SVEN: SINGULAR VALUE DESCENT AS A COMPUTATIONALLY EFFICIENT NATURAL GRADIENT METHOD

Anonymous authors
Paper under double-blind review

ABSTRACT

We introduce Sven (Singular Value dEsceNt), a new optimization algorithm for neural networks that exploits the natural decomposition of loss functions into a sum over individual data points, rather than reducing the full loss to a single scalar before computing a parameter update. Sven treats each data point's residual as a separate condition to be satisfied simultaneously, using the Moore-Penrose pseudoinverse of the loss Jacobian to find the minimum-norm parameter update that best satisfies all conditions at once. In practice, this pseudoinverse is approximated via a truncated singular value decomposition, retaining only the k most significant directions. We show that Sven can be understood as a natural gradient method generalized to the overparametrized regime, recovering natural gradient descent in the underparametrized limit. We test Sven on a variety of regression and classification tasks, including small-scale language modeling with transformers, and find that it is competitive with leading baselines such as Adam, Muon, and K-FAC. We also discuss Sven's memory overhead, which presents a barrier to scaling under a naive implementation, and introduce an optimized implementation that keeps memory usage on par with standard baselines under mild restrictions on model architecture. Beyond standard machine learning benchmarks, we anticipate that Sven will find natural application in scientific computing settings where custom loss functions decompose into several conditions.

1 INTRODUCTION

Every standard loss function is a sum. Whether fitting a regression model or training a classifier, the objective decomposes into a sum over data points, with each term encoding an individual condition the model should satisfy. Despite this structure, the dominant paradigm in machine learning (ML) discards it immediately. Gradient descent reduces the entire collection of conditions to a single scalar and updates parameters accordingly, treating the decomposition as an implementation detail rather than a source of information.

In this paper, we develop a new optimizer for neural networks which takes a global view of this loss decomposition. Our methodology, which we call *Sven* (**S**ingular **V**alue d**E**sce**N**t), can be viewed as an efficient approximation to natural gradient methods, extrapolated to the overparametrized regime. At each training step, rather than computing a single gradient direction for the total loss, Sven asks a more ambitious question: given the individual residuals of every data point in the batch, what single parameter update would bring each of them closest to zero simultaneously? This is a linear algebra problem, the solution to which makes use of the Moore-Penrose pseudoinverse (Moore, 1920; Penrose, 1955). In practice, computing the full pseudoinverse can be expensive, so Sven approximates it via a truncated singular value decomposition (SVD), retaining only the k most significant directions. The result is an optimizer that is aware of the geometry of the loss landscape in a way that standard gradient descent is not.

We focus our proof-of-concept studies on relatively small-scale regression, classification, and language modeling tasks. Scaling to larger models is a natural direction for future work, and one we believe is tractable due to a memory-efficient implementation of Sven that we introduce in this work.[1] We find that Sven performs on par with standard baselines such as Adam(W) (Kingma & Ba, 2015;

[1] Efficient relative to a naive implementation; ours uses more memory than Adam, but only modestly so.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

Algorithm 1 Sven update step. The truncated SVD considers only the k largest singular values, and sets singular values that are a factor of `rtol` smaller than the largest singular value to zero. Appendix E details computational backends for the SVD.

Input: Jacobian M, residual vector R, rank k, tolerance rtol, parameters θ, learning rate η
Output: Updated parameters θ'
$U, s, V^T \leftarrow \text{TruncatedSVD}(M, k, \text{rtol})$ ▷ Compute truncated SVD
for $i = 1$ to $\text{length}(s)$ **do**
  if $s_i > 0$ **then**
    $s_i^{-1} \leftarrow 1/s_i$
  else
    $s_i^{-1} \leftarrow 0$
  end if
end for
$M^+ \leftarrow V\text{diag}(s^{-1})U^T$ ▷ Compute Moore-Penrose inverse
$\delta\theta \leftarrow -\eta M^+ R$ ▷ Compute parameter update
return $\theta + \delta\theta$

Loshchilov & Hutter, 2019), SGD, RMSprop (Hinton et al., 2012), Muon (Jordan et al., 2024), and K-FAC (Martens & Grosse, 2015). It is also competitive with methodologically similar methods such as Half-Inverse Gradients (HIG) (Schnell et al., 2022) and Jacobian Descent (JD) (Quinton & Rey, 2024), and with the popular quasi-Newton method L-BFGS (Liu & Nocedal, 1989). Beyond standard ML benchmarks, Sven finds natural application in scientific computing, with a recent application in theoretical physics (Anonymous, 2026).

The paper is organized as follows. Section 2 derives the update rule, relates it to natural gradients, and describes our optimized implementation; Section 3 surveys related work and optimizer baselines; Section 4 gives experimental results and analyzes Sven's learning dynamics; Section 5 concludes and states limitations. The appendices contain extensive additional information on Sven's mathematical basis (B, C), experimental details (D, E), further experimental results (F–P) including an illuminating singular value spectrum analysis (G), stability studies (Q), and memory/compute profiling (R).

2 METHODOLOGY

In this section, we present the Sven update rule and build intuition for it through the lens of least-squares regression. The theoretical backbone is in Appendices B and C, which develop the connection to natural gradient methods from a functional perspective. Neither is needed to understand the core methodology.

In ML, a common problem is the minimization of a loss function composed of several components:

$$L(\theta) = \sum_{\alpha \in \aleph} \ell^\alpha(\theta), \tag{1}$$

where each sub-loss is non-negative, $\ell^\alpha(\theta) \geq 0$ for all $\theta \in \mathbb{R}^P$ and $\alpha \in \aleph$. Each term ℓ^α encodes a condition the model should satisfy, and the overall loss aggregates these conditions into a single objective. This structure naturally arises, for instance, in the L_2-loss for regression and the cross-entropy loss for classification, both of which decompose as a sum over individual data points. From now on, we take the index set $\aleph$ to enumerate individual data points, keeping the more general framework in mind and returning to it when relevant.

Despite this structure, minimizing $L(\theta)$ typically proceeds by applying gradient descent to the aggregate loss, disregarding its decomposition (though in practice this is usually done over batches). A more principled approach, however, can explicitly account for the fact that $L(\theta)$ aggregates several distinct conditions. The algorithm we propose, Sven, is given in Algorithm 1. We now describe its derivation and its relation to existing methods.

2.1 Inspiration from Least-Squares Regression

Although we will generalize to an arbitrary loss shortly, we begin by considering a regression problem, where we use the L_2-loss between our model $f_\theta(x)$ and some target function $g(x)$. Such a loss is of the form

$$L(\theta) = \sum_{\alpha \in \aleph} \big(\mathcal{R}^\alpha(\theta)\big)^2 = \sum_{\alpha \in \aleph} \big(f_\theta(\alpha) - g(\alpha)\big)^2. \tag{2}$$

Consider the linear expansion of the residual $\mathcal{R}^\alpha(\theta)$ around θ_0:

$$L(\theta_0 + \delta\theta) = \sum_{\alpha \in \aleph} \left(\mathcal{R}^\alpha(\theta_0) + \sum_i M_i^\alpha \, \delta\theta^i \right)^2 + \mathcal{O}\left(|\delta\theta|^2\right), \tag{3}$$

where the Jacobian is

$$M_i^\alpha \equiv \left. \frac{\partial \mathcal{R}^\alpha}{\partial \theta^i} \right|_{\theta=\theta_0} = \frac{\partial f_\theta(\alpha)}{\partial \theta^i}, \tag{4}$$

and we will omit the higher order terms from now on, taking them as understood. In such a situation, this can be viewed as a linear least squares regression problem for $\delta\theta^i$.

Within the region that the linear expansion is reliable, we seek to solve for all contributions simultaneously. That is, we seek solutions to

$$\mathcal{R}^\alpha(\theta_0) + \sum_i M_i^\alpha \, \delta\theta^i = 0, \tag{5}$$

such that the total loss is zero. Although a solution may not exist, the *closest* approximation to one (in a sense to be made precise shortly) is given by

$$\delta\theta^i = -\sum_{\alpha \in \aleph} (M^+)^i{}_\alpha \, \mathcal{R}^\alpha(\theta_0) \equiv -(M^+ R)_i, \tag{6}$$

where M^+ is the Moore–Penrose pseudoinverse of M (Moore, 1920; Penrose, 1955) and $R \in \mathbb{R}^{|\aleph|}$ is the vector of residuals. We defer the discussion of its efficient computation and approximation to later, and for now focus on its interpretation. For a linear system such as Equation 5, the update $\delta\theta^i$ in Equation 6 admits the following interpretation depending on the regime: with more conditions than parameters, Equation 5 is overdetermined, generically has no exact solution, and $\delta\theta^i$ is the unique least-squares minimizer; with fewer (overparametrized) it is the *minimum-norm solution* among those that minimize the residual. Our interest lies primarily in the second case, especially with large neural networks and batched data, though the first case is relevant for drawing connections with natural gradient methods. In either case, we propose a learning scheme where each update step consists of

$$\boxed{\delta\theta = -\eta M^+ R(\theta_0), \qquad M_i^\alpha \equiv \left. \frac{\partial \mathcal{R}^\alpha}{\partial \theta^i} \right|_{\theta=\theta_0}}, \tag{7}$$

and the proportionality constant η is the learning rate.

In a typical machine learning context, α indexes a batch of size B sampled from training set of size N, and the model has parameters $\theta \in \mathbb{R}^P$. The Jacobian M is a $B \times P$ matrix with rows containing per-sample loss gradients, and is generally distinct from the *model* Jacobian $\partial f_i / \partial \theta_j$.[2] We distinguish two notions of overparametrization: *batchwise* ($P > B$), which applies in any typical ML context and motivates the pseudoinverse Sven update rule; and *dataset-wise* ($P > N$), studied in Appendix I.

2.2 General Case

Let us now return to the generic loss of Equation 1, which we can rewrite suggestively as

$$L(\theta) = \sum_{\alpha \in \aleph} \left(\mathcal{R}^\alpha_{\text{eff}}(\theta) \right)^{\frac{2}{\kappa}}, \qquad \mathcal{R}^\alpha_{\text{eff}}(\theta) \equiv \left(\ell^\alpha(\theta) \right)^{\frac{\kappa}{2}}, \tag{8}$$

[2]The equality in equation 4 is a special case, and only holds for an MSE loss on a 1D model output.

where $\kappa > 0$ is a hyperparameter and we term $\mathcal{R}_{\text{eff}}$ *effective residuals*. Equation 8 is trivially true for every κ: it does not change the loss being minimized but rather the quantity being treated as the residual, and hence the rows of M. If $\kappa = 1$, then we are simply treating the $\sqrt{\ell^\alpha(\theta)}$ terms as if they were L_2-loss residuals $\mathcal{R}^\alpha(\theta)$, and the above derivation goes through essentially unchanged. For generic κ, this takes the form of an $L_{2/\kappa}$-loss, which does not have a closed form minimum, even in the linear approximation. We can, however, simply treat the $\ell^\alpha(\theta)^{\kappa/2}$ terms as if they were residuals of an L_2-loss, and use the effective residuals to define the update step and Jacobian in Equation 7. We emphasize that the $L_{2/\kappa}$ and L_2 norms differ, so this update step is κ-dependent.[3] Throughout this work we keep $\kappa = 2$ ($\mathcal{R}^\alpha_{\text{eff}} = \ell^\alpha$) as the default to avoid pathologies associated with taking fractional powers of generic loss function. We investigate varying κ in Appendix O, and find that it has little impact on Sven's performance.

2.3 Underparametrized Limit and the Relation to Natural Gradients

In gradient descent, parameters are updated according to

$$\delta\theta^i = -\eta \frac{\partial L(\theta)}{\partial \theta^i}, \tag{9}$$

where η is the learning rate. However, this can be generalized by introducing an inverse metric g^{ij} (with corresponding metric g_{ij}) on the space of parameters:

$$\delta\theta^i = -\eta \sum_j g^{ij} \frac{\partial L(\theta)}{\partial \theta^j}. \tag{10}$$

Such an inverse metric is called a preconditioner in the ML literature, with Adam being the most famous example (Kingma & Ba, 2015; Harvey, 2025; Absil et al., 2009; Fei et al., 2025; Hinton et al., 2012; Jordan et al., 2024). Natural gradients offer a principled, though computationally expensive, choice of preconditioner (Amari, 1998; Bernacchia et al., 2018; Shrestha, 2023). If one had access to this preconditioner at no cost, it would be provably the most efficient choice for training. In particular, in ideal conditions the loss decreases exponentially with training time, rather than with power-like scaling laws (Hestness et al., 2017; Kaplan et al., 2020).

For our purposes, when dealing with functions, the natural gradient metric is given by

$$g_{ij} = \frac{1}{|\mathcal{D}|} \sum_{x \in \mathcal{D}} \frac{\partial f_\theta(x)}{\partial \theta^i} \frac{\partial f_\theta(x)}{\partial \theta^j}, \tag{11}$$

where $\mathcal{D}$ is our dataset. An analogous expression exists for probability distributions, where the metric is given the name Fisher Information Metric (FIM).

