A Genealogy of Optimizers

Siddharth Choudhary and Claude (Anthropic)

***Abstract*—This is a guided tour through the optimizers used to train neural networks, from stochastic gradient descent to the recent matrix-aware methods Muon, Shampoo, and MuonH. The organizing idea is that each optimizer fixes a specific, namable failure of the one before it: momentum gives the optimizer a memory, Adam adds per-parameter learning rates, AdamW corrects how weight decay interacts with them, the sign-based methods strip the machinery back to its load-bearing core, and the matrix-aware methods exploit structure that per-parameter methods throw away. We trace that lineage, and along the way isolate a single recurring axis — how each method turns a gradient into a step — that explains most of the progress. An interactive version with live simulations is available online.**[1]

I. SGD and the stochastic gradient

A neural network is a giant pile of numbers, the parameters. Training means finding the values that make the model's predictions good, measured by a single scalar called the *loss*: lower is better. So training is a search problem. Picture the loss as the height of a landscape and every setting of the parameters as a location on it; training is finding the lowest valley while wandering blindfolded.

You cannot see the landscape, but at any point you can feel which way the ground tilts. That tilt is the gradient, and it points uphill, so you step the opposite way:

$$w \leftarrow w - \eta\, g, \tag{1}$$

where $g = \nabla_w L$ and the step size η (the learning rate) is the one knob you really tune — too small and you crawl, too big and you overshoot.

The *stochastic* part is the trick that makes this practical. Computing the true gradient means evaluating the loss on every training example before taking a single step, which is wildly impractical. Instead we compute a noisy estimate of the gradient on a small random batch: wrong in the details but right on average, and nearly free. So instead of one careful step on all the data we take thousands of slightly drunk steps on slivers of it. The noise that looks like a bug is part of why SGD works at scale — random kicks help models escape bad regions.

Three problems are already visible. First, even at the optimum the noise never goes away, so w keeps wobbling. Second, every step is memoryless: if the last twenty gradients all said "go right," the twenty-first ignores that history. Third, in real models different parameters have wildly different gradient scales, and one global learning rate is asking a lot. Momentum fixes the second and partly the first; RMSProp and Adam tackle the third.

[1] https://itzsid.github.io/publications/optimizer-ladder.html

II. Momentum

Stretch the loss bowl into a long narrow valley — steep walls one way, a gentle slope along the length. This is what real loss landscapes look like: a few directions are extremely curved and most are nearly flat. Plain SGD lurches across the steep walls and barely creeps along the valley floor, because its steps are memoryless.

The fix is to treat the gradient not as the direction to move but as a force on a velocity that the optimizer carries from step to step:

$$v \leftarrow \beta v + g, \qquad w \leftarrow w - \eta\, v. \tag{2}$$

With β near 1 (typically 0.9), useless directions whose gradients flip sign cancel in v, while the consistent valley direction accumulates — at steady state with $\beta = 0.9$ the velocity is roughly ten gradients' worth, so progress along the valley is about ten times faster. Under noise, the independent noise terms also average toward zero in v while the signal survives.

Momentum fixes the memoryless complaint and partly the noise one, but it does *not* solve heterogeneous gradient scales: a parameter with a $100\times$ larger gradient still gets a $100\times$ larger step. That is the next target.

III. Adam: from AdaGrad to RMSProp to Adam

In a real network the single global learning rate is catastrophic. The bias of a softmax output and a weight two layers back can have gradient magnitudes differing by $1000\times$. The field converged on per-parameter learning rates computed automatically from gradient statistics, in three steps.

AdaGrad keeps a running sum of squared gradients per parameter and divides the step by its square root, $w \leftarrow w - \eta\, g/\sqrt{G+\epsilon}$ with $G \leftarrow G + g^2$. Loud parameters get shrunk, quiet ones keep a near-full rate. The catch: G only grows, so after a few thousand steps every effective learning rate approaches zero and the optimizer stalls [1].

