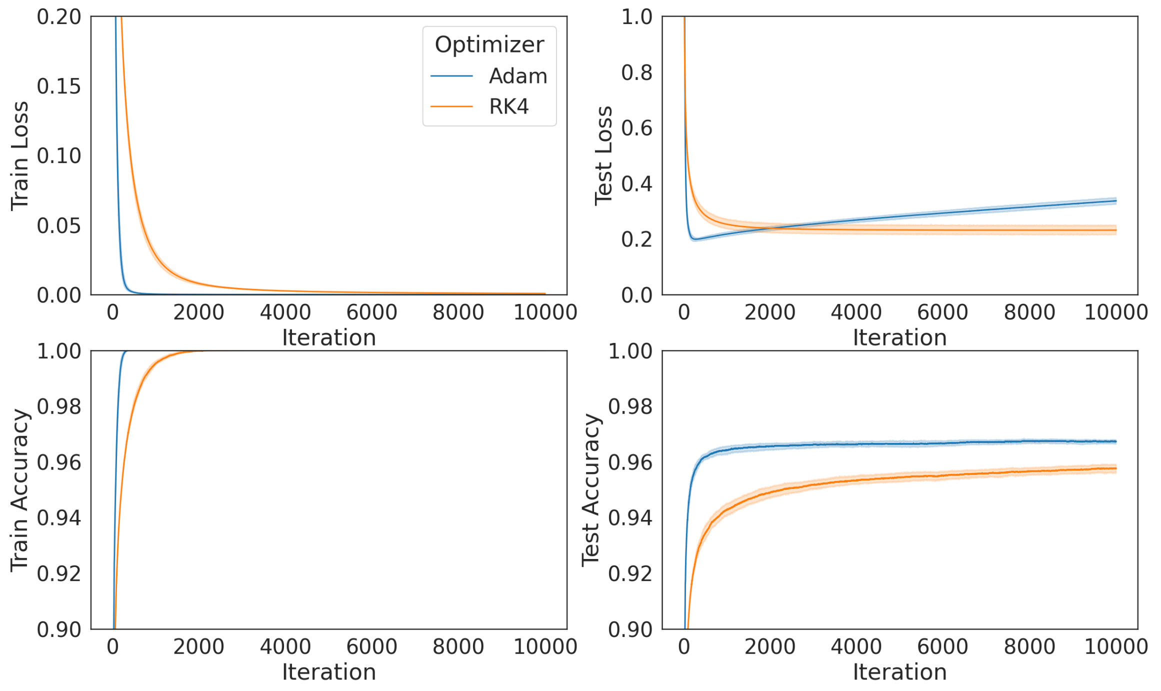

Figure 4: RK4 suffers from a generalization gap w.r.t. Adam on MNIST with a 3 hidden layer MLP of 500 neurons each when trained with full batches of size 60000 for 10000 steps. Adam's learning rate was tuned to 0.001 while the decay parameters were set to Optax defaults [31], and RK4 learning rate was tuned to 0.004; 5 seeds.

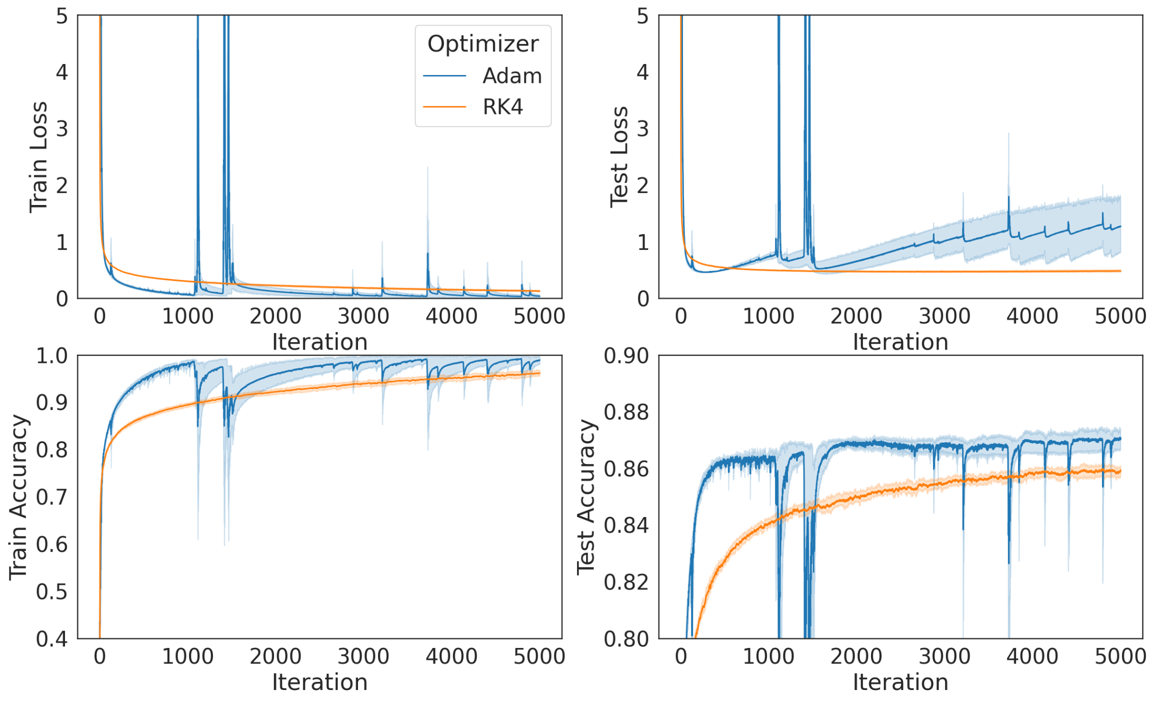

Figure 5: RK4 suffers from a generalization gap w.r.t. Adam on Fashion MNIST with a 3 hidden layer MLP of 500 neurons each when trained with full batches of size 60000 for 10000 steps. Adam's learning rate was tuned to 0.001 while the decay parameters were set to Optax defaults [31], and RK4 learning rate was tuned to 0.003; 5 seeds.