Returning to the regression example of Equation 2, consider the underparametrized limit where the parameter index i spans fewer values than the data index α. In this special case, the Moore-Penrose inverse can be written as

$$M^+ = (M^T M)^{-1} M^T. \tag{12}$$

Substituting this into Equation 7, we find

$$\delta\theta^i \propto -\sum_j \left[\left(M^T M \right)^{-1} \right]^{ij} \frac{\partial L(\theta)}{\partial \theta^j}, \tag{13}$$

which, up to a normalization, is the same as what we expect from natural gradient methods, where the sum is replaced with an integral over the data.

2.4 Overparametrized Limit and Our Algorithm

Natural gradient methods are not straightforwardly defined in the overparametrized regime: the metric of Equation 11 becomes singular once P exceeds the number of data points and cannot be inverted to yield an update. This is precisely the regime modern ML operates in, and the natural

[3] In the underparametrized limit, the choice of κ only rescales the update and can be absorbed into η; it becomes meaningful only in the overparametrized regime, due to the singular value truncation.

resolution — the pseudoinverse of g_{ij} — is an intractably large $P \times P$ matrix. Sven instead takes the pseudoinverse of the $B \times P$ Jacobian M, which is much smaller in the typical case where $B \ll P$. The derivation in the overparametrized limit follows the same initial steps as the previous section until the substitution of the specific form of the pseudoinverse (Equation 12). In the $P > B$ regime, we calculate the pseudoinverse M^+ using the SVD of the matrix M.

We can further approximate this inverse by retaining only the k largest singular values, and the cost then depends on how the truncated pseudoinverse is obtained. Using randomized algorithms (Halko et al., 2011) to compute the truncated SVD of the $B \times P$ Jacobian costs $\mathcal{O}(kPB)$, a factor of k over a gradient descent step, and the Jacobian must be stored in memory. In Appendix E we show this cost can often be avoided, as the Sven update depends on M only through the eigendecomposition of the $B \times B$ Gram matrix MM^T and a vector–Jacobian product, so one step reduces to a constant number of forward and backward passes and a small eigendecomposition. In many cases, the Gram matrix can be accumulated from autograd artifacts without ever instantiating M fully in memory. The full cost is then *independent* of k, as k and `rtol` decide how many directions are inverted rather than how many are computed.

Sven can be understood as a natural gradient method generalized to the overparametrized regime. The pseudoinverse provides a principled update rule when the natural gradient metric becomes singular, and reduces cleanly to natural gradient descent in the underparametrized limit. Combining the above, we are left with the update step shown in Algorithm 1. Anonymized code for a PyTorch (Ansel et al., 2024) implementation of Sven is linked in App. A, with implementation details in Appendix E. All experiments in the main text use the efficient Sven optimization referenced above, which works for architectures *not* including BatchNorm (Ioffe & Szegedy, 2015) or tied weights. Appendix L applies Sven to a ResNet18 trained on CIFAR-10, where BatchNorm prohibits the efficient implementation and introduces larger memory overheads for Sven. Appendix R provides detailed memory and wall-time profiling data for all Sven backend implementations.

3 Related Work

Sven sits at the intersection of several active areas of research: natural gradient methods, second-order and quasi-Newton optimizers, and Jacobian-based pseudoinverse methods. We survey the most closely related work, highlighting how Sven differs from and extends prior approaches.

Natural gradient methods. Natural gradient descent (Amari, 1998) replaces the Euclidean update with one computed using the FIM, yielding parameter-invariant updates that are theoretically optimal in the information-geometric sense. Martens (2020) shows that the FIM is equivalent to the generalized Gauss-Newton (GGN) matrix for exponential family losses. As we showed in Section 2.3, Sven recovers natural gradient descent exactly in the underparametrized limit, and generalizes it to the overparametrized regime by using the pseudoinverse of the full loss Jacobian in place of the singular natural gradient metric.

K-FAC (Martens & Grosse, 2015) approximates the FIM as a block-diagonal matrix with Kronecker-product blocks, yielding efficient natural gradient updates at roughly 2–3$\times$ the cost of SGD. Unlike K-FAC, which imposes a Kronecker structure within each layer and accumulates statistics across iterations, Sven computes a per-step truncated SVD of the loss Jacobian across all parameters simultaneously. K-FAC does not resolve the overparametrized singularity that motivates Sven: its Fisher is still a $P \times P$ block-Kronecker object, rank-deficient when P exceeds the number of conditions, and exists only through damping, which gives a biased rather than a minimum-norm update.

Second-order and quasi-Newton optimizers. Adam (Kingma & Ba, 2015; Loshchilov & Hutter, 2019) remains the dominant optimizer for deep learning, maintaining diagonal estimates of first and second gradient moments as a cheap preconditioner; Sven includes off-diagonal structure that Adam ignores entirely. Shampoo (Gupta et al., 2018) and its descendant SOAP (Vyas et al., 2025) maintain Kronecker-factored preconditioning matrices, accumulating second moment statistics over time rather than computing a per-step spectral decomposition as Sven does. Muon (Jordan et al., 2024) orthogonalizes the gradient matrix via its polar factorization UV^T, equivalently replacing all singular values with 1; in contrast, Sven *inverts* the singular values, applying more weight to

directions with smaller singular values as a curvature-correcting strategy. The Levenberg-Marquardt algorithm (Marquardt, 1963; Hagan & Menhaj, 1994) interpolates between gradient descent and Gauss-Newton via a damping parameter, shrinking all singular values uniformly; Sven's hard truncation of the top-k directions provides an alternative regularization strategy, though we do not compare the two empirically.

Jacobian pseudoinverse and Gauss-Newton methods. The most closely related work to Sven is Exact Gauss-Newton (EGN) (Korbit et al., 2025), which also factorizes a $Bc \times Bc$ matrix[4] rather than a $P \times P$ one when the batch size $B \ll P$, doing so via the Woodbury identity. Sven differs in operating on the per-sample *loss* Jacobian rather than the output Jacobian J_f, and in using a truncated SVD of tunable rank k rather than solving the system exactly. Jacobian Descent (JD) (Quinton & Rey, 2024) shares the same starting point as Sven, but aggregates Jacobian rows with a projection-based operator whereas Sven takes the pseudoinverse. The natural-gradient framing is also absent from JD. Half-Inverse Gradients (HIG) (Schnell et al., 2022) is mechanistically the closest precursor to Sven: it computes the SVD of a Jacobian and raises singular values to a fractional power κ, with $\kappa = -1$ recovering the full pseudoinverse. Sven differs in using hard truncation rather than a continuous fractional power, targeting general neural network training rather than physics-in-the-loop optimization, and operating on the per-sample loss Jacobian with an explicit natural gradient interpretation. HIG is also much more computationally expensive than Sven, as it uses the *model* Jacobian which scales with the number of outputs.[5]

To summarize the key distinction: while prior work has taken the SVD over other summation indices of the loss, we are unique in decomposing over the data index and drawing the resulting connection to natural gradient methods.

4 Experiments

We demonstrate Sven on relatively small-scale problems where we can run detailed hyperparameter scans to understand its convergence properties. Anonymized code for reproducing all experiments is linked in App. A. Our main results, summarized in Fig. 1, involve three datasets:

1. **Random polynomial**: a random polynomial over $\mathbb{R}^6$ built from all monomials of total degree ≤ 4, each coefficient variance-normalized so no degree dominates. Inputs are IID Gaussian and we evaluate on three independently drawn polynomial targets to account for variation in learnability.
2. **MNIST** (LeCun et al., 1998) under a *label regression* loss $\|f_\theta(\mathbf{x}_i) - \mathbf{y}_i\|^2$ against the one-hot label, which puts classification on the same footing as regression; cross-entropy on the same model and splits is in Appendix K.
3. **nanoGPT** (Karpathy, 2022): a four-layer decoder-only transformer on a character-level Shakespeare dataset.

Appendix D.1 provides full details for each. We also analyze a toy 1D regression problem (Appendix F), ResNet18 on CIFAR-10 (Appendix L), and a limited pilot LLM training run using GPT-2-small (Radford et al., 2019) on FineWebText-edu (Penedo et al., 2024) (Appendix N). We omit CIFAR-10 from the main results because its BatchNorm prevents using the efficient Sven implementation, and omit GPT-2 because resource constraints prevented us from performing hyperparameter optimization or running multiple seeds.

We compare Sven against 14 common baselines: SGD, SGD with momentum, RMSprop (Hinton et al., 2012), Adam (Kingma & Ba, 2015), AdamW (Loshchilov & Hutter, 2019), Polyak-step SGD (Polyak, 1969; Hazan & Kakade, 2019), stochastic L-BFGS (Liu & Nocedal, 1989), SOAP (Vyas et al., 2025), Shampoo Gupta et al. (2018), K-FAC (Martens & Grosse, 2015), Muon (Jordan et al., 2024) with and without weight decay, HIG (Schnell et al., 2022), and JD (Quinton & Rey, 2024). On MNIST, K-FAC was too unstable to optimize and SOAP failed to converge; both are omitted. Muon optimizes embedding layers and output heads with AdamW, using the orthogonalized Muon update for all

[4] c is the model output dimension

[5] For example, HIG takes $\sim$10x longer than Sven in our MNIST experiments, and is totally intractable for a transformer-scale model.

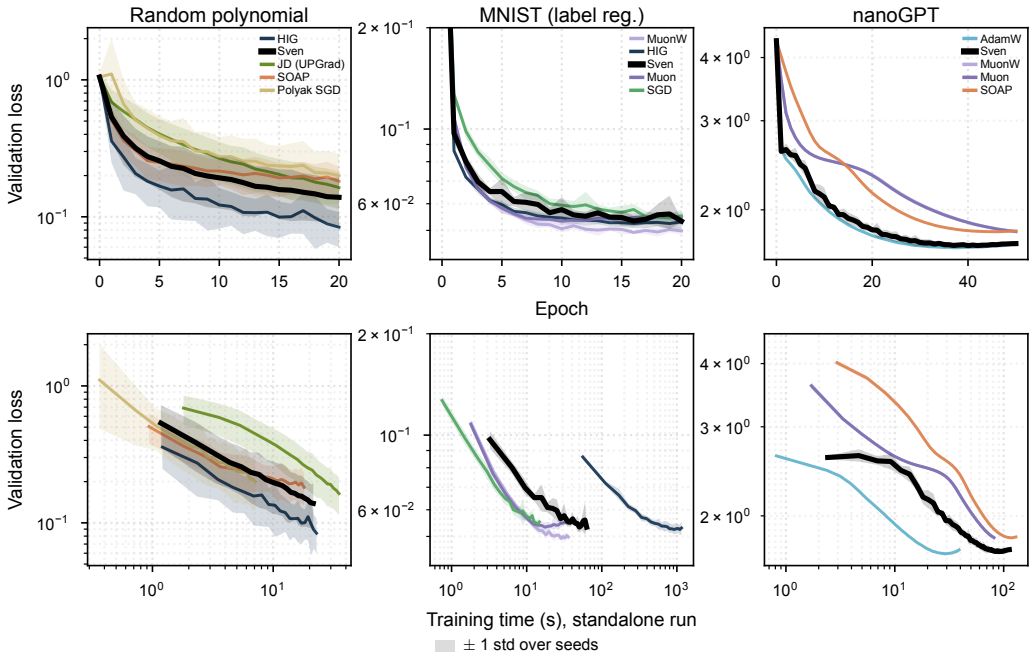

Figure 1: Validation loss against epoch (top) and against wall time (bottom) for Sven and four best-performing baselines on the three headline tasks: random polynomial regression (left), MNIST classification with label regression (middle), and tiny-Shakespeare language modeling with nanoGPT (right). Curves are means over 5 seeds, with optimizer parameter settings chosen according to lowest mean validation loss at the end of training.

other linear and convolutional weights (with kernels flattened), and the step RMS-matched to an AdamW step (Liu et al., 2025). Due to resource constraints, we optimize nanoGPT with only Sven, AdamW, Muon/MuonW[6], and SOAP. Polynomial regression and MNIST are trained using MLPs with three hidden layers and a larger hidden width for MNIST ($P = 673$ and 27562, respectively), while nanoGPT uses a four-layer decoder-only transformer ($P \approx 826$k). Each baseline is tuned over a learning rate grid, while Sven is tuned over a $(\eta, k, \texttt{rtol})$ grid. For comparison, we select the best-performing hyperparameter configuration from the grid for each optimizer, where performance is measured using final validation loss. Results are averaged over five independent model seeds. See Appendix D for full experimental details.

4.1 Results

Figure 1 summarizes the main results, showing validation loss over the course of training (top) and as a function of training wall-time (bottom) for Sven and the four best-performing baselines (Fig. 5 in Appendix F shows all optimizers). Sven matches or beats most baselines within seed uncertainty. HIG outperforms Sven on the polynomial task but suffers when scaled even modestly to the MNIST task, where it costs a factor of ten more wall-time. Sven matches AdamW on nanoGPT, showing that it can perform well on cross-entropy targets as well, and costs similar wall-time to Muon and SOAP.