RMSProp replaces the running sum with an exponential moving average, $v \leftarrow \beta_2 v + (1-\beta_2) g^2$, so v tracks *recent* magnitude and plateaus instead of growing forever [2]. **Adam** then stacks the two ideas — a momentum EMA of the gradient and an RMSProp EMA of the squared gradient, both bias-corrected [3]:

$$\begin{aligned} m &\leftarrow \beta_1 m + (1-\beta_1)\, g, & v &\leftarrow \beta_2 v + (1-\beta_2)\, g^2, \\ \hat{m} &= m/(1-\beta_1^t), & \hat{v} &= v/(1-\beta_2^t), \end{aligned} \tag{3}$$

$$w \leftarrow w - \eta\, \hat{m}/(\sqrt{\hat{v}} + \epsilon). \tag{4}$$

The defaults $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$ have proven nearly universal. The punchline took the field years to internalize: if gradients are roughly stationary then $\hat{v} \approx g^2$ and

TABLE I
ADAM'S UPDATE AS A SIGNAL-TO-NOISE RATIO. ONLY THE RATIO OF SIGNAL TO MAGNITUDE MATTERS, NOT THE MAGNITUDE.

scenario	signal	σ	m	$\sqrt{v}$	update
A clean strong	0.3	0	0.3	0.3	1.00
B moderate noise	0.3	1.0	0.3	1.04	0.29
C weak signal	0.1	1.0	0.1	1.00	0.10
D pure noise	0	1.0	0	1.0	0.00
E oscillating	± 1	0	0	1.0	0.00
F loud parameter	100	0	100	100	1.00

$\hat{m} \approx g$, so $\hat{m}/\sqrt{\hat{v}} \approx \operatorname{sign}(g)$. Adam's update is a momentum-smoothed sign of the gradient: every parameter moves by about η regardless of its gradient magnitude. That is why default Adam works on heterogeneous networks, and why it became the reflexive choice for nearly a decade.

IV. ADAM AS A SIGNAL-TO-NOISE RATIO

A cleaner reading of $\hat{m}/\sqrt{\hat{v}}$ is a per-parameter signal-to-noise ratio. Write each gradient as signal + noise with noise standard deviation σ. Then $m \to$ signal (noise cancels in the mean), $v \to \text{signal}^2 + \sigma^2$ (the variance survives in the second moment), so

$$\frac{m}{\sqrt{v}} \to \frac{\text{signal}}{\sqrt{\text{signal}^2 + \sigma^2}}, \tag{5}$$

exactly the signal-to-noise ratio, bounded in $[-1, 1]$ in steady state. Table I walks through the regimes: a strong clean gradient gives a full step, a noisy one a small step, pure noise or oscillation gives nothing, and magnitude alone (rows A versus F) does not change the answer.

One caveat: the bound holds in steady state. During transients — early training, or just after a learning-rate change — the mismatch between the short β_1 window and the long β_2 window can push the update above 1. This is part of why gradient clipping is standard for transformer training.

V. ADAMW: THE ONE-LINE FIX

Adam was published in 2014; AdamW, the version everyone actually uses, came in 2017, and the change is one line [4]. Weight decay gently pulls parameters toward zero each step to improve generalization. For SGD there is an identity: shrinking the parameters is equivalent to adding λw to the gradient, so for years people implemented decay by folding λw into the gradient. That is correct for any optimizer that just multiplies the gradient by η and subtracts.

Adam does not. It divides by $\sqrt{v}$ first, so a folded-in decay term becomes $\eta\,\lambda w/\sqrt{v}$: parameters with large v (loud gradients) get decayed *less*, quiet ones *more*. This is exactly backwards — the parameters most at risk of overfitting receive the least regularization. The fix is to decouple decay from the gradient and apply it directly to the weights:

$$w \leftarrow w - \eta\,\hat{m}/(\sqrt{\hat{v}} + \epsilon), \qquad w \leftarrow w - \eta\,\lambda\,w. \tag{6}$$

Now every parameter gets the same proportional decay regardless of v, and λ is an independent knob. AdamW trained essentially every transformer between 2017 and roughly 2023 — the silent default behind GPT-2, GPT-3, BERT, T5, ViT, and CLIP.

VI. INTERLUDE: SIGNSGD

So far the story has been "track more statistics, use them more cleverly." SignSGD asks the rude question: what if we track *less*? Since Adam's update is approximately $\operatorname{sign}(g)$, why not compute the sign directly [5]:

$$w \leftarrow w - \eta\,\operatorname{sign}(g). \tag{7}$$

No m, no v, no per-parameter scale — zero optimizer state. The wins are concrete: AdamW costs two extra values per parameter (roughly 560 GB of m and v for a 70B model in fp32), while SignSGD costs nothing, and in distributed training each gradient compresses to one bit, a $32\times$ bandwidth reduction.