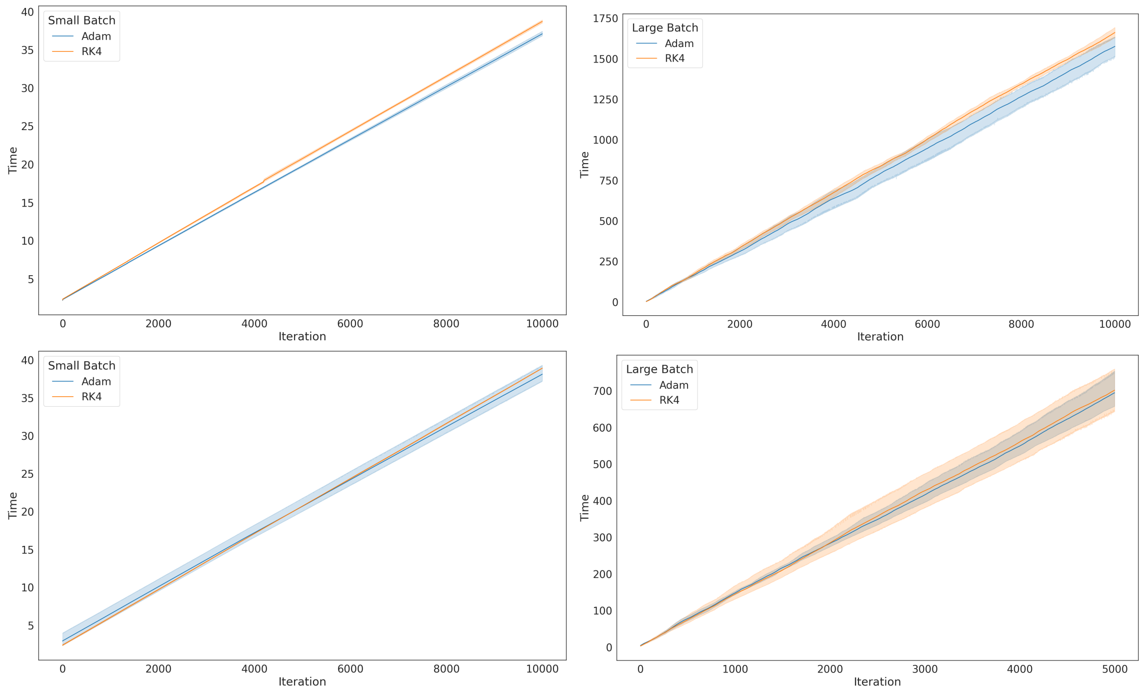

Figure 6: Wall-clock time versus steps between Adam and vanilla RK4 for MNIST (**top row**) and Fashion MNIST (**bottom row**) in the small batch regime (**left column**) and the large batch regime (**right column**).

D.2. Bridging the gap with preconditioning (Figure 7)

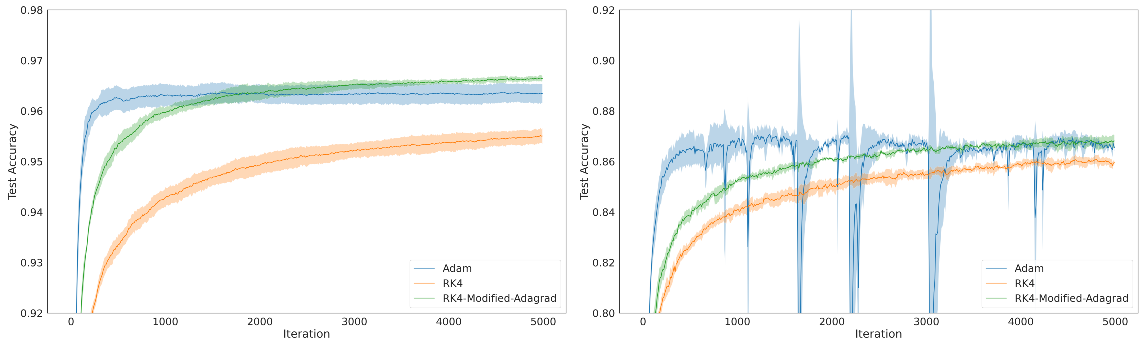

Figure 7: RK4 with AdaGrad-like preconditioning helps bridge the gap between Adam and vanilla RK4 in the large batch regime for MLP trained on MNIST (**left**) and Fashion-MNIST (**right**). See Appendix D.2 for experiment details. Additional experiments on CIFAR-10 and a ResNet-18 model yield similar results; see Fig. 17 and Appendix E.2.

The modified AdaGrad preconditioning for RK4 helps bridge the generalization gap with Adam on MNIST (left) and Fashion MNIST (right). For the MNIST dataset, the tuned learning rates were as follows: Adam (0.002), RK4 (0.004), RK4 + ADGR (0.032), and RK4 + Modified-ADGR (0.064). Similarly, for the Fashion MNIST dataset, the learning rates were: Adam (0.002), RK4 (0.003), RK4 + ADGR (0.016), and RK4 + Modified-ADGR (0.032). The model architecture consists of three fully connected layers, each with 500 neurons. Models are trained using the full training dataset in each iteration (step). Fig. 8 and Fig. 9 display the training curves for these experiments alongside the previously presented test curves for comparison. The random seeds took roughly 5h of training on the `COLAB-TPU` configuration for each workload.