Figure 2 provides a more detailed account of Sven's performance against all baselines, and includes toy 1D regression and MNIST cross-entropy results. In each panel we show *paired differences* relative to Sven: each optimizer O is trained on N independent model seeds using its best hyperparameter configuration, and for each seed $i = 1, \ldots, N$ we compute test set loss $L^{\text{test}}_{i,O}$ and the difference $\Delta_{i,O} = L^{\text{test}}_{i,\text{Sven}} - L^{\text{test}}_{i,O}$. Fig. 2 plots the mean paired difference and 95% confidence intervals over N seeds. Sven beats a large fraction of the baselines in most cases, though it is never unambiguously the best. Interestingly, it more consistently outperforms baselines on MNIST cross-entropy than label

[6]We denote Muon + weight decay as MuonW; Sven is un-regularized, so we include un-regularized Adam and Muon in the baseline pool

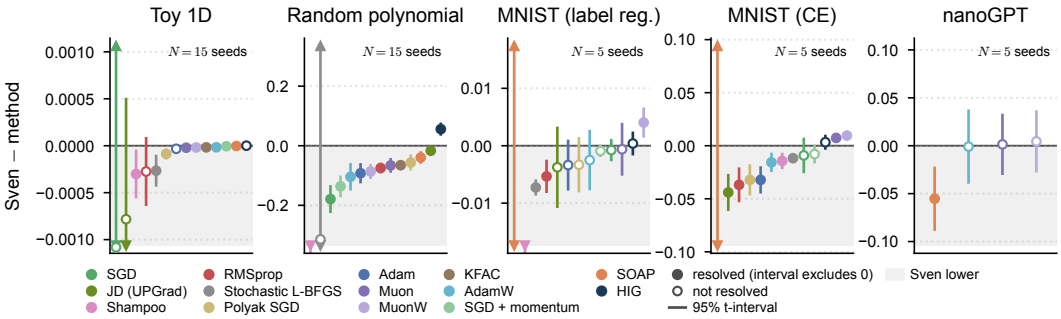

Figure 2: Mean paired differences in test set loss relative to Sven for all baseline optimizers on the synthetic regression, MNIST, and nanoGPT tasks. Error bars are Student-t 95% intervals on the mean difference over the N seed pairs (indicated in each plot); a filled marker is an interval that excludes zero. An interval wider than the panel is clipped to an arrow.

regression. These results also use losses on an independent test set, which explains the larger error bars and changed order on nanoGPT relative to Fig. 1.[7]

4.2 Analyzing Singular Value Spectra

Sven's optimization behavior is closely related to the singular value spectrum of a batch's loss Jacobian M, and in this section we analyze that structure to better understand what Sven is doing. First, we note that k and `rtol` are not truly independent parameters depending on the shape of the SV spectrum: if $\sigma_i/\sigma_1 > \texttt{rtol}$ for all B SVs of M, then k fully controls truncation; likewise, if k is set high and the spectrum falls rapidly, `rtol` controls the truncation threshold which will vary batch-to-batch and generally be $\leq k$. In practice we find optimal $(k, \texttt{rtol})$ settings via a hyperparameter scan, but portions of this grid may be effectively identical depending on the shape of the SV spectrum for a particular task.

In Fig. 3 (a,b) we plot full SV spectra at the beginning and end of training for the toy 1D, polynomial, MNIST, and nanoGPT tasks. Spectra are recorded from runs with the best-performing Sven hyperparameters as measured in the previous section. On each panel we indicate each task's selected `rtol` and k with horizontal and vertical dashed lines, respectively. The different shapes of each problem are immediately clear. The 1D regression task is very hierarchical, with much of its spectrum lying below `rtol` and below the float32 round-off floor, below which SVs are not meaningful.[8] The MNIST and nanoGPT spectra are quite the opposite, lying above their `rtol` cutoff (with the exception of MNIST-CE). The spectra also evolve during training, some becoming steeper and others shallower without a clear trend. These curves also make it clear that `rtol` seems more influential than k. MNIST-LR, toy 1D, and nanoGPT all select $k = B$, leaving `rtol` to control truncation, which it only does in practice for toy 1D. For MNIST-CE and polynomial, where $k = B/2$ is chosen, the spectrum shapes suggest `rtol` still controls truncation.

Beyond inspecting the spectra, we can directly probe how much of the per-batch residual the Sven update accounts for. Denoting $R = (\mathcal{R}_1, \ldots, \mathcal{R}_B)^T$ as the vector of batch residuals (i.e. per-sample losses for $\kappa = 2$) and the Jacobian as $M = U\Sigma V^T$, the untruncated Sven update amounts to $\delta\theta \propto -M^+R = -V\Sigma^{-1}U^TR$. Letting $\tilde{U} = (\mathbf{u}_1, \ldots, \mathbf{u}_k)$ denote the *truncated* set of singular vectors, we can measure how much of the residual vector is accounted for by the truncated update – i.e. how much of R lives in $\operatorname{span}\tilde{U}$. Fig. 3 (c) plots $1 - ||\tilde{U}^TR||^2/||R||^2$ – the fraction of R *lost* by truncating – over the course of training. MNIST-LR and nanoGPT use an un-truncated spectrum throughout training and sit near zero, while for MNIST-CE it appears beneficial to discard $\sim$10% of the residual-containing directions during optimization. The toy 1D fraction evolves over training, starting near zero and ending around 10%. Looking at the spectra, it is clear that the vast majority of

[7] Given the small dataset size, nanoGPT likely overfits.

[8] Sven takes $\sigma_i = \sqrt{\lambda_i}$ from the Gram matrix $G = MM^\top$, whose entries inherit the float32 rounding of M as an absolute error $\sim\epsilon_{32}\sigma_{\max}^2$ on every eigenvalue. SVs are only resolvable above $\sqrt{\epsilon_{32}}\,\sigma_{\max} \approx 3\times10^{-4}\sigma_{\max}$ (forming $MM^\top$ squares the condition number).

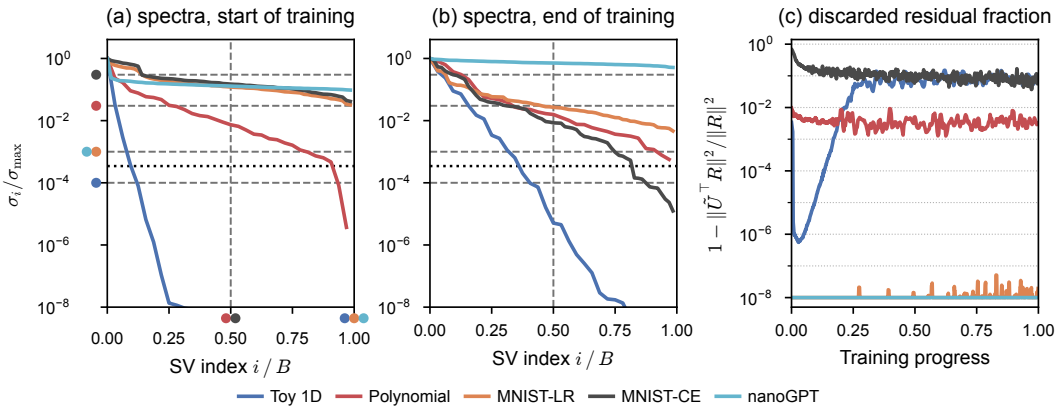

Figure 3: Normalized singular value spectra at the beginning (a) and end (b) of training. Horizontal lines mark the selected `rtol` and vertical lines the selected k/B, with colored dots indicating the corresponding datasets. The dotted black line is the float32 round-off floor. (c) The fraction of the batch residual the truncation discards over the course of training.

the residual lives in a small handful of directions early in training (steep spectrum) but expands to significantly more by the end of training, where `rtol` cuts off half the spectrum and discards 10% of the residual.

These results suggest that Sven shapes and interacts with a rich and varying singular value landscape over the course of training. Studying this interplay in closer detail is a promising avenue for future work. The results of Fig. 3 also suggest a potential hyperparameter prescription for Sven that avoids k and `rtol`: selecting a truncation threshold dynamically at each step based on a desired target for the "fraction of explained residual" $||\tilde{U}^T R||^2/||R||^2$. We leave this for future work.

5 Conclusion

In this paper we introduced a new optimizer, Sven, which can be viewed as a natural-gradient-inspired method that is tractable in the overparametrized regime. While natural gradient methods are prohibitively expensive in practice, a Sven step with the efficient backend costs only a constant number of forward and backward passes and a $B \times B$ eigendecomposition. Our experiments show that Sven is competitive with mainstream baselines, is effective on both regression and classification problems, and is both viable and competitive on a small-scale transformer language modeling task. We view Sven not as a replacement for existing optimization machinery, but as a complementary technique to be added to the practitioner's toolkit. Modern neural network training already combines a rich collection of methods (weight decay, gradient clipping, learning rate scheduling, momentum, and adaptive scaling among them) each addressing a distinct aspect of optimization. Sven adds to this toolkit a principled mechanism for exploiting the singular value structure of the loss Jacobian, and can be composed with these existing techniques.

Limitations and future work. Sven is not easily transferable to architectures with BatchNorm or tied weights, which must use the less memory/compute-efficient backend. Appendix L reports poor performance on CIFAR-10/ResNet18, though the reasons why are not understood. Appendix N also shows poor performance in a preliminary scaling study on GPT-2-small, though Sven was not tuned *at all* due to resource limitations and this result should be considered preliminary at best. Immediate future directions are scaling to LLMs and finding solutions that enable an efficient backend for more architectures. Sven is also promising for scientific computing, where losses might decompose over collocation points or boundary conditions whose contributions carry meaning, as in the theoretical physics application (Anonymous, 2026).

AI USE STATEMENT

In preparing this work we used LLM assistants for a wide variety of tasks (Anthropic's Claude Code and OpenAI's Codex). The core idea behind the paper came from the authors, as did the design of numerical experiments and the interpretation/commentary on their results. An early version of the Sven optimizer codebase containing its core functionality was written fully by the authors. Coding agents assisted with the improved memory-efficient backend, adding logging functionality, and optimizing the authors' implementations. They also helped clean up the codebase and convert it into an installable package form. An early version of the framework for running numerical experiments and creating figures was similarly written fully by the authors, and subsequently refined and expanded using coding agents.

We did *not* use generative AI to write any of the main text. LLM assistants helped in writing skeleton versions of the technical appendices, including tables and text summarizing the details of our experimental setup (hyperparameter grids, model configurations, datasets and splits, etc.). We also consulted LLMs for feedback on the design and scope of our numerical experiments, and incorporated their feedback when it was judged to be sound (e.g. expanding hyperparameter grids). We also used LLMs to aid with literature review.

Every piece of LLM-generated appendix text was thoroughly checked for correctness and heavily edited by the authors before presentation. We take full responsibility for all figures, text, code, and experimental artifacts associated with this work, regardless of the level of LLM involvement.

REPRODUCIBILITY STATEMENT

The results presented in this paper are reproducible from the anonymized code linked in Appendix A, up to unavoidable variations due to floating-point error and/or randomized algorithms (where applicable). The appendices include full experimental details, and all results are derived from runs over a fixed set of model, data, split, and data loader seeds written in configuration files in the experimental codebase. Results related to memory and/or time profiling will vary based on the hardware used to run those experiments.

REFERENCES

P.A. Absil, R. Mahony, and R. Sepulchre. *Optimization Algorithms on Matrix Manifolds*. Princeton University Press, 2009. ISBN 9781400830244. URL https://books.google.com/books?id=NSQGQeLN3NcC.

Shun-ichi Amari. Natural gradient works efficiently in learning. *Neural Computation*, 10(2):251–276, 1998. doi: 10.1162/089976698300017746.

Shun-ichi Amari. *Information geometry and its applications*. Springer, 2016.

Anonymous. Anonymized title. *Citation anonymized for double-blind review*, 2026.

Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, C. K. Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Shunting Zhang, Michael Suo, Phil Tillet, Xu Zhao, Eikan Wang, Keren Zhou, Richard Zou, Xiaodong Wang, Ajit Mathews, William Wen, Gregory Chanan, Peng Wu, and Soumith Chintala. PyTorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '24, pp. 929–947, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400703850. doi: 10.1145/3620665.3640366. URL https://doi.org/10.1145/3620665.3640366.

Alberto Bernacchia, Máté Lengyel, and Guillaume Hennequin. Exact natural gradient in deep linear networks and its application to the nonlinear case. *Advances in Neural Information Processing Systems*, 31, 2018.

Nikolai Nikolaevich Chentsov. *Statistical decision rules and optimal inference*. American Mathematical Society, 1982.

Thomas M Cover and Joy A Thomas. *Elements of information theory (wiley series in telecommunications and signal processing)*. Wiley-Interscience, 2006.

Yanhong Fei, Yingjie Liu, Chentao Jia, Zhengyu Li, Xian Wei, and Mingsong Chen. A survey of geometric optimization for deep learning: from euclidean space to riemannian manifold. *ACM Computing Surveys*, 57(5):1–37, 2025.