But it has two real problems. It carries no magnitude information, so a parameter near its optimum overshoots by a fixed amount forever. And it is fragile to noise: the sign of a near-zero gradient buried in noise is essentially random, and SignSGD takes a full step in that random direction. SignSGD is rarely used as-is, but it cleanly shows that the *directionally normalized step* — everything moves by about η — is doing most of the work that makes Adam great. Add back a single momentum buffer and most of the brittleness disappears.

VII. LION: ONE BUFFER INSTEAD OF TWO

Lion emerged from a large program search over optimizer update rules and is what won [6]. It is substantially simpler than AdamW, uses half the optimizer state, and is competitive across many large-scale benchmarks (its wins are task- and tuning-dependent). The rule smooths the gradient and then takes the sign, with two different mixing rates:

$$\begin{aligned} \text{update} &= \operatorname{sign}(\beta_1 m + (1-\beta_1)g), \\ w &\leftarrow w - \eta\,(\text{update} + \lambda w), \\ m &\leftarrow \beta_2 m + (1-\beta_2)g, \end{aligned} \tag{8}$$

with $\beta_1 = 0.9$, $\beta_2 = 0.99$. It steps using a fast EMA (window ≈ 10) but stores a slow one (window ≈ 100); the asymmetry materially outperforms using one rate for both. Lion's update is in $\{-1, 0, +1\}$ per parameter — strict sign, no fractional confidence — and its per-parameter normalization comes from the sign function rather than $\sqrt{v}$. What is intellectually interesting is not that Lion beats AdamW (sometimes it does, sometimes not) but that it is competitive while throwing away half the machinery.

VIII. THE HIDDEN INVARIANT: PER-PARAMETER NORMALIZATION

Looking back from SGD to Lion, a single axis explains most of why each method outperforms the last: how does a parameter's gradient magnitude affect how far it moves? Table II makes it explicit.

This corrects a common error: momentum does *not* solve heterogeneity. It accumulates gradients over time within each

TABLE II
EFFECTIVE STEP PER PARAMETER i, AND WHETHER IT SCALES WITH THE GRADIENT MAGNITUDE. THE ADAPTIVE AND SIGN-BASED METHODS FLATTEN IT.

optimizer	effective step	scales with $\lvert g_i \rvert$?
SGD	$\eta\, g_i$	yes, linearly
SGD+momentum	$\eta\, v_i$	yes, linearly
AdamW	$\eta\, m_i/\sqrt{v_i} \approx \eta\ \mathrm{sign}(g_i)$	no, flat
Lion	$\eta\ \mathrm{sign}(\beta m_i + (1-\beta) g_i)$	no, flat
SignSGD	$\eta\ \mathrm{sign}(g_i)$	no, flat

parameter (temporal smoothing) but preserves per-parameter magnitude. The thing that solved heterogeneity was the $\sqrt{v}$ divisor or the sign function, both of which produce roughly equal step sizes per parameter — spatial normalization across parameters. This reframes Lion's success: it is not that $\mathrm{sign}(\cdot)$ is smarter than $m/\sqrt{v}$, but that both reach the same destination, flat per-parameter steps, and Lion gets there with one buffer and one operation. It also points at the next question: is per-parameter even the right granularity? Real parameters are organized into matrices, which have richer structure.

IX. MUON: PER-MATRIX ORTHOGONALIZATION

Every method so far smuggles in the assumption that the parameters are a flat list. Adam scales each scalar entry of a weight matrix independently; the fact that those scalars form a matrix — with rows, columns, and structure that survives matrix multiplication — is thrown away. Muon's bet is that this structure is exactly the information you need [10], [11].

Any matrix has a singular value decomposition $M = U\Sigma V^\top$, where Σ holds the singular values — how much the matrix stretches space along each principal direction. Gradient matrices in training have very unequal singular values: a few directions dominate, most are nearly zero, so a vanilla step makes progress in one or two directions and barely moves the rest. Muon computes the momentum-smoothed gradient and then flattens all its singular values to 1 before stepping:

$$u \leftarrow \beta u + g, \quad \hat{u} = \mathrm{orth}(u), \quad W \leftarrow W - \eta\, \hat{u}, \qquad (9)$$

where $\mathrm{orth}(\cdot)$ replaces Σ with the identity (keeping U, V). Computing this by SVD every step is too slow, so Muon uses a degree-5 Newton–Schulz iteration — a polynomial in the matrix, applied about five times, that pushes the singular values toward 1 using only matrix multiplications. The standard coefficients are tuned for speed, not accuracy: they leave the singular values in a band near 1 rather than exactly at 1.