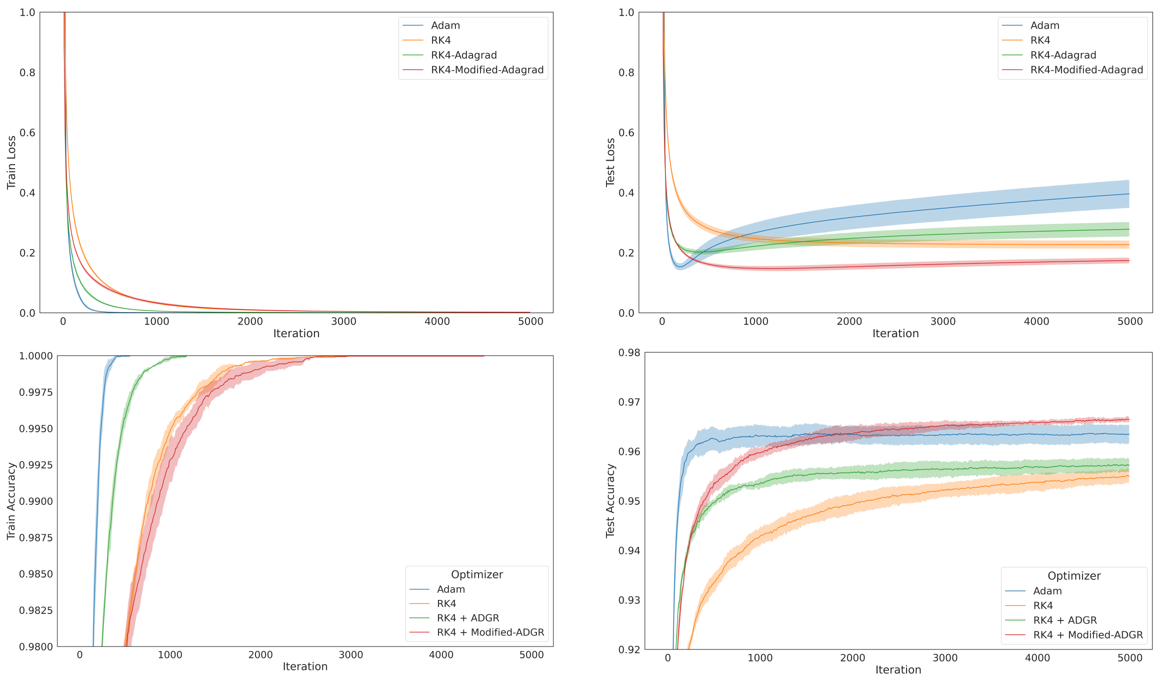

Figure 8: RK4 with preconditioning competes effectively with Adam on MNIST when using a 3-layer MLP (500 neurons per layer) trained with full-batch gradient descent for 5,000 steps.

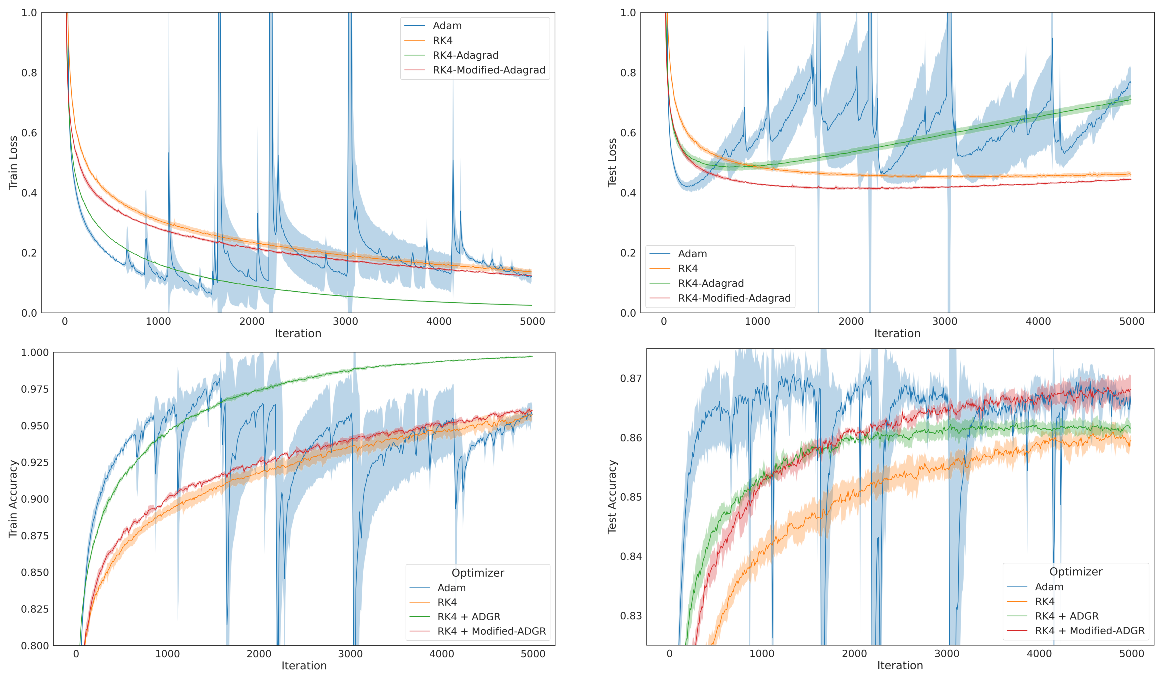

Figure 9: RK4 with preconditioning competes effectively with Adam on Fashion MNIST when using a 3-layer MLP (500 neurons per layer) trained with full-batch gradient descent for 5,000 steps.

D.3. Bridging the gap with adaptive learning rates (Figure 10)

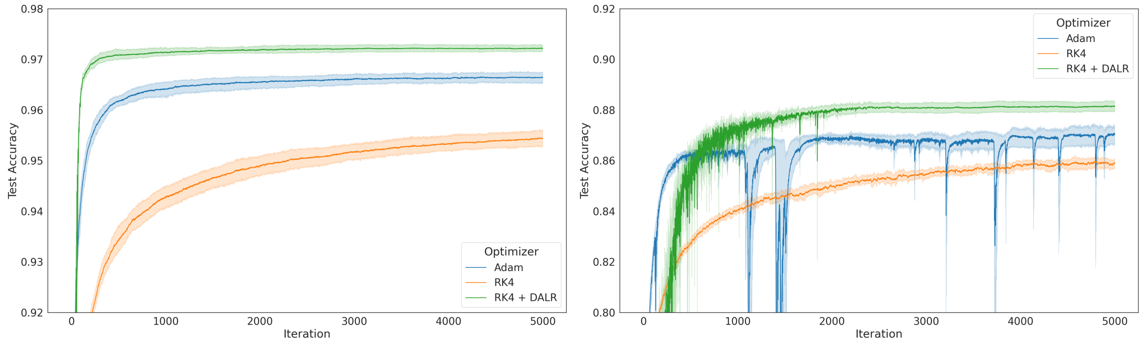

Figure 10: RK4 combined with an adaptive learning rate bridges the generalization gap in the large batch setting and is competitive with Adam for MLP trained on MNIST (**left**) and Fashion-MNIST (**right**). See Appendix D.3 for experiment details. Additional experiments on CIFAR-10 and a ResNet-18 model yield similar results; see Fig. 18 and Appendix E.3.