Ian Goodfellow. Efficient per-example gradient computations. *arXiv preprint arXiv:1510.01799*, 2015.

Vineet Gupta, Tomer Koren, and Yoram Singer. Shampoo: Preconditioned stochastic tensor optimization. In Jennifer Dy and Andreas Krause (eds.), *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pp. 1842–1850. PMLR, 2018. URL https://proceedings.mlr.press/v80/gupta18a.html.

Martin T. Hagan and Mohammad B. Menhaj. Training feedforward networks with the Marquardt algorithm. *IEEE Transactions on Neural Networks*, 5(6):989–993, 1994. doi: 10.1109/72.329697.

Nathan Halko, Per-Gunnar Martinsson, and Joel A Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM Review*, 53 (2):217–288, 2011.

Thomas R Harvey. The optimiser hidden in plain sight: Training with the loss landscape's induced metric. *arXiv preprint arXiv:2509.03594*, 2025.

Elad Hazan and Sham Kakade. Revisiting the polyak step size. *arXiv preprint arXiv:1905.00313*, 2019.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*, 2016.

Joel Hestness, Sharan Narang, Newsha Ardalani, Gregory Diamos, Heewoo Jun, Hassan Kianinejad, Md Mostofa Ali Patwary, Yang Yang, and Yanqi Zhou. Deep learning scaling is predictable, empirically. *arXiv preprint arXiv:1712.00409*, 2017.

Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky. Neural networks for machine learning, 2012. Coursera, Lecture 6.5.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pp. 448–456. pmlr, 2015.

Keller Jordan, Yuchen Jin, Vlado Boza, Jiacheng You, Franz Cesista, Laker Newhouse, and Jeremy Bernstein. Muon: An optimizer for hidden layers in neural networks, 2024. URL https://kellerjordan.github.io/posts/muon/. Blog post.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

Andrej Karpathy. nanogpt. https://github.com/karpathy/nanoGPT, 2022. Includes the character-level tiny-shakespeare corpus.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations (ICLR)*, 2015. URL https://arxiv.org/abs/1412.6980.

Mikalai Korbit, Adeyemi D. Adeoye, Alberto Bemporad, and Mario Zanon. Exact Gauss-Newton optimization for training deep neural networks. *Neurocomputing*, 658:131738, 2025. ISSN 0925-2312. doi: https://doi.org/10.1016/j.neucom.2025.131738. URL https://www.sciencedirect.com/science/article/pii/S0925231225024105.

Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. Technical report, University of Toronto, Toronto, Ontario, 2009. URL https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf.

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Dong C Liu and Jorge Nocedal. On the limited memory bfgs method for large scale optimization. *Mathematical Programming*, 45(1):503–528, 1989.

Jingyuan Liu, Jianlin Su, Xingcheng Yao, Zhejun Jiang, Guokun Lai, Yulun Du, Yidao Qin, Weixin Xu, Enzhe Lu, Junjie Yan, et al. Muon is Scalable for LLM Training. *arXiv preprint arXiv:2502.16982*, 2025.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=Bkg6RiCqY7.

Donald W. Marquardt. An algorithm for least-squares estimation of nonlinear parameters. *Journal of the Society for Industrial and Applied Mathematics*, 11(2):431–441, 1963. doi: 10.1137/0111030.

James Martens. New insights and perspectives on the natural gradient method. *Journal of Machine Learning Research*, 21(146):1–76, 2020. URL http://jmlr.org/papers/v21/17-678.html.

James Martens and Roger Grosse. Optimizing neural networks with Kronecker-factored approximate curvature. In Francis Bach and David Blei (eds.), *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pp. 2408–2417, Lille, France, 2015. PMLR. URL https://proceedings.mlr.press/v37/martens15.html.

Eliakim H. Moore. On the reciprocal of the general algebraic matrix. *Bulletin of the American Mathematical Society*, 26:394–395, 1920.

Guilherme Penedo, Hynek Kydlíček, Loubna Ben allal, Anton Lozhkov, Margaret Mitchell, Colin Raffel, Leandro Von Werra, and Thomas Wolf. The fineweb datasets: Decanting the web for the finest text data at scale. In *The Thirty-eighth Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2024. URL https://openreview.net/forum?id=n6SCkn2QaG.

Roger Penrose. A generalized inverse for matrices. In *Mathematical Proceedings of the Cambridge Philosophical Society*, volume 51, pp. 406–413. Cambridge University Press, 1955.

Boris Teodorovich Polyak. Minimization of unsmooth functionals. *USSR Computational Mathematics and Mathematical Physics*, 9(3):14–29, 1969.

Pierre Quinton and Valérian Rey. Jacobian descent for multi-objective optimization, 2024. URL https://arxiv.org/abs/2406.16232.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. Technical report, OpenAI, 2019. URL https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf.

Patrick Schnell, Philipp Holl, and Nils Thuerey. Half-inverse gradients for physical deep learning. In *International Conference on Learning Representations (ICLR)*, 2022. URL https://arxiv.org/abs/2203.10131.

Rajesh Shrestha. Natural gradient methods: Perspectives, efficient-scalable approximations, and analysis. *arXiv preprint arXiv:2303.05473*, 2023.

TorchVision maintainers and contributors. TorchVision: PyTorch's Computer Vision library, November 2016. URL https://github.com/pytorch/vision.

Nikhil Vyas, Depen Morwani, Rosie Zhao, Itai Shapira, David Brandfonbrener, Lucas Janson, and Sham Kakade. SOAP: Improving and stabilizing Shampoo using Adam. In *International Conference on Learning Representations (ICLR)*, 2025. URL https://arxiv.org/abs/2409.11321.

A Code Availability

We release anonymized code for the Sven optimizer package at https://anonymous.4open.science/r/sven-E82B/, and for the reproducible experiments presented in this paper at https://anonymous.4open.science/r/sven-experiments-82E8/.

B Natural Gradients and Functional Gradient Descent

As mentioned in Section 2.3, natural gradient methods offer a principled and powerful approach to preconditioning gradient-based optimization (Amari, 1998; Shrestha, 2023). Furthermore, natural gradient methods can be viewed as an approximation to functional gradient descent. In this appendix, we present key elements of natural gradient methods from this functional perspective for the interested reader. Throughout this section, we assume the object being optimized is a function, as opposed to a probability distribution as would be the case in classification problems. Analogous arguments can be carried out for probability distributions, where one would begin with the Kullback–Leibler divergence between distributions, rather than the L_2 distance between functions.[9] This appendix is intended as a pedagogical introduction; the arguments presented are not formal proofs but instead focus on intuition rather than rigor.[10] The approach taken here also offers a somewhat different perspective than the majority of the literature on this subject.

Our approach is as follows: we define a metric on the abstract space of functions, then pull back that metric to the submanifold defined by our function parameterization. This submanifold can be thought of as the loss landscape. We can then use this metric as the preconditioner for gradient descent. Consider two functions $f(x)$ and $g(x)$, and we define the distance between them as being the L_2 distance:

$$d^2_{L_2}[f, g] = \int d\mu(x) \big(f(x) - g(x)\big)^2, \tag{14}$$

where $d\mu(x)$ is the measure induced by the data. In other words, this should be understood as an integral over the data manifold. Now if we consider the distance between $f(x)$ and $f(x) + \delta f(x)$, where $\delta f(x)$ is some small function variation, we can define a line element on the space of functions:

$$|df|^2 = d^2_{L_2}[f, f + \delta f] = \int d\mu(x) \big(\delta f(x)\big)^2. \tag{15}$$

We now impose that we are using some parametrized family of functions $f(x) \approx f_\theta(x)$, where $\{\theta^i : i = 1 \dots N\}$ are our parameters and the space they parametrize is the loss landscape $\mathcal{L}$. Upon

[9] By Chentsov's theorem, any f-divergence yields the same metric up to a scalar, making the FIM the unique reparametrization-invariant metric on the space of probability distributions (Amari, 2016; Chentsov, 1982; Cover & Thomas, 2006).

[10] In particular, we will frequently transform between different parametrizations, without worrying about the expressivity of those parametrizations. Such considerations become less important as the number of parameters increases, and would only complicate this discussion.

restricting to the loss landscape, our functional variations pull back to

$$\delta f(x)|_{\mathcal{L}} = \sum_i \frac{\partial f_\theta(x)}{\partial \theta^i} d\theta^i, \tag{16}$$

and the metric on the abstract space of functions pulls back to

$$g_{ij} = \int d\mu(x) \frac{\partial f_\theta(x)}{\partial \theta^i} \frac{\partial f_\theta(x)}{\partial \theta^j}, \tag{17}$$

which, once the integral is evaluated via Monte Carlo sampling, yields the natural gradient metric presented in Section 2.3.

While the above demonstrates that the natural gradient method provides a principled metric on the loss landscape, we have not yet shown why it is useful. To do so, we must invert the natural gradient metric and consider the resulting gradient flow. In other words, we seek solutions to

$$\frac{d\theta^i(t)}{dt} = -\sum_j g^{ij} \frac{\partial L(\theta)}{\partial \theta^j}, \tag{18}$$

where g^{ij} is the inverse of g_{ij}. To make this problem tractable, we first consider whether it is possible to find a parametrization where the natural gradient metric reduces to the Euclidean metric. Using the shorthand

$$f_i(x) = \frac{\partial f_\theta(x)}{\partial \theta^i}, \tag{19}$$

setting $g_{ij} = \delta_{ij}$ implies that the functions $f_i(x)$ are orthonormal. In other words, if we express the function as a linear combination of orthonormal functions on the data manifold, then gradient descent on these coefficients is equivalent to natural gradient descent with an arbitrary parametrization (up to the expressivity concerns mentioned in the earlier footnote).

One may worry whether a set of orthonormal functions exists on an arbitrary data manifold. While the necessary conditions for such a set to exist are somewhat complicated, one sufficient condition is that the manifold be compact. Given a finite dataset, there are clearly infinitely many possible data manifolds from which it could have been sampled, infinitely many of which are compact. Therefore, while constructing such functions may be complicated, it is reasonable to assume their existence for our analysis. As such, the natural gradient method is actually a flat metric on the data manifold.

Let us consider a simple regression problem with the loss function

$$L = \int d\mu(x) \big(f_\theta(x) - f(x)\big)^2, \tag{20}$$

where $f(x)$ is some target function determined by our data. We can, at least formally, expand both $f_\theta(x)$ and $f(x)$ in our orthonormal basis

$$f_\theta(x) = \sum_i \alpha^i(\theta) f_i(x) \quad \& \quad f(x) = \sum_i \beta^i f_i(x). \tag{21}$$

We want to study gradient descent with respect to θ^i using the natural gradient metric. As argued earlier, this is equivalent to gradient descent with respect to α^i:

$$\frac{d\theta^i(t)}{dt} = -\sum_j g^{ij} \frac{\partial L}{\partial \theta^j} \quad \equiv \quad \frac{d\alpha^i(t)}{dt} = -\frac{\partial L}{\partial \alpha^i}, \tag{22}$$

which dramatically simplifies our analysis. In fact, we can analytically solve this differential equation. The solution is given by

$$\alpha^i(t) = \big(\alpha^i(0) - \beta^i\big) e^{-2t} + \beta^i, \tag{23}$$

which implies that the loss decays as

$$L \propto e^{-4t}. \tag{24}$$

Therefore, if we were given the natural gradient metric without computational cost (or equivalently, a set of orthonormal functions on the data manifold), the loss would decay exponentially rather than exhibiting the power-law behavior predicted by scaling laws (Hestness et al., 2017; Kaplan et al., 2020).