Muon applies to hidden weight matrices only; embeddings, biases, and norm parameters use AdamW alongside it. Practically, Muon set the speed records for nanoGPT training and, after work showing it scales, was used at the trillion-parameter scale for Kimi K2 [12]. It also transfers its learning rate cleanly across model sizes when paired with μP-style width scaling [13].

X. THE RIGHT KIND OF UNIFORM: DIRECTIONS, NOT ENTRIES

The normalization story has a subtlety worth isolating. "Uniform per entry," as Adam and SignSGD give, is uniform in the basis of matrix *coordinates* — this row, that column — but that basis is arbitrary. What a weight matrix *does* is set by its singular *directions*; rotate the coordinate frame and every entry changes while the function does not move. So the real question is not whether every entry moved equally, but whether the update pushed equally hard in every direction it acts on. A per-entry-uniform step can still be wildly lopsided in direction space, pouring almost all of its effect into one or two singular directions; when the gradient concentrates, $\mathrm{sign}(\cdot)$ of a near-rank-one matrix is still near-rank-one, funneling the entire step into a single direction.

Muon's orthogonalization keeps the gradient's directions but flattens its singular values, so the update is isotropic: it advances every direction equally. This is a genuinely high-dimensional effect (in two dimensions, sign matrices are usually orthogonal and the distinction vanishes). The deeper reason per-matrix beats per-parameter is basis-independence: per-entry normalization asks "is each number stepping equally," a question whose answer depends on how the matrix was laid out, while Muon asks "is each direction stepping equally," a question about the layer's real geometry [14].

XI. WHAT FIRST-ORDER OPTIMIZERS CAN'T SEE

Every method so far — including Muon — uses one piece of information per parameter: the gradient, which gives the slope but not the *curvature*. Two parameters with the same gradient but different curvature want very different steps: the gently curving one can take a big step safely, the sharply curving one is about to overshoot. For a single parameter, curvature is the second derivative H, and the quadratic Taylor model $L(w+\Delta) \approx L(w) + g\Delta + \frac{1}{2}H\Delta^2$ is minimized by Newton's step $\Delta = -g/H$ — walk until the slope hits zero.

For N parameters the curvature is the Hessian $H_{ij} = \partial^2 L/\partial w_i \partial w_j$, an $N \times N$ matrix. Materializing it needs N backward passes; for modern models with $N \approx 10^9$–10^{12} that is not merely expensive but physically impossible, so full Newton is dead on arrival. The escape hatch: we rarely want H itself, only Hv for some vector, and that is one extra backward pass (a Hessian-vector product). The practical second-order story is therefore: do not materialize H, do not invert it, get cheap factored approximations that match the matrix structure of the network. Shampoo is the field's main matrix-aware answer.

XII. SHAMPOO: CURVATURE-AWARE PRECONDITIONING

Shampoo predates Muon (2018) and was the first matrix-aware optimizer to gain attention; Muon got most of the practical traction [7]. Shampoo approximates the full Hessian

with two matrices per layer, one for how rows interact and one for columns:

$$\begin{aligned} L &\leftarrow \beta L + (1-\beta)\, GG^\top, \\ R &\leftarrow \beta R + (1-\beta)\, G^\top G, \\ W &\leftarrow W - \eta\, L^{-1/4}\, G\, R^{-1/4}. \end{aligned} \tag{10}$$

The $-1/4$ exponents come from a Kronecker-factorization argument: the Hessian is approximately $L \otimes R$, so its inverse square root applied to G multiplies by $L^{-1/4}$ on the left and $R^{-1/4}$ on the right. (The original 2018 version accumulated L, R as un-decayed sums, exactly the AdaGrad form; modern variants use the EMA shown here [8], [9].)

Muon versus Shampoo. The two answer different questions about the same gradient matrix. Muon asks which directions the gradient spans and steps equally along all of them, whitening the update's singular values; it is purely first-order and cheap every step. Shampoo asks how curved the loss is along rows and columns and rescales by that estimated curvature; it is second-order-flavored and pays for it with eigendecompositions. At small and moderate scale the two are about tied, and Shampoo sometimes edges ahead. The gap opens at large scale for reasons unrelated to the update rule: Muon transfers its learning rate cleanly under μP, Shampoo's matrix roots are expensive enough that the preconditioner is recomputed only every few hundred steps (running on stale curvature), and it has more knobs. SOAP — Shampoo's preconditioner with Adam in its eigenbasis — folds the two together and often beats both [9].