RK4 used with the DALR adaptive learning rate bridges the gap between and even surpasses Adam on MNIST (left) and Fashion MNIST (right). The experiment settings for Adam and RK4 for both workloads are the exact same as for the experiments in Fig. 1 right column (large batch setting). The DALR adaptive learning rate has been tuned with values $p = 0.8$ and $c = 4.0$ for MNIST (left), and $p = 0.8$ and $c = 1.0$ for Fashion MNIST (right). The learning curves for MNIST are in Fig. 11 and those for Fashion MNIST are in Fig. 12. The 5 random seeds took roughly 6h of training on the `COLAB-TPU` configuration for each workload.

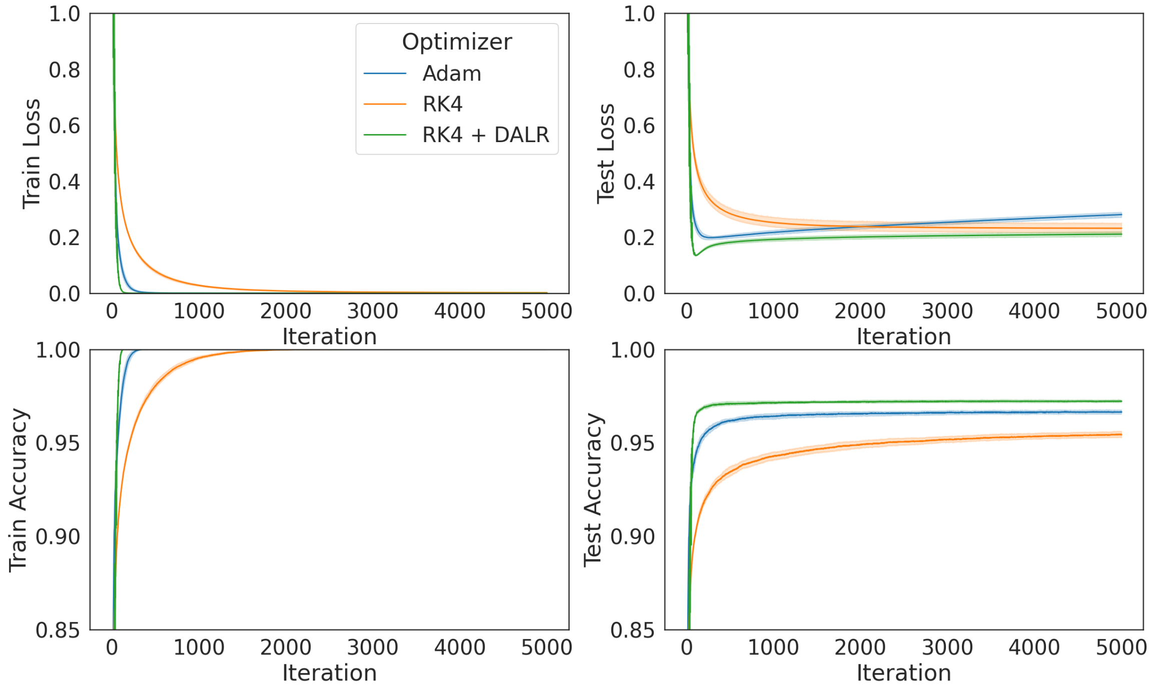

Figure 11: RK4 with adaptive learning rate (DALR) competes effectively with Adam on MNIST when using a 3-layer MLP (500 neurons per layer) trained with full-batch gradient descent for 5,000 steps.

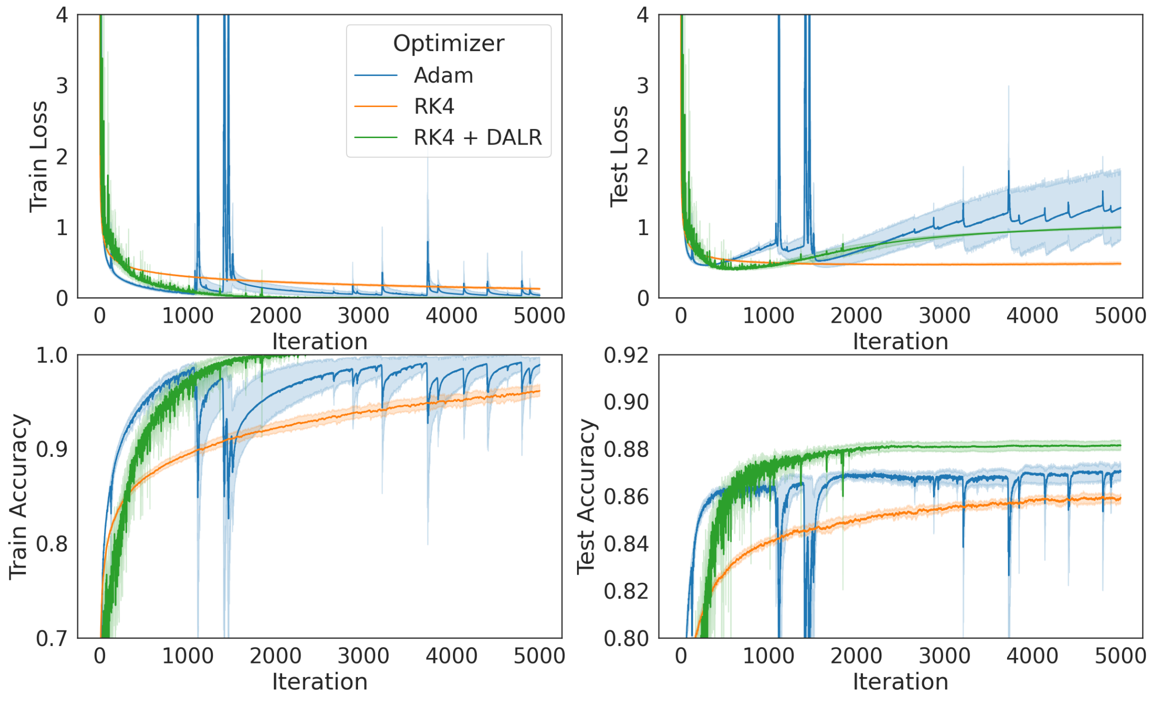

Figure 12: RK4 with adaptive learning rate (DALR) competes effectively with Adam on Fashion MNIST when using a 3-layer MLP (500 neurons per layer) trained with full-batch gradient descent for 5,000 steps.