C GRADIENT DESCENT AND FUNCTIONAL ANALYSIS

In this appendix, we would like to discuss metric-driven gradient descent from a different viewpoint, using the language of functional analysis and differential geometry. This will allow us to put some of the statements in the previous section on a more rigorous footing. We start with a neural network $f_\theta : \mathbb{R}^d \to \mathbb{R}$ (real-valued for simplicity but easily generalized) with parameters $\theta = (\theta^i)$, where $i = 1 \dots N$, and a loss function $L : \mathbb{R}^N \to \mathbb{R}$. The standard gradient descent equation,

$$\delta\theta^i = -\eta \, \frac{\partial L(\theta)}{\partial \theta^i} \ , \tag{25}$$

as written (where η is the learning rate) is clearly somewhat unsavory since the LHS corresponds to a vector field and the RHS to a differential form (as indicated by the different positions of the indices). The obvious way to rectify this is to introduce a structure on parameter space which induces an identification of tangent and co-tangent bundle, such as a Riemannian or symplectic structure. Opting for the former (although it might be interesting to explore the latter option) we introduce a metric $g = (g_{ij})$ with inverse $g^{-1} = (g^{ij})$ on parameter space and modify equation 25 to

$$\delta\theta^i = -\eta \sum_j g^{ij} \, \frac{\partial L(\theta)}{\partial \theta^j} \ . \tag{26}$$

Here is a way to choose the metric g. We introduce a Hilbert space $\mathcal{H} = \mathcal{L}^2_w(\mathbb{R}^d)$, an $\mathcal{L}^2$ space with weight function $w : \mathbb{R}^d \to \mathbb{R}^{>0}$, standard scalar product

$$\langle f, h \rangle := \int_{\mathbb{R}^d} d^d x \, w(x) f(x) h(x) \ , \tag{27}$$

and associated norm $|| \cdot ||$. We choose the weight function w such that $f_\theta \in \mathcal{H}$ for all θ. (This is related to footnote 10 on expressivity of different function classes in the previous appendix.) In this case, the map $\theta \mapsto f_\theta$ defines an N-dimensional manifold $\mathcal{L}$ in $\mathcal{H}$, which is the same manifold we identified with the loss-landscape in the previous appendix. Relative to an orthonormal basis (h_I) on $\mathcal{H}$ we can expand the neural network as well as a function $f \in \mathcal{H}$, representing the "data", as

$$f_\theta \simeq \sum_{I \in \mathcal{I}} a_I(\theta) h_I \ , \qquad\qquad a_I(\theta) = \langle h_I, f_\theta \rangle, \tag{28}$$

$$f \simeq \sum_{I \in \mathcal{I}} \alpha_I h_I \ , \qquad\qquad \alpha_I = \langle h_I, f \rangle \ , \tag{29}$$

where we have truncated the sums to a finite index set $\mathcal{I}$, as would be required for a computational realization. The associated linear subspace $\mathcal{W} = \mathrm{span}(h_I)_{I \in \mathcal{I}} \subset \mathcal{H}$ should be "large enough" (and it would be interesting to quantify this requirement) to approximate the manifold $\mathcal{L}$ sufficiently well. We note that the expansion in equation 29 provides the optimal approximation to f within $\mathcal{W}$ in the sense that $||f - \sum_{I \in \mathcal{I}} \alpha_I h_I||$ is minimal for the values of α_I as in equation 29. In other words, in the Hilbert space picture there is a unique and well-defined optimal approximation for the 'data function' f, provided by equation 29.

Let us now discuss gradient descent and, for simplicity, work with a mean square loss, which can be written as $L(\theta) = \frac{1}{2}||f_\theta - f||^2 \simeq \frac{1}{2} \sum_{I \in \mathcal{I}} (a_I(\theta) - \alpha_I)^2$. Applying the metric gradient descent in equation 26 to this loss function gives

$$\delta\theta^i = -\eta \sum_j g^{ij} \left\langle \frac{\partial f}{\partial \theta^j}, f_\theta - f \right\rangle \simeq -\eta \sum_j g^{ij} \sum_{I \in \mathcal{I}} \frac{\partial a_I}{\partial \theta^j}(\theta)(a_I(\theta) - \alpha_I) \ . \tag{30}$$

How does the training trajectory described by equation 30 translate to the Hilbert space expansion coefficients a_I? A short calculation shows that

$$\delta a^I = -\eta \sum_J G^{IJ} \, (a_J - \alpha_J) \ , \qquad G^{-1} = J g^{-1} J^T \ , \tag{31}$$

where $J = \frac{\partial a}{\partial \theta}$ is the Jacobi matrix of the coordinate change. There is a canonical choice of metric on $\mathcal{H}$, namely $G_{IJ} = \delta_{IJ}$, the metric induced by the scalar product relative to the orthonormal basis. For this choice of G, we obtain the equation

$$J g^{-1} J^T = \mathbb{1} \ , \tag{32}$$

for g and, defining $\epsilon_I = a_I - \alpha_I$, the continuous version of equation 31 turns into

$$\frac{d\epsilon_I}{dt}(t) = -\eta\,\epsilon_I(t) \quad \Rightarrow \quad \epsilon_I(t) \sim e^{-\eta t}\ . \tag{33}$$

Evidently, this means the unique best approximation for f in $\mathcal{W} \subset \mathcal{H}$, represented by the coefficients α_I, is approached at an exponential rate. By choosing the natural metric on the Hilbert space we have obtained efficient exponentially fast training. To translate this training trajectory to the original neural network parameters we have to consider equation 32. A metric g which solves this equation does not necessarily exist. However, we can introduce the Moore-Penrose pseudoinverse $J^+ \equiv (J^T J)^{-1} J^T$ of J which satisfies $J^+ J = \mathbb{1}$. The inverse in this equation is well-defined as long as the Jacobi matrix J has maximal rank. This is (generically) the case when the number $|\mathcal{I}| = \dim(\mathcal{W})$ of Hilbert space modes is taken to be greater than or equal to the number N of neural network parameters. Multiplying equation 32 with J^+ and $(J^+)^T$ from the left and the right, respectively, leads to

$$g_{ij}(\theta) = (J^T J)_{ij} = \sum_{I\in\mathcal{I}} \frac{\partial a_I}{\partial\theta^i}(\theta)\frac{\partial a_I}{\partial\theta^j}(\theta) = \left\langle \frac{\partial f_\theta}{\partial\theta^i}, \frac{\partial f_\theta}{\partial\theta^j} \right\rangle\ . \tag{34}$$

As mentioned, this metric does not necessarily solve equation 32 but it provides the best approximation to a solution in the sense that the (mean square of the) difference between the LHS and RHS of equation 32 is minimal. The metric g in equation 34 has a beautiful geometrical interpretation: it is the pull-back of the natural Hilbert space metric induced by the scalar product $\langle\cdot,\cdot\rangle$ to the loss landscape $\mathcal{L} \subset \mathcal{H}$. When used in the metric gradient descent equation 26 it is expected to lead to exponentially efficient training. For simple neural networks f_θ (such as perceptrons) and a choice of Hilbert space g can be computed analytically by inserting f_θ into the RHS in equation 34 and carrying out the scalar product. For more complicated neural networks this is difficult to do and a data-driven approach may be more appropriate.

We introduce data $\mathcal{D} = (x_\beta, y_\beta)_{\beta=1,\ldots,|\mathcal{D}|}$, where $x_\beta \in \mathbb{R}^d$ and $y_\beta \in \mathbb{R}$, batches $B \subset \{1,\ldots,|\mathcal{D}|\}$ and assume that the x_β are distributed according to a data measure ρ. For this data and $\alpha, \beta \in B$ we define

$$M_{\beta i} = \frac{\partial f_\theta}{\partial\theta^i}(x_\beta)\ , \qquad \mathcal{R}_\beta = f_\theta(x_\beta) - y_\beta\ , \qquad \mu_{\alpha\beta} = \delta_{\alpha\beta}\frac{w(x_\beta)}{\rho(x_\beta)}\ , \tag{35}$$

that is, the $|B| \times n$ matrix M which represents the Jacobi matrix evaluated on the data, the neural network residual $\mathcal{R}$ and the relative measure μ. With these quantities we can approximate

$$\left(\left\langle \frac{\partial f_\theta}{\partial\theta^i}, f_\theta - f \right\rangle\right) \simeq M^T\mu\mathcal{R}\ , \qquad g \simeq M^T\mu M \tag{36}$$

and, assuming that the data measure ρ equals the Hilbert space measure w, so that $\mu = \mathbb{1}$, the metric gradient descent equation 30 can be written in the form

$$\gamma\,\delta\theta = -\eta M^T\mathcal{R} \quad \text{where} \quad \gamma = M^T M\ . \tag{37}$$

If the numerical metric γ is invertible this is equivalent to

$$\delta\theta = -\eta\,M^+\mathcal{R} \quad \text{where} \quad M^+ = \gamma^{-1}M^T = (M^T M)^{-1}M^T \tag{38}$$

is the Moore-Penrose pseudoinverse of M. If the batch size $|B|$ is greater than or equal to the number of parameters N then γ is generically invertible and equation 38 is well-defined. Even in this case the metric might have small eigenvalues which lead to unwanted large excursions in parameter space. In the opposite case, when $|B| < N$, the metric γ is necessarily singular. In either case, we can still make sense of equation 38 and regularize the singularities (or small eigenvalues) by writing down an approximation $M \simeq U\sigma V^T$ for M, based on a singular value decomposition which only keeps the singular values $\sigma = \mathrm{diag}(\sigma_1,\ldots,\sigma_k)$ larger than a certain given fraction of the maximal singular value σ_1. In other words, we keep all singular values σ_i, where $i = 1,\ldots,k$, for which $\sigma_i/\sigma_1 \geq r$ for some $r \in (0,1)$. Then the numerical metric can be approximated by $\gamma \simeq V\sigma^2 V^T$ and the Moore-Penrose inverse of M by $M^+ \simeq V\sigma^{-1}U^T$. With these approximations the gradient descent equation 38 can be re-written as

$$\delta\theta \simeq -V\sigma^{-1}U^T\mathcal{R}\ . \tag{39}$$

This equation can be used in practice to realize a metric-based gradient descent, and is precisely the update rule implemented in Algorithm 1.

The above argument has been carried out for a mean square loss function but it can be generalized to loss functions of the form $L(\theta) = \frac{1}{2}||F(f_\theta, f)||^2$ for some function F. (For the above case of a mean square loss we have $F(f_\theta, f) = f_\theta - f$. For a cross-entropy loss we should take $F(f_\theta, f) = \sqrt{-f \ln \circ f_\theta}$. In general, this is the $\kappa = 1$ version of equation 8.) The gradient descent equation can then be written in the form

$$\delta\theta^i = -\eta\, g^{ij} \left\langle \frac{\partial F(f_\theta, f)}{\partial \theta^j}, F(f_\theta, f) \right\rangle \simeq -\eta\, g^{ij} \sum_{I\in\mathcal{I}} \frac{\partial c_I}{\partial \theta^j}(\theta) c_I(\theta) \tag{40}$$

where

$$c_I(\theta) = \langle h_I, F(f_\theta, f)\rangle \ . \tag{41}$$

Translating the training trajectory described by equation 40 to the coefficients c_I leads to

$$\delta c_I = -\eta\, G^{IJ} c_J \quad \text{where} \quad G^{-1} = J g^{-1} J^T \quad \text{and} \quad J = \frac{\partial c}{\partial \theta} \ . \tag{42}$$

If we choose the natural metric $G_{IJ} = \delta_{IJ}$ the coefficients c_I approach zero at an exponential rate, indicating efficient training. Finding the metric g closest to solving $J g^{-1} J^T = \mathbb{1}$ proceeds as above, by using the Moore-Penrose pseudoinverse of J, and results in

$$g_{ij} = (J^T J)_{ij} = \sum_I \frac{\partial c_I}{\partial \theta^i} \frac{\partial c_I}{\partial \theta^j} = \left\langle \frac{\partial F(f_\theta, f)}{\partial \theta^i}, \frac{\partial F(f_\theta, f)}{\partial \theta^j} \right\rangle \ . \tag{43}$$

As before, this metric can be calculated analytically for simple neural networks by evaluating the RHS of equation 43. For a data-driven approach we define the quantities analogous to the ones in equation 35 by

$$M_{\beta i} = \partial_1 F(f_\theta(x_\beta), y_\beta) \frac{\partial f_\theta}{\partial \theta^i}(x_\beta) \ , \quad \mathcal{R}_\beta = F(f_\theta(x_\beta), y_\beta) \ , \quad \mu_{\alpha\beta} = \delta_{\alpha\beta} \frac{w(x_\beta)}{\rho(x_\beta)} \ , \tag{44}$$

where (x_β, y_β) is the data, with a distribution ρ for the x_β values. Assuming $w = \rho$, as before, we find the numerical version $\gamma = M^T M$ of the metric g and the resulting gradient descent equation has the same form as equation 38, however with M and $\mathcal{R}$ now defined as in equation 44. Possible singularities in this equation can be regularized as before, by using an approximation for M based on its singular value decomposition but with only the largest singular values kept. The gradient descent then takes the same form as in equation 39 but the singular value decomposition should now be applied to M as defined in equation 44.

D Experiment Details

In the subsections below, we provide detailed information about experiments performed for the results presented in this paper. All results come from a campaign of training runs performed over six datasets and hyperparameter grids for each optimizer. We also perform three result-dependent follow-up runs to measure unbiased wall-times,[11] optimizer diagnostics, and confirmation runs. We run five model seeds per grid point for most experiments, limiting to three in the CIFAR-10 parameter-fraction study, and one in the GPT-2-small comparison, whose per-run cost (Appendix N) rules out seed-averaging. On the seven headline grids the configuration selected for each optimizer is re-run on five fresh seeds, and those confirmation runs are the numbers reported throughout the paper. Code for the Sven optimizer and all experiments is available at the links in Appendix A.

D.1 Datasets and Splits

Each dataset exposes three fixed, disjoint splits—train, validation and test—of the sizes in Table 1. The splits are identical for every optimizer, are fixed by a split seed that is independent of the model and data-loading seeds, and do not move when the training set is subsampled: the overparameterization study of Appendix I varies N by subsampling the *training* part only, so the held-out data and the target scale are the same in every arm. Hyperparameters are selected based on validation loss. For the synthetic datasets (1D regression and random polynomials), the data are generated in a reproducible fashion using configurable seeds.

[11] Multiple models were trained simultaneously per GPU to optimize resource usage during the hyperparameter grid scans. This artificially increases measured wall-time, so we re-run standalone at the best-performing parameter settings to measure timing.