One fairness point that bites in practice: Shampoo is only competitive when it preconditions a *momentum-smoothed* gradient. Feed its curvature factors a raw, noisy minibatch gradient and the $L^{-1/4}, R^{-1/4}$ terms amplify the noise — a near-zero curvature estimate becomes a large multiplier. Muon's noise-robustness comes from the same place, its own momentum buffer.

XIII. MuonH: pinning the weight norm

Every method so far concerns the update *direction*. MuonH instead asks how to control the *magnitude* of the weights [15]. Modern transformers wrap each weight matrix in RMSNorm, which is scale-invariant: scaling the input by any positive factor leaves the output unchanged, so the weight matrix's magnitude does not affect the function the layer computes — only its direction does. Yet weights still grow during training (roughly as $\sqrt{t}$ without decay), which inflates the parameter scale relative to the step. AdamW handles this with decoupled decay; MuonH handles it more radically by *fixing* the magnitude.

Pick a target radius $R = \|W_0\|_F$ and, after every step, project the weight matrix back to that radius:

$$\begin{aligned} U_t &= -\eta\, R\ \mathrm{Normalize}(\mathrm{update}), \\ W_{t+1} &= R\ \mathrm{Normalize}(W_t + U_t). \end{aligned} \tag{11}$$

The weight lives on a hypersphere of radius R for all of training, and η becomes the relative step size in units of weight norm. Geometrically, the retraction cancels any radial component of the update, so only tangential motion survives.

MuonH does not uniformly beat MuonW (Muon with ordinary weight decay). At lower learning rates MuonH wins; at higher rates MuonW pulls ahead, because letting the norm grow provides an implicit annealing schedule. There is also a failure mode at scale: the retraction multiplies the matrix by a single scalar, shrinking all singular values proportionally, so the trailing singular values that Muon patiently grows get repeatedly pulled back down and the spectrum collapses into a few dominant modes — the "spectral squeeze." Weight magnitude has nonetheless emerged as a third axis of optimizer design, orthogonal to update direction and curvature-awareness.

XIV. Coda: the through-line

SGD took small noisy steps along the gradient. Momentum gave the optimizer a memory. Adam added per-parameter learning rates that we then read as a signal-to-noise ratio. AdamW noticed the obvious way to add weight decay silently broke it. SignSGD and Lion showed the per-parameter normalized step was the active ingredient all along. Muon jumped from per-parameter to per-matrix, orthogonalizing each update; Shampoo tracked actual curvature per row and column; MuonH opened a third axis, controlling weight magnitude directly. Each method either asked what the previous one was failing to use, or what it was doing redundantly that could be thrown away — and both directions produced wins. The matrix-aware methods are new and evolving, but the through-line should hold: every optimizer fixes a specific failure of the one before it, and noticing the failure clearly is the hard part.

The lineage above rests on a broader body of work: stochastic approximation and momentum [16], [17], [18], [19], analyses and critiques of adaptive methods [20], [21], the normalization literature [22], [23], [24], [25], [26], second-order and natural-gradient methods [27], [28], [29], the spectral-norm and duality view of matrix optimizers [30], [31], [32], and learning-rate schedules and large-batch scaling [33], [34], [35].

References

[1] J. Duchi, E. Hazan, and Y. Singer, "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization (AdaGrad)," JMLR 12, 2011.

[2] T. Tieleman and G. Hinton, "Lecture 6.5 — RMSProp," Coursera: Neural Networks for Machine Learning, 2012.

[3] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," ICLR, 2015. arXiv:1412.6980.

[4] I. Loshchilov and F. Hutter, "Decoupled Weight Decay Regularization (AdamW)," ICLR, 2019. arXiv:1711.05101.

[5] J. Bernstein, Y.-X. Wang, K. Azizzadenesheli, and A. Anandkumar, "signSGD: Compressed Optimisation for Non-Convex Problems," ICML, 2018. arXiv:1802.04434.

[6] X. Chen et al., "Symbolic Discovery of Optimization Algorithms (Lion)," NeurIPS, 2023. arXiv:2302.06675.

[7] V. Gupta, T. Koren, and Y. Singer, "Shampoo: Preconditioned Stochastic Tensor Optimization," ICML, 2018. arXiv:1802.09568.