D.4. Bridging the gap with momentum (Figure 13)

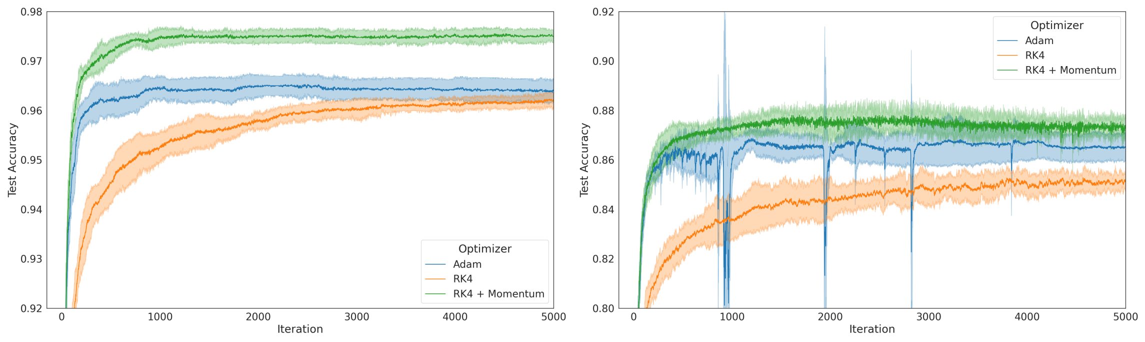

Figure 13: Combining momentum with RK4 bridges the generalization gap in the large batch regime and achieves better test accuracy than both Adam and vanilla RK4 on MNIST (**left**) and Fashion MNIST (**right**). See Appendix D.4 for experiment details. Additional experiments on CIFAR-10 and a ResNet-18 model yield similar results; see Fig. 18 and Appendix E.3.

RK4 used with momentum (see Section 3.3) surpasses Adam on MNIST (left) and Fashion MNIST (right). The experiment settings for Adam and RK4 for both workloads are the exact same as for the experiments in the second row. The momentum has been tuned with values $\beta = 0.95$ and learning rate 0.004 for MNIST (left) and $\beta = 0.95$ and learning rate 0.001 for Fashion MNIST (right). The learning curves for MNIST are in Fig. 14 and those for Fashion MNIST are in Fig. 15.

All 5 random seeds for each experiment took 5 hours and 10 minutes of training on the `SMALL-TPU` configuration for each workload.

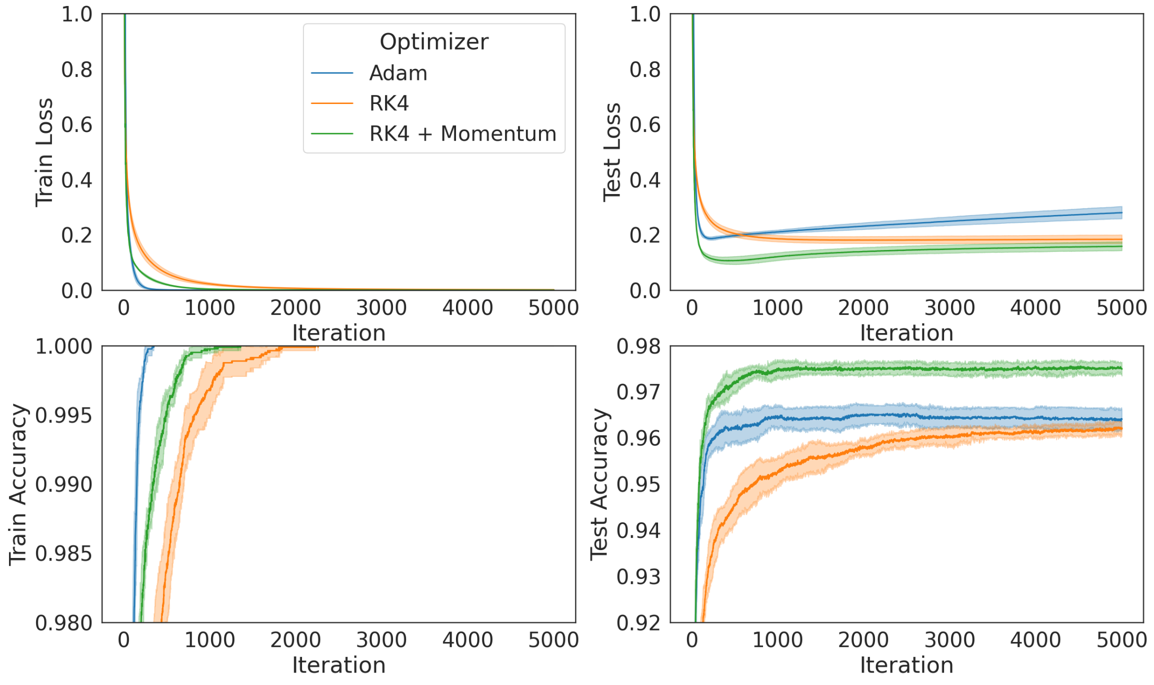

Figure 14: RK4 with momentum competes effectively with Adam on MNIST when using a 3-layer MLP (500 neurons per layer) trained with full-batch gradient descent for 5,000 steps.