Table 1: Dataset splits. Sizes are examples, or blocks for the two language-modeling corpora.

Dataset	Loss	Train	Val.	Test	Construction
1D regression	L_2	10k	10k	10k	independent draws; pool-standardized
Random polynomial	L_2	10k	10k	10k	independent draws; pool-standardized
MNIST	label reg./CE	50k	10k	10k	official train split; official test set
CIFAR-10	label reg./CE	45k	5k	10k	official train split; official test set
Shakespeare (char)	token CE	6,971	871	871	contiguous 80/10/10 by position
FineWeb-edu (BPE)	token CE	210k	200	200	val/test from disjoint documents

One-dimensional regression We sample train/val/test data from the function

$$f(x) = e^{-10x^2} \sin(2x), \tag{45}$$

with inputs drawn from $U[-1, 1]$. The training pool, the validation set and the test set are drawn from three independent generators, and the *targets* of all three are standardized by the mean and standard deviation of the training pool.

Random polynomials We fit a random polynomial over $\mathbb{R}^6$ built from all monomials of total degree at most four,

$$f(\mathbf{x}) = \sum_{\mathbf{d}} c_{\mathbf{d}}\, x_1^{d_1} \cdots x_6^{d_6}, \qquad \mathbf{d} \in \mathbb{N}^6, \quad \textstyle\sum_i d_i \le 4, \tag{46}$$

which is $\binom{6+4}{4} = 210$ terms including the constant and the linear terms. Inputs are drawn from $\mathcal{N}(0, I_6)$, and each coefficient is drawn as

$$c_{\mathbf{d}} \sim \mathcal{N}(0,1) \Big/ \sqrt{\mathbb{E}\big[m_{\mathbf{d}}(\mathbf{x})^2\big]}, \qquad \mathbb{E}\big[m_{\mathbf{d}}^2\big] = \prod_{j=1}^{6} (2d_j - 1)!!, \tag{47}$$

so that every monomial contributes the same variance and no single degree dominates the target. As for the 1D task, pool, validation and test inputs come from separate generators and the targets are standardized by the pool statistics. The coefficient draw is fixed by the data seed; the two synthetic tasks are replicated over three data seeds in the confirmation pass, so that between-instance variation is reported separately from seed variation.

MNIST We use MNIST (LeCun et al., 1998) with the official training set split $50{,}000/10{,}000$ into train and validation by a fixed split seed, and the official $10{,}000$-image test set held out. Images are standardized with the published statistics and flattened.

CIFAR-10 In Appendix L we present results from training a ResNet (He et al., 2016) on the CIFAR-10 image dataset (Krizhevsky & Hinton, 2009). The official training set is split $45{,}000/5{,}000$ into train and validation by the same fixed split seed and the official $10{,}000$-image test set is held out. Images are standardized with the published channel statistics.

Character-level Shakespeare The tiny-shakespeare corpus of nanoGPT (Karpathy, 2022) is encoded at character level over a 65-symbol vocabulary and split $80/10/10$ *contiguously by position*, train first, so that no sequence spans a split boundary. Each part is then chunked independently into non-overlapping blocks of 128 characters, with the target the input shifted by one. Results are in Appendix N.

FineWeb-edu (GPT-2 tokens) For the GPT-2-small (Radford et al., 2019) comparison we use FineWeb-edu (Penedo et al., 2024) tokenized with the GPT-2 BPE vocabulary (padded to $50{,}304$) and served as non-overlapping blocks of $1{,}024$ tokens. Validation and test blocks are drawn from documents *disjoint* from the training documents. One pass over the training blocks is the token budget of a run. Results are in Appendix N.

Table 2: Models and parameter counts P. The MLP notation lists input, hidden and output widths.

Task	Model	Configuration	P
1D regression	MLP	$1-16-16-16-1$, GeLU	593
Random polynomial	MLP	$6-16-16-16-1$, GeLU	673
MNIST	MLP	$784-32-32-32-10$, GeLU	27,562
CIFAR-10	ResNet18	10 classes, functional BatchNorm	11,181,642
Shakespeare (char)	nanoGPT	4 layers, 4 heads, $d = 128$, block 128	826,368
FineWeb-edu	GPT-2 small	12 layers, 12 heads, $d = 768$, block 1024	163,109,376

D.2 Model Architectures

We use MLPs with three hidden layers and GeLU activations (Hendrycks & Gimpel, 2016) for the regression and MNIST experiments. The hidden layer width is set to 16 for the 1D and polynomial datasets, and 32 for MNIST. For the CIFAR-10 experiments in Appendix L we use the `ResNet18` (He et al., 2016) architecture provided in `torchvision` (Ansel et al., 2024; TorchVision maintainers and contributors, 2016), with every `BatchNorm2d` layer replaced by a drop-in variant that does not write its running statistics in place; the numerics are unchanged, but the forward pass then composes with the `torch.func` transforms that Sven's Jacobian capture uses (Appendix E). The two language-modeling experiments use a decoder-only transformer of the nanoGPT form with untied embeddings and no dropout. Parameter counts are listed in Table 2.

D.3 Evaluation Protocol

Model Evaluation Validation, test and train losses are evaluated before training and after each epoch, with a fixed subset of the training data used to evaluate train loss. The reported loss is a mean over examples (over tokens for the language models), not a mean of per-batch means. Accuracy is the fraction of correct `argmax` predictions over the raw model outputs. The train loss subset consists of at most $10{,}000$ examples ($1{,}000$ on nanoGPT, 200 blocks on GPT-2-small) shared by all optimizers in a scan. The evaluation batch size is a separate setting and is never the training batch size. The GPT-2-small comparison is a single $13{,}125$-step epoch, so it is evaluated every 500 steps instead of every epoch.

Diverged Runs A run fails if it ends with a non-finite training or validation loss, if it stops early on a non-finite training batch, or if its final validation loss exceeds ten times the loss of the untrained model. Failed runs are tracked and excluded from seed means, and tables print the number of runs that finished out of the number attempted. Appendix Q gives the full accounting per scan and per method.

Selection. For each optimizer and each scan we select one configuration by a fixed rule applied to validation loss only: a configuration is eligible if more than half of its seeds finished; among eligible configurations we take the one with the fewest failed seeds, breaking ties by the lowest seed-mean final validation loss.

Confirmation seeds. For each of the headline scans, the selected configuration of every optimizer is re-run on five model seeds disjoint from the five it was tuned on, and the two synthetic tasks (1D and polynomial regression) are additionally replicated over three independent data draws. These confirmation runs are the numbers reported in the paper; the tuning-seed values are shown beside them, so that the optimism introduced by selection is visible rather than absorbed.

Paired comparisons. Within a scan, a model seed fixes both the initialization and the data order, so every optimizer at a given seed sees the same initial weights and the same sequence of batches. The uncertainty band drawn in figures is the mean $\pm$ one standard deviation over seeds.

Normalization layers. ResNet18 uses running statistics in its BatchNorm layers (Ioffe & Szegedy, 2015). These are updated once per optimizer step over the course of training during all CIFAR-10 scans, then frozen for evaluation. Sven's Jacobian capture makes several passes over the batch per

Table 3: Summary of the hyperparameter scans. "Seeds" indicates the five tuning seeds plus the five fresh confirmation seeds, with the number of independent data draws for the two synthetic tasks; "methods" indicates the number of optimizers used; "grid runs" is the total number of runs across optimizers, seeds, and hyperparameters; and the last column is the fraction of non-diverged confirmation runs.

Task	**Loss**	P	B	**Epochs**	**Seeds**	**Methods**	**Grid runs**	**Confirm fin./att.**
Toy 1D	squared error	593	32	20	5+5×3	15	1770	224/225
Random Polynomial	squared error	673	32	20	5+5×3	15	1630	212/225
MNIST (label reg.)	label regression	27562	64	20	5+5	14	1360	70/70
MNIST (CE)	cross-entropy	27562	64	20	5+5	14	1610	68/70
CIFAR-10 (label reg.)	label regression	11181642	128	20	5+5	11	600	55/55
CIFAR-10 (CE)	cross-entropy	11181642	128	20	5+5	11	785	55/55
nanoGPT (tiny-shakespeare)	token cross-entropy	826368	64	50	5+5	5	140	25/25

Table 4: Sven's hyperparameter grids. B is the training batch size, and "runs" counts the full grid including seeds. All grids use $\kappa = 2$. The CIFAR-10 cross-entropy grid was extended at $k = 128$, $\eta \in \{0.05, 0.1, 0.5\}$ with $\texttt{rtol} \in \{0.03, 0.1, 0.3\}$, because the optimum sat on the `rtol` edge.

Scan	B	k	η	`rtol`	Runs
1D regression	32	1,2,4,8,16,32	.01,.02,.05,.1,.5,1	$10^{-6}-10^{-2}$ (5)	900
Random polynomial	32	1,2,4,8,16,32	.05,.1,.5,1	$10^{-5}-10^{-1}$ (6)	720
MNIST label reg.	64	1,2,4,8,16,32,48,64	.05,.1,.5,1	$10^{-4}-10^{-1}$ (4)	640
MNIST CE	64	1,2,4,8,16,32,48,64	.05,.1,.5,1	$10^{-4}-0.3$ (5)	800
CIFAR-10 label reg.	128	64,128	.1,.5,1	$10^{-4}, 10^{-3}, 10^{-2}$	90
CIFAR-10 CE	128	64,128	.02,.05,.1,.5,1	$10^{-4}, 10^{-3}, 10^{-2}$	150
nanoGPT	64	64	.05,.1,.5,1	10^{-3}	20
GPT-2 small	16	16	.02,.05,.1,.5,1	10^{-3}	5

step, so we run them in a context that suppresses updates without switching the layer to eval mode (which would normalize with the running statistics instead and change the Jacobian). We also lightly modify the PyTorch `BatchNorm` implementation to avoid in-place tensor updates that are invalid inside the functional API `torch.func`, which is used for full-Jacobian capture (see Appendix E for full details of Sven's implementation).

Epochs. The four MLP scans and both CIFAR-10 scans train for 20 epochs, which was found to be sufficient for convergence in validation loss; nanoGPT trains for 50, the two full-batch overparameterization scans for 200, and GPT-2-small for a single epoch of 13,125 steps. Table 3 lists the protocol of the seven headline scans together with the number of confirmation runs that finished. The studies of Table 5 run five model seeds and their parent task's 20 epochs – three seeds for the CIFAR-10 parameter-fraction study, 200 epochs for the two full-batch P/N sweeps – and use no dedicated confirmation pass.

D.4 Hyperparameter Grids

We perform hyperparameter scans to find good working points for Sven and the other baseline optimizers. Tables 4 and 5 give Sven's grids, Table 6 the baselines' learning-rate grids, and Table 7 the settings that are held fixed. Weight decay is not swept: AdamW uses its default of 0.01, and MuonW[12] its default of 0.1. The selected best-performing configuration of each optimizer is reported in Tables 9–12 of Section D.6. In all cases we use Sven with $\kappa = 2$, treating the raw losses as the "residuals" (see Sec. 2.1). Appendix O explores other settings of κ. Furthermore, Sven's k and `rtol` are not independent: the rank actually used on a step is the smaller of k and the number of singular values above $\texttt{rtol} \cdot \sigma_{\max}$. We sweep both, but depending on `rtol` and the shape of the SV spectrum for a given problem, fewer than k may actually be used for Sven's parameter updates.

[12]We distinguish "Muon" and "MuonW" even though Muon has nonzero weight-decay by default. Sven uses no regularization, so we focus on non-regularized baselines.

Table 5: Sven's grids in follow-up studies varying κ, (micro-)batch size (μ), updated parameter fraction (f), and overparametrization (P/N). For MNIST, the two `rtol` values separated by a slash are the label-regression and cross-entropy settings. The P/N sweeps on the two synthetic tasks are full-batch ($B = N$).

Study	B	k	η	`rtol`	Swept axis	Runs
κ (MNIST lab. reg.)	64	32,64	.125–1.5 (7)	10^{-4}	κ: 1,2,3	210
Micro-batch (1D, poly.)	32	32	.05,.1,.5,1	10^{-3}	μ: 1–32 (6)	120
Micro-batch (MNIST)	64	64	.05,.1,.5,1	$10^{-4}/10^{-1}$	μ: 1–64 (7)	140
Param. frac. (1D, poly.)	32	32	.05,.1,.5,1	10^{-3}	f: .1–1 (5)	100
Param. frac. (MNIST)	64	64	.05,.1,.5,1	$10^{-4}/10^{-1}$	f: .1–1 (5)	100
Param. frac. (CIFAR-10)	128	128	.5	10^{-3}	f: .05–1 (5)	15
P/N (1D)	N	$.25B, .5B, B$	.05,.1,.5,1	$10^{-4}-10^{-2}$ (3)	N: 150–1200 (4)	720
P/N (polynomial)	N	$.25B, .5B, B$	.01–1 (6)	$10^{-4}-10^{-1}$ (5)	N: 170–1350 (4)	1800
P/N (MNIST)	64	16,32,48,64	.1,.5,1	10^{-4}	N: 2500–50000 (6)	360
Batch size (polynomial)	B	$k = B$	.05,.1,.5,1	$10^{-5}-10^{-2}$ (4)	B: 8–256 (6)	480

Table 6: Baseline learning-rate grids. The second column is the shared grid used by all standard optimizers; L-BFGS, JD and HIG have their own. L-BFGS additionally sweeps `max_iter` $\in \{1, 2, 3\}$ and `history_size` $\in \{2, 5, 10\}$; HIG additionally sweeps its cutoff $\tau \in \{10^{-8}, 10^{-6}, 10^{-4}, 10^{-2}, 3\cdot10^{-2}, 10^{-1}\}$. Polyak SGD has no learning rate to sweep. A dash means the method was not run on that scan.