[8] H.-J. M. Shi, T.-H. Lee, S. Iwasaki, J. Gallego-Posada et al., "A Distributed Data-Parallel PyTorch Implementation of the Distributed Shampoo Optimizer," 2023. arXiv:2309.06497.

[9] N. Vyas, D. Morwani, R. Zhao et al., "SOAP: Improving and Stabilizing Shampoo using Adam," 2024. arXiv:2409.11321.

[10] K. Jordan, "Muon: An Optimizer for Hidden Layers in Neural Networks," blog, 2024. https://kellerjordan.github.io/posts/muon/.

[11] J. Liu et al. (Moonshot AI), "Muon is Scalable for LLM Training," 2025. arXiv:2502.16982.

[12] Kimi Team (Moonshot AI), "Kimi K2: Open Agentic Intelligence," 2025. arXiv:2507.20534.

[13] G. Yang et al., "Tensor Programs V: Tuning Large Neural Networks via Zero-Shot Hyperparameter Transfer (μP)," NeurIPS, 2021. arXiv:2203.03466.

[14] L. Chen, J. Li, and Q. Liu, "Muon Optimizes Under Spectral Norm Constraints," 2025. arXiv:2506.15054.

[15] L. Ren et al., "Rethinking Language Model Scaling under Transferable Hypersphere Optimization (HyperP / MuonH)," 2026. arXiv:2603.28743.

[16] H. Robbins and S. Monro, "A Stochastic Approximation Method," Annals of Mathematical Statistics 22(3), 1951.

[17] B. T. Polyak, "Some Methods of Speeding Up the Convergence of Iteration Methods (heavy-ball momentum)," USSR Comput. Math. and Math. Phys. 4(5), 1964.

[18] Y. Nesterov, "A Method for Solving the Convex Programming Problem with Convergence Rate $O(1/k^2)$," Soviet Math. Doklady 27, 1983.

[19] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, "On the Importance of Initialization and Momentum in Deep Learning," ICML, 2013.

[20] S. J. Reddi, S. Kale, and S. Kumar, "On the Convergence of Adam and Beyond (AMSGrad)," ICLR, 2018. arXiv:1904.09237.

[21] L. Balles and P. Hennig, "Dissecting Adam: The Sign, Magnitude and Variance of Stochastic Gradients," ICML, 2018. arXiv:1705.07774.

[22] S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," ICML, 2015. arXiv:1502.03167.

[23] J. L. Ba, J. R. Kiros, and G. E. Hinton, "Layer Normalization," 2016. arXiv:1607.06450.

[24] T. Salimans and D. P. Kingma, "Weight Normalization: A Simple Reparameterization to Accelerate Training of Deep Neural Networks," NeurIPS, 2016. arXiv:1602.07868.

[25] B. Zhang and R. Sennrich, "Root Mean Square Layer Normalization (RMSNorm)," NeurIPS, 2019. arXiv:1910.07467.

[26] T. van Laarhoven, "L2 Regularization versus Batch and Weight Normalization," 2017. arXiv:1706.05350.

[27] S. Amari, "Natural Gradient Works Efficiently in Learning," Neural Computation 10(2), 1998.

[28] J. Martens and R. Grosse, "Optimizing Neural Networks with Kronecker-factored Approximate Curvature (K-FAC)," ICML, 2015. arXiv:1503.05671.

[29] R. Anil, V. Gupta, T. Koren, K. Regan, and Y. Singer, "Scalable Second Order Optimization for Deep Learning," 2020. arXiv:2002.09018.

[30] J. Bernstein and L. Newhouse, "Old Optimizer, New Norm: An Anthology," 2024. arXiv:2409.20325.

[31] J. Bernstein and L. Newhouse, "Modular Duality in Deep Learning," 2024. arXiv:2410.21265.

[32] N. J. Higham, "Functions of Matrices: Theory and Computation (Newton–Schulz iteration)," SIAM, 2008.

[33] I. Loshchilov and F. Hutter, "SGDR: Stochastic Gradient Descent with Warm Restarts," ICLR, 2017. arXiv:1608.03983.

[34] Y. You, I. Gitman, and B. Ginsburg, "Large Batch Training of Convolutional Networks (LARS)," 2017. arXiv:1708.03888.

[35] Y. You, J. Li, S. Reddi, J. Hseu, S. Kumar et al., "Large Batch Optimization for Deep Learning: Training BERT in 76 Minutes (LAMB)," ICLR, 2020. arXiv:1904.00962.