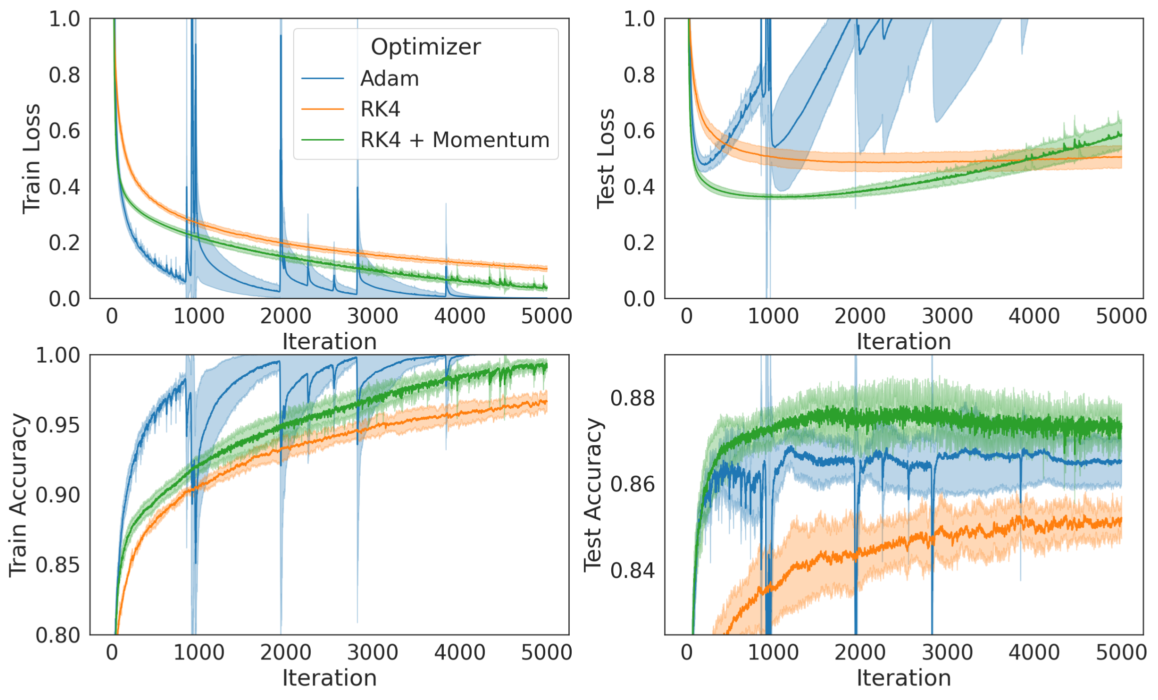

Figure 15: RK4 with momentum competes effectively with Adam on Fashion MNIST when using a 3-layer MLP (500 neurons per layer) trained with full-batch gradient descent for 5,000 steps.

D.5. Experiment Details for Table 1

This table presents a comparison of the best test set accuracy for various workloads, each comprising a specific dataset and model pairing. Models within each workload were trained using either the RK4 optimizer or a competitive baseline optimizer. For a fair comparison, all models for a given workload were trained for an identical number of steps. The reported accuracy and standard for each optimizer and workload combination is derived from measuring peak test accuracy, repeated across 5 independent trials with each independent trial employing a distinct random seed per trial. This seed influenced model weight initialization, data shuffling, and the stochastic elements of any data augmentation techniques.

For each of the experiments described below, we used one the following TPU configurations

- Google Cloud TPU V2 arranged as a 4x4 TPU pod
- Google Cloud TPU V5 arranged as 2x4 TPU pod

MNIST on DNN We trained a 3 layer DNN with 500 neurons per layer for 10,000 steps measuring training evaluation metrics every step. For the baseline, we used Adam optimizer with learning rate 0.001 and batch size 16. The 5 random seeds took roughly 3 hrs of training on TPUv2. For the RK4 optimizer, we used a learning rate of 0.003 and a batch size of 16. The 5 random seeds took roughly 7 hrs of training on TPUv2.

MNIST on CNN We trained a 2 layer CNN model, both layers having kernel size [3, 3], the first convolutional layer with 20 filters and the second convolutional layer with 10 filters. The model was trained for 10,000 steps measuring evaluation metrics every 10 steps. The 5 random seeds took roughly 3 hrs of training on TPUv2. For the baseline, we used stochastic gradient descent with a momentum of 0.9, constant learning rate of 0.01, batch size 64 and l2 weight decay factor of 0.002. For the RK4 optimizer, we used a constant learning rate of 0.1 with batch size of 64 and no weight decay. The 5 random seeds took roughly 7 hrs of training on TPUv2.

Fashion-MNIST on DNN We trained a 3 layer DNN with 500 neurons per layer for 10,000 steps measuring training evaluation metrics every step. For the baseline, we used Adam optimizer with learning rate 0.001 and batch size 16. The 5 random seeds took roughly 3 hrs of training on TPUv2. For the RK4 optimizer, we used a learning rate of 0.003 and a batch size of 16. The 5 random seeds took roughly 7 hrs of training on TPUv2.

Fashion-MNIST on CNN We trained a 2 layer CNN model, both layers having kernel size [3, 3], the first convolutional layer with 20 filters and the second convolutional layer with 10 filters. The model was trained for 10,000 steps measuring evaluation metrics every 10 steps. For the baseline, we used stochastic gradient descent with a momentum of 0.9, constant learning rate of 0.01, batch size 512 and l2 weight decay factor of 0.0005. The 5 random seeds took roughly 3 hrs of training on TPUv2. For the RK4 optimizer, we used a constant learning rate of 0.5 with batch size of 64 and no weight decay. The 5 random seeds took roughly 7 hrs of training on TPUv2.

CIFAR-10 on WRN We trained a Wide-ResNet (WRN), specifically the WRN 28-10 model, as proposed by [65]. This network has a depth of 28 convolutional layers and a widening factor of 10. The WRN 28-10 was trained for 117,187 steps (or 200 epochs) using stochastic gradient descent with a momentum of 0.9, a learning rate of 0.1 with cosine decay and batch size of 128. The 5 random seeds took roughly 19 hrs of training on TPUv2. For the RK4 optimizers, we used a base learning rate of 0.5 with cosine decay and a batch size of 512. The 5 random seeds took roughly 32 hrs of training on TPUv2.