Scan	Standard optimizers	L-BFGS	JD	HIG
1D regression	$10^{-5}-1$ (8)	.01,.03,.1,.5,1	$10^{-5}-10^{-1}$ (6)	$5\cdot10^{-4}-0.25$ (7)
Random polynomial	$10^{-6}-1$ (10)	.1,.5,1,2,4	$10^{-5}-10^{-1}$ (6)	$5\cdot10^{-3}-0.25$ (5)
MNIST label reg.	$10^{-5}-1$ (8)	.1,.5,1	$10^{-5}-10^{-1}$ (6)	$5\cdot10^{-3}-0.25$ (5)
MNIST CE	$10^{-5}-1$ (8)	.1,.5,1,2,4	$10^{-5}-10^{-1}$ (6)	$5\cdot10^{-3}-0.25$ (5)
CIFAR-10 label reg.	$10^{-5}-0.3$ (7)	.1,.5,1,2,4	—	—
CIFAR-10 CE	$10^{-5}-3$ (9)	.1,.5,1,2,4	—	—
nanoGPT	$10^{-5}-3\cdot10^{-3}$ (6)	—	—	—
GPT-2 small	$10^{-5}-3\cdot10^{-3}$ (6)	—	—	—

Muon parameter grouping Muon uses a subset of parameters in its orthogonalized updates, with the remaining ones optimized by AdamW. In our configuration, we send embedding layers, final linear/convolutional layers, biases, and normalization weights to AdamW. All other linear/convolutional weights use Muon, with convolution kernels flattened to two dimensions. The single shared learning rate is made comparable across the two groups by the RMS-matching rule (Liu et al., 2025), which scales the orthogonalized step to the size of an AdamW step.

HIG computational cost HIG computes the SVD of the stacked *model* Jacobian, whose row count is the batch size times the output dimension. Its Jacobian is therefore ten times taller than Sven's on MNIST and as tall as the vocabulary on a language model. This becomes intractable for larger models such as transformers or ResNets, and HIG is thus omitted from those scans.

Grids in follow-up studies The overparametrization and batch size sweeps use the same learning-rate grid for all ten standard optimizers, together with Polyak SGD and stochastic L-BFGS. L-BFGS sweeps learning rates against the same `max_iter` and `history_size` grids as before, except in the batch-size sweep where its shape is pinned at `max_iter` $= 3$, `history_size` $= 2$. JD and HIG are run on the four MLP headline scans only; the κ, micro-batch and parameter-fraction studies only use Sven.

D.5 Compute Resources and Wall-Time Measurement

All runs were performed on either 80GB NVIDIA A100 GPUs or 20GB MIG slices of 80GB A100s. The MLP scans used small enough models that several could be run on a single device, while the

Table 7: Baseline optimizers and the settings held fixed.

Optimizer	Fixed settings
Adam (Kingma & Ba, 2015)	PyTorch defaults, weight decay 0
AdamW (Loshchilov & Hutter, 2019)	PyTorch defaults, weight decay 0.01
SGD	no momentum
SGD + momentum	momentum 0.9
RMSprop (Hinton et al., 2012)	PyTorch defaults
Muon (Jordan et al., 2024)	hidden projections only, weight decay 0 (see below)
MuonW	as Muon, weight decay 0.1 (0.01 on the AdamW group)
SOAP (Vyas et al., 2025)	$\beta = (0.95, 0.95)$, preconditioning every 10 steps
Shampoo (Gupta et al., 2018)	update frequency 1, $\epsilon = 10^{-4}$
K-FAC (Martens & Grosse, 2015)	SGD + momentum 0.9 base, damping $3 \cdot 10^{-3}$, factors updated every step
Stochastic L-BFGS (Liu & Nocedal, 1989)	strong-Wolfe line search; `max_iter` and `history_size` swept
Polyak SGD (Polyak, 1969)	$\eta = (L - f^\star)/(\|\nabla L\|^2 + \epsilon)$, $f^\star = 0$, capped at 1
JD (UPGrad) (Quinton & Rey, 2024)	UPGrad aggregator, Adam inner optimizer
HIG (Schnell et al., 2022)	$\Sigma^{-1/2}$ of the stacked output Jacobian; cutoff τ swept

Table 8: Number of distinct configurations of each method on each scan (excluding seed duplicates); a blank means the method was not run. Sven sweeps $(k, \eta, \texttt{rtol})$ where most baselines sweep a learning rate alone.

Method	**1D reg.**	**Poly.**	**MNIST-LR**	**MNIST-CE**	**CIFAR-LR**	**CIFAR-CE**	**nanoGPT**
Sven	180	144	128	160	18	39	4
Adam	8	10	8	8	7	9	–
AdamW	8	10	8	8	7	9	6
HIG	42	30	30	30	–	–	–
JD (UPGrad)	6	6	6	6	–	–	–
KFAC	8	10	–	–	–	–	–
Stochastic L-BFGS	45	45	27	45	45	45	–
Muon	8	10	8	8	7	9	6
MuonW	8	10	8	8	7	9	6
Polyak SGD	1	1	1	1	1	1	–
RMSprop	8	10	8	8	7	9	–
SGD	8	10	8	8	7	9	–
SGD + momentum	8	10	8	8	7	9	–
SOAP	8	10	8	8	7	9	6
Shampoo	8	10	8	8	–	–	–

CIFAR-10, nanoGPT and GPT-2 runs used a full 80GB A100. Per-step and per-epoch memory and wall-time costs for every method and architecture are in Appendix R.

Wall-clock numbers reported in the paper are not taken from the grid scans, because several processes shared a device to optimize utilization and thus increased step time relative to a standalone run. Instead, each scan has a dedicated timing run using the selected best configuration of each optimizer. These are run one process at a time on a single device, with checkpointing and diagnostic logging switched off. Timing runs reproduce the grid runs' hyperparameters exactly and are used for step and epoch times only; loss values come from the grid or confirmation passes.

D.6 Full Grid Summary

Table 8 counts the distinct configurations of each method on each scan. Tables 9, 10, 11, 12, and 13 fully enumerate the grids for each scan, and indicate the selected best-performing configuration measured by final validation loss. A selected value carrying $\uparrow$ ($\downarrow$) is the largest (smallest) value that axis was swept over, i.e. a grid edge, and indicates where results may be bounded by the grid.

Table 9: Hyperparameter grid for the 1D regression scan, with the best-performing configuration indicated. Weight decay is at each optimizer's default.

Method	Grid swept	Configs	Runs	Selected
Sven	$\eta \in$ 0.01, 0.02, 0.05, 0.1, 0.5, 1; $k \in$ 1, 2, 4, 8, 16, 32; rtol $\in$ 1e-06, 1e-05, 0.0001, 0.001, 0.01	180	900	k=32$^{\uparrow}$, lr=0.01$^{\downarrow}$, rtol=0.0001
Adam	$\eta \in$ 1e-05, 3e-05, 0.0001, 0.001, 0.01, 0.1, 0.3, 1	8	40	lr=0.001
AdamW	$\eta \in$ 1e-05, 3e-05, 0.0001, 0.001, 0.01, 0.1, 0.3, 1	8	40	lr=0.001, weight_decay=0.01
HIG	$\eta \in$ 0.0005, 0.0015, 0.005, 0.015, 0.05, 0.1, 0.25; $\tau \in$ 1e-08, 1e-06, 0.0001, 0.01, 0.03, 0.1	42	210	lr=0.005, tau=1e-06
JD (UPGrad)	$\eta \in$ 1e-05, 3e-05, 0.0001, 0.001, 0.01, 0.1	6	30	lr=0.01, aggregator=UPGrad, inner_optimizer=Adam
KFAC	$\eta \in$ 1e-05, 3e-05, 0.0001, 0.001, 0.01, 0.1, 0.3, 1	8	40	lr=0.001
Stochastic L-BFGS	$\eta \in$ 0.01, 0.03, 0.1, 0.5, 1; max iter $\in$ 1, 2, 3; history $\in$ 2, 5, 10	45	225	lr=0.03, max_iter=3, history_size=10
Muon	$\eta \in$ 1e-05, 3e-05, 0.0001, 0.001, 0.01, 0.1, 0.3, 1	8	40	lr=0.001
MuonW	$\eta \in$ 1e-05, 3e-05, 0.0001, 0.001, 0.01, 0.1, 0.3, 1	8	40	lr=0.001, weight_decay=0.1
Polyak SGD	no swept hyperparameter	1	5	f_star=0, max_lr=1, eps=1e-08
RMSprop	$\eta \in$ 1e-05, 3e-05, 0.0001, 0.001, 0.01, 0.1, 0.3, 1	8	40	lr=0.001
SGD	$\eta \in$ 1e-05, 3e-05, 0.0001, 0.001, 0.01, 0.1, 0.3, 1	8	40	lr=0.3
SGD + momentum	$\eta \in$ 1e-05, 3e-05, 0.0001, 0.001, 0.01, 0.1, 0.3, 1	8	40	lr=0.1
SOAP	$\eta \in$ 1e-05, 3e-05, 0.0001, 0.001, 0.01, 0.1, 0.3, 1	8	40	lr=0.001
Shampoo	$\eta \in$ 1e-05, 3e-05, 0.0001, 0.001, 0.01, 0.1, 0.3, 1	8	40	lr=0.1

Table 10: Hyperparameter grid for polynomial regression. Conventions as in Table 9.

Method	Grid swept	Configs	Runs	Selected
Sven	$\eta \in$ 0.05, 0.1, 0.5, 1; $k \in$ 1, 2, 4, 8, 16, 32; rtol $\in$ 1e-05, 0.0001, 0.001, 0.01, 0.03, 0.1	144	720	k=16, lr=0.5, rtol=0.03
Adam	$\eta \in$ 1e-06, 3e-06, 1e-05, 3e-05, 0.0001, 0.001, 0.01, 0.1, 0.3, 1	10	50	lr=0.01
AdamW	$\eta \in$ 1e-06, 3e-06, 1e-05, 3e-05, 0.0001, 0.001, 0.01, 0.1, 0.3, 1	10	50	lr=0.01, weight_decay=0.01
HIG	$\eta \in$ 0.005, 0.015, 0.05, 0.1, 0.25; $\tau \in$ 1e-08, 1e-06, 0.0001, 0.01, 0.03, 0.1	30	150	lr=0.1, tau=1e-08$^{\downarrow}$
JD (UPGrad)	$\eta \in$ 1e-05, 3e-05, 0.0001, 0.001, 0.01, 0.1	6	30	lr=0.001, aggregator=UPGrad, inner_optimizer=Adam
KFAC	$\eta \in$ 1e-06, 3e-06, 1e-05, 3e-05, 0.0001, 0.001, 0.01, 0.1, 0.3, 1	10	50	lr=1e-05
Stochastic L-BFGS	$\eta \in$ 0.1, 0.5, 1, 2, 4; max iter $\in$ 1, 2, 3; history $\in$ 2, 5, 10	45	225	lr=1, max_iter=1, history_size=2
Muon	$\eta \in$ 1e-06, 3e-06, 1e-05, 3e-05, 0.0001, 0.001, 0.01, 0.1, 0.3, 1	10	50	lr=0.01
MuonW	$\eta \in$ 1e-06, 3e-06, 1e-05, 3e-05, 0.0001, 0.001, 0.01, 0.1, 0.3, 1	10	50	lr=0.01, weight_decay=0.1
Polyak SGD	no swept hyperparameter	1	5	f_star=0, max_lr=1, eps=1e-08
RMSprop	$\eta \in$ 1e-06, 3e-06, 1e-05, 3e-05, 0.0001, 0.001, 0.01, 0.1, 0.3, 1	10	50	lr=0.01
SGD	$\eta \in$ 1e-06, 3e-06, 1e-05, 3e-05, 0.0001, 0.001, 0.01, 0.1, 0.3, 1	10	50	lr=0.1
SGD + momentum	$\eta \in$ 1e-06, 3e-06, 1e-05, 3e-05, 0.0001, 0.001, 0.01, 0.1, 0.3, 1	10	50	lr=0.01
SOAP	$\eta \in$ 1e-06, 3e-06, 1e-05, 3e-05, 0.0001, 0.001, 0.01, 0.1, 0.3, 1	10	50	lr=0.01
Shampoo	$\eta \in$ 1e-06, 3e-06, 1e-05, 3e-05, 0.0001, 0.001, 0.01, 0.1, 0.3, 1	10	50	lr=0.1

Table 11: Hyperparameter grids for MNIST label regression (top) and cross-entropy (bottom). Conventions as in Table 9.