CIFAR-100 on WRN We trained a Wide-ResNet (WRN), specifically the WRN 28-10 model, as proposed by [65]. This network has a depth of 28 convolutional layers and a widening factor of 10. The WRN 28-10 was trained for 117,187 steps using stochastic gradient descent with a momentum of 0.9, a learning rate of 0.1 with cosine decay, a batch size of 128 and l2 weight decay factor of 0.0005. The 5 random seeds took roughly 2 hrs of training on TPUv5. For the RK4 optimizers, we used a base learning rate of 0.75 with cosine decay and a batch size of 256. The 5 random seeds took roughly 3 hrs of training on TPUv5.

ImageNet on ViT We trained a Vision Transformer (ViT) model, ViT-B/16 [16], on the ImageNet ILSVRC 2012 dataset. This model processes images by dividing them into 16x16 patches, which are then linearly embedded and supplied with positional embeddings before being fed into a series of Transformer encoder blocks. The ViT-B/16 model was trained directly on the ImageNet 1000-class dataset for 186,666 steps with evaluation every 1,866 steps. The baseline optimizer we used was NAdamW. This optimizer combines the Nesterov-accelerated adaptive moment estimation (NAdam) [17] with the decoupled weight decay approach from AdamW [40]. We used $\beta_1 = 0.9414$, $\beta_2 = 0.9768$, $\epsilon = 1 \times 10^{-8}$, weight decay of 0.02, and 20% label smoothing. The 5 random seeds took roughly 12 hrs of training on TPUv5. The RK4 optimizer was trained using learning rate 0.16 with cosine warmup and batch size 1024. The 5 random seeds took roughly 26 hrs of training on TPUv5.

Appendix E. Additional Experiments

E.1. Wall-clock time Adam v.s. RK4 on CIFAR-10

In Fig. 16, we trained a ResNet-18 [29, 65] on CIFAR-10 [35] for 1000 steps measuring the wall-clock time at every step for 5 random seeds. Adam's learning rate was tuned to 0.001 while the decay parameters were set to Optax defaults [31], and RK4 learning rate was tuned to 0.004. **Left plot:** The batch-size was set to 1. The 5 random seeds took roughly 2h of training on a single Google Cloud TPU V3 for each workloads. **Right plot:** The batch-size was set to 8192. The 5 random seeds took roughly 23h of training on the `COLAB-TPU` configuration for each workloads.

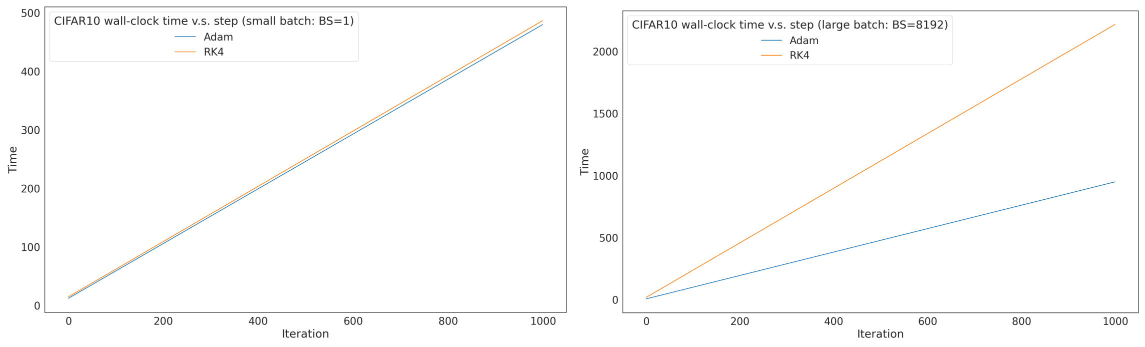

Figure 16: The wall-clock time for Adam and RK4 on CIFAR-10 is essentially the same for small batches but more than doubles for RK4 for large batches.

E.2. Preconditioners on CIFAR-10

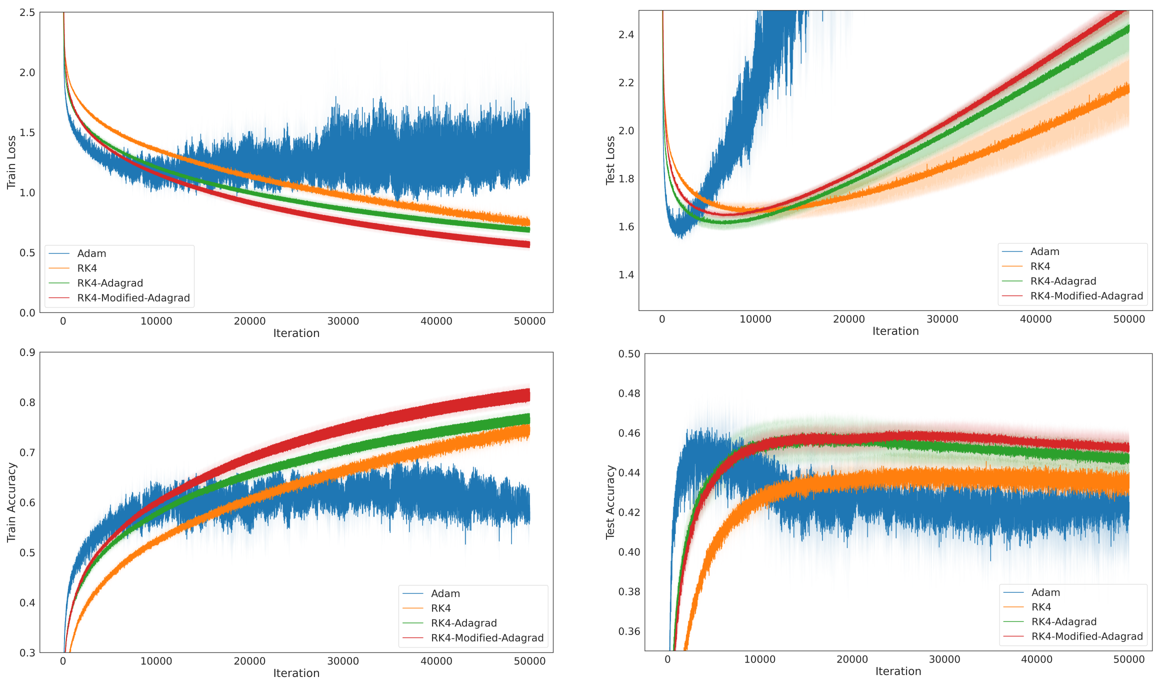

Figure 17: The figure displays side-by-side training curves (left) versus test curves (right), illustrating that RK4 with preconditioning competes effectively with Adam on CIFAR-10 when using a 3-layer MLP (500 neurons per layer) trained with a batch size of 1024 for 50,000 steps. For these experiments, the tuned learning rates were: Adam (0.002), RK4 (0.001), RK4 + ADGR (0.008) and RK4 + Modified-ADGR (0.004). The random seeds took roughly 6h of training on the `COLAB-TPU` configuration.

E.3. Momentum and adaptive learning rate

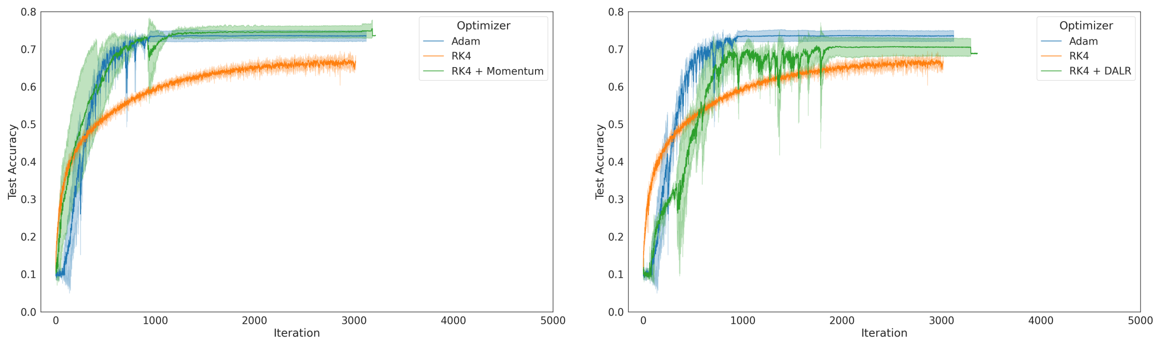

Figure 18: The figure displays side-by-side training curves for momentum (left) and adaptive learning rate (DALR) (right) comparing it with Adam on CIFAR-10 when using a ResNet-18 [30] trained with a batch size of 8192 for 3,000 steps. For these experiments, the tuned learning rates were: Adam (0.001), RK4 (0.004), RK4 + Momentum (0.004, $\beta = 0.99$) and RK4 + DALR. No regularization, augmentation, or schedule was used in either case. The 5 random seeds for each experiment took roughly 3h 20 minutes of training on the `LARGE-TPU` configuration for each workload.

E.4. Related Work

RK methods in optimization. There have been surprisingly few and limited studies on the application of higher-order RK methods [26–28] to deep learning. To our knowledge there are only two bench-marking works, both comparing higher-order RK methods to SGD only. In [3], the authors observe better performance of a second-order RK method, the Heun method (applied to the gradient flow ODE) on MNIST, Fashion-MNIST, and CIFAR-10 with a CNN architecture over SGD. In [57], the classical 4$^{\text{th}}$-order RK method shows also improved test accuracy over SGD on the same workloads plus CIFAR-100, but with ResNet architectures. However, the paper provides limited specifics regarding hyperparameter configurations and the experiment setup, making it difficult to ascertain the precise extent of hyperparameter tuning applied in the comparisons. A very interesting work [20] uses a second order RK method on a relativistic Hamiltonian system to obtain Relativistic Gradient Descent (RGD), which is shown to interpolate between classical momentum and Nesterov momentum depending on the values of some hyper-parameters. This RGD is not evaluated in the context of neural network optimization in that paper. Another noteworthy work in this context is [50] where they used a gradient regularized RK method to train GANs with good success. Lastly, first-order implicit RK methods have been used with benefit in the case of Physics-Informed Neural Networks (PINN) [38, 62]. Devising optimization schemes using ODE's has been also considered in [8, 19]. In a sort of converse way, a number of works have observed that central optimizers in deep learning can be realized as *first-order* RK methods applied to specific differential equations: gradient descent, momentum methods [7, 20, 23, 34, 45, 54], and accelerated gradient methods [20–22, 58, 67]. These findings point toward a potentially high impact of bridging RK theory with deep learning optimization, a connection we believe has been understudied.

Preconditioning. Most modern deep-learning optimizers involve a form of raw gradient modification by the application of a matrix to the gradient to decrease its variability [19, 21]. For instance, the Adam family uses at each step a diagonal preconditioning matrix formed by averaging the gradient squares over the trajectory [2, 11, 17, 18, 21, 33, 41, 61, 66]. The resulting matrix can be thought of as an approximation of the Hessian inverse. More recently, the Shampoo family of optimizers uses block diagonal approximations of the same Hessian inverse [2, 55]. Amari in [1], defines the natural gradient as the preconditioning of the raw gradient with the inverse of a natural metric on the parameter space generated by the family of probability distributions. In Section 3.1 we explain how to incorporate preconditioning into RK updates by performing local modifications of gradient flow, and we analyze a variation of the AdaGrad preconditioner [18].

Momentum. Another technique prevalent in deep-learning is that of averaging the gradient updates along the trajectory to tame the raw gradient. This technique is known as momentum [48, 53, 59]. One can understand momentum as an application of a first-order RK scheme to a special Hamiltonian equation with friction using the loss as potential function [20, 23, 34, 45, 54]. The use of higher-order RK methods to solve this Hamiltonian system with friction has been explored in [20] leading to improved stability. Here we take a different approach by simply averaging RK updates instead of raw gradient updates leading to improved test performance as explained in Section 3.3.

Adaptive learning rates. Higher-order RK methods can follow the exact solutions of ODE very precisely [27]. However, they suffer when the ODE is stiff [26], which means that its vector field has large local variations. One common method for handling this is to use an adaptive step-size [26] that automatically reduces the step-size in these regions of large variation. We propose a modification of a recent adaptive learning rate, the Drift-Adjusted Learning rate (DAL) from [52], in conjunction with RK methods in Section 3.2. Note that other learning rates like the Polyak step-size [13, 46, 48] or the NGN step-size [47] may also be worth studying in this context.