Method	Grid swept	Configs	Runs	Selected
Sven	$\eta \in$ 0.05, 0.1, 0.5, 1; $k \in$ 1, 2, 4, 8, 16, 32, 48, 64; rtol $\in$ 0.0001, 0.001, 0.01, 0.1	128	640	k=64$^{\uparrow}$, lr=0.5, rtol=0.001
Adam	$\eta \in$ 1e-05, 3e-05, 0.0001, 0.001, 0.01, 0.1, 0.3, 1	8	40	lr=0.001
AdamW	$\eta \in$ 1e-05, 3e-05, 0.0001, 0.001, 0.01, 0.1, 0.3, 1	8	40	lr=0.001, weight_decay=0.01
HIG	$\eta \in$ 0.005, 0.015, 0.05, 0.1, 0.25; $\tau \in$ 1e-08, 1e-06, 0.0001, 0.01, 0.03, 0.1	30	150	lr=0.05, tau=0.03
JD (UPGrad)	$\eta \in$ 1e-05, 3e-05, 0.0001, 0.001, 0.01, 0.1	6	30	lr=0.001, aggregator=UPGrad, inner_optimizer=Adam
Stochastic L-BFGS	$\eta \in$ 0.1, 0.5, 1; max iter $\in$ 1, 2, 3; history $\in$ 2, 5, 10	27	135	lr=0.1$^{\downarrow}$, max_iter=3, history_size=2
Muon	$\eta \in$ 1e-05, 3e-05, 0.0001, 0.001, 0.01, 0.1, 0.3, 1	8	40	lr=0.001
MuonW	$\eta \in$ 1e-05, 3e-05, 0.0001, 0.001, 0.01, 0.1, 0.3, 1	8	40	lr=0.001, weight_decay=0.1
Polyak SGD	no swept hyperparameter	1	5	f_star=0, max_lr=1, eps=1e-08
RMSprop	$\eta \in$ 1e-05, 3e-05, 0.0001, 0.001, 0.01, 0.1, 0.3, 1	8	40	lr=0.001
SGD	$\eta \in$ 1e-05, 3e-05, 0.0001, 0.001, 0.01, 0.1, 0.3, 1	8	40	lr=0.1
SGD + momentum	$\eta \in$ 1e-05, 3e-05, 0.0001, 0.001, 0.01, 0.1, 0.3, 1	8	40	lr=0.01
SOAP	$\eta \in$ 1e-05, 3e-05, 0.0001, 0.001, 0.01, 0.1, 0.3, 1	8	40	lr=0.01
Shampoo	$\eta \in$ 1e-05, 3e-05, 0.0001, 0.001, 0.01, 0.1, 0.3, 1	8	40	lr=0.1

Method	Grid swept	Configs	Runs	Selected
Sven	$\eta \in$ 0.05, 0.1, 0.5, 1; $k \in$ 1, 2, 4, 8, 16, 32, 48, 64; rtol $\in$ 0.0001, 0.001, 0.01, 0.1, 0.3	160	800	k=32, lr=0.5, rtol=0.3$^{\uparrow}$
Adam	$\eta \in$ 1e-05, 3e-05, 0.0001, 0.001, 0.01, 0.1, 0.3, 1	8	40	lr=0.0001
AdamW	$\eta \in$ 1e-05, 3e-05, 0.0001, 0.001, 0.01, 0.1, 0.3, 1	8	40	lr=0.001, weight_decay=0.01
HIG	$\eta \in$ 0.005, 0.015, 0.05, 0.1, 0.25; $\tau \in$ 1e-08, 1e-06, 0.0001, 0.01, 0.03, 0.1	30	150	lr=0.05, tau=0.03
JD (UPGrad)	$\eta \in$ 1e-05, 3e-05, 0.0001, 0.001, 0.01, 0.1	6	30	lr=0.0001, aggregator=UPGrad, inner_optimizer=Adam
Stochastic L-BFGS	$\eta \in$ 0.1, 0.5, 1, 2, 4; max iter $\in$ 1, 2, 3; history $\in$ 2, 5, 10	45	225	lr=0.5, max_iter=1, history_size=2
Muon	$\eta \in$ 1e-05, 3e-05, 0.0001, 0.001, 0.01, 0.1, 0.3, 1	8	40	lr=0.0001
MuonW	$\eta \in$ 1e-05, 3e-05, 0.0001, 0.001, 0.01, 0.1, 0.3, 1	8	40	lr=0.0001, weight_decay=0.1
Polyak SGD	no swept hyperparameter	1	5	f_star=0, max_lr=1, eps=1e-08
RMSprop	$\eta \in$ 1e-05, 3e-05, 0.0001, 0.001, 0.01, 0.1, 0.3, 1	8	40	lr=0.0001
SGD	$\eta \in$ 1e-05, 3e-05, 0.0001, 0.001, 0.01, 0.1, 0.3, 1	8	40	lr=0.01
SGD + momentum	$\eta \in$ 1e-05, 3e-05, 0.0001, 0.001, 0.01, 0.1, 0.3, 1	8	40	lr=0.001
SOAP	$\eta \in$ 1e-05, 3e-05, 0.0001, 0.001, 0.01, 0.1, 0.3, 1	8	40	lr=0.0001
Shampoo	$\eta \in$ 1e-05, 3e-05, 0.0001, 0.001, 0.01, 0.1, 0.3, 1	8	40	lr=0.1

Table 12: Hyperparameter grid for nanoGPT on character-level Shakespeare. We only use one k and `rtol` for Sven, so the scan is over η. Conventions as in Table 9.

Method	Grid swept	Configs	Runs	Selected
Sven	$\eta \in$ 0.05, 0.1, 0.5, 1	4	20	k=64, lr=0.1, rtol=0.001
AdamW	$\eta \in$ 1e-05, 3e-05, 0.0001, 0.0003, 0.001, 0.003	6	30	lr=0.0003, weight_decay=0.01
Muon	$\eta \in$ 1e-05, 3e-05, 0.0001, 0.0003, 0.001, 0.003	6	30	lr=0.0001
MuonW	$\eta \in$ 1e-05, 3e-05, 0.0001, 0.0003, 0.001, 0.003	6	30	lr=0.0001, weight_decay=0.1
SOAP	$\eta \in$ 1e-05, 3e-05, 0.0001, 0.0003, 0.001, 0.003	6	30	lr=0.0001

Table 13: Hyperparameter grids for CIFAR-10 label regression (top) and cross-entropy (bottom). The cross-entropy `rtol` axis was extended after the first pass placed the optimum at its edge, which is why that grid is ragged. Conventions as in Table 9.

Method	Grid swept	Configs	Runs	Selected
Sven	$\eta \in$ 0.1, 0.5, 1; $k \in$ 64, 128; rtol $\in$ 0.0001, 0.001, 0.01	18	90	k=128$^{\uparrow}$, lr=0.5, rtol=0.001
Adam	$\eta \in$ 1e-05, 3e-05, 0.0001, 0.001, 0.01, 0.1, 0.3	7	35	lr=0.01
AdamW	$\eta \in$ 1e-05, 3e-05, 0.0001, 0.001, 0.01, 0.1, 0.3	7	35	lr=0.01, weight_decay=0.01
Stochastic L-BFGS	$\eta \in$ 0.1, 0.5, 1, 2, 4; max iter $\in$ 1, 2, 3; history $\in$ 2, 5, 10	45	225	lr=2, max_iter=3, history_size=2
Muon	$\eta \in$ 1e-05, 3e-05, 0.0001, 0.001, 0.01, 0.1, 0.3	7	35	lr=0.01
MuonW	$\eta \in$ 1e-05, 3e-05, 0.0001, 0.001, 0.01, 0.1, 0.3	7	35	lr=0.01, weight_decay=0.1
Polyak SGD	no swept hyperparameter	1	5	f_star=0, max_lr=1, eps=1e-08
RMSprop	$\eta \in$ 1e-05, 3e-05, 0.0001, 0.001, 0.01, 0.1, 0.3	7	35	lr=0.001
SGD	$\eta \in$ 1e-05, 3e-05, 0.0001, 0.001, 0.01, 0.1, 0.3	7	35	lr=0.001
SGD + momentum	$\eta \in$ 1e-05, 3e-05, 0.0001, 0.001, 0.01, 0.1, 0.3	7	35	lr=0.001
SOAP	$\eta \in$ 1e-05, 3e-05, 0.0001, 0.001, 0.01, 0.1, 0.3	7	35	lr=0.01

Method	Grid swept	Configs	Runs	Selected
Sven	$\eta \in$ 0.02, 0.05, 0.1, 0.5, 1; $k \in$ 64, 128; rtol $\in$ 0.0001, 0.001, 0.01, 0.03, 0.1, 0.3	39	195	k=128$^{\uparrow}$, lr=0.5, rtol=0.3$^{\uparrow}$
Adam	$\eta \in$ 1e-05, 3e-05, 0.0001, 0.001, 0.01, 0.1, 0.3, 1, 3	9	45	lr=0.001
AdamW	$\eta \in$ 1e-05, 3e-05, 0.0001, 0.001, 0.01, 0.1, 0.3, 1, 3	9	45	lr=0.01, weight_decay=0.01
Stochastic L-BFGS	$\eta \in$ 0.1, 0.5, 1, 2, 4; max iter $\in$ 1, 2, 3; history $\in$ 2, 5, 10	45	225	lr=2, max_iter=2, history_size=5
Muon	$\eta \in$ 1e-05, 3e-05, 0.0001, 0.001, 0.01, 0.1, 0.3, 1, 3	9	45	lr=0.01
MuonW	$\eta \in$ 1e-05, 3e-05, 0.0001, 0.001, 0.01, 0.1, 0.3, 1, 3	9	45	lr=0.01, weight_decay=0.1
Polyak SGD	no swept hyperparameter	1	5	f_star=0, max_lr=1, eps=1e-08
RMSprop	$\eta \in$ 1e-05, 3e-05, 0.0001, 0.001, 0.01, 0.1, 0.3, 1, 3	9	45	lr=0.001
SGD	$\eta \in$ 1e-05, 3e-05, 0.0001, 0.001, 0.01, 0.1, 0.3, 1, 3	9	45	lr=3$^{\uparrow}$
SGD + momentum	$\eta \in$ 1e-05, 3e-05, 0.0001, 0.001, 0.01, 0.1, 0.3, 1, 3	9	45	lr=0.3
SOAP	$\eta \in$ 1e-05, 3e-05, 0.0001, 0.001, 0.01, 0.1, 0.3, 1, 3	9	45	lr=0.01

E EFFICIENT IMPLEMENTATION OF SVEN

The cost of a naive Sven step is dominated by forming the $B \times P$ Jacobian, which is tens of gigabytes at ResNet18 scale. This appendix describes the optimized implementation we develop for this paper, which avoids fully materializing M in memory. It rests on the observation that the truncated pseudoinverse update depends on M only through the $B \times B$ Gram matrix MM^T and through one vector–Jacobian product $M^T w$, both of which can be obtained from a constant number of ordinary forward and backward passes. The classic path — fully materializing M and performing a randomized truncated SVD (Appendix R, Algorithms 3 and 4) — was implemented but is not used in practice.

Note: The efficient Gram implementation is only possible for architectures that do not mix between samples in a batch (e.g. BatchNorm) or reuse components in multiple places (e.g. transformer input/output weight tying).

E.1 THE GRAM IDENTITY

Write the thin SVD of the batch Jacobian as $M = UsV^T$, with $U \in \mathbb{R}^{B\times B}$, $s = \mathrm{diag}(\sigma_1 \geq \cdots \geq \sigma_B)$ and $V \in \mathbb{R}^{P\times B}$. Truncating to an index set $\mathcal{K}$ of retained directions, the update of Equation 7 is

$$M^{+}\mathcal{R} \;=\; \sum_{i\in\mathcal{K}} v_i\, \sigma_i^{-1}\, (u_i^T \mathcal{R}) \;=\; M^T \underbrace{\left[\sum_{i\in\mathcal{K}} u_i\, \sigma_i^{-2}\, (u_i^T \mathcal{R})\right]}_{w \;\in\; \mathbb{R}^B}, \tag{48}$$

using $M^T u_i = \sigma_i v_i$. The right singular vectors v_i, the only objects of dimension P, have canceled. The pairs (σ_i, u_i) are the square roots of the eigenvalues and the eigenvectors of the Gram matrix

$$G \;=\; MM^T \;=\; Us^2U^T \;\in\; \mathbb{R}^{B\times B}. \tag{49}$$

This requires only one eigendecomposition of a symmetric matrix of size $B \ll P$. The product $M^T w$, which is the gradient of the scalar $\sum_\alpha w_\alpha \mathcal{R}^\alpha(\theta)$ with respect to θ, requires one ordinary backward pass of a weighted residual sum. Both G and its eigendecomposition are kept in double precision regardless of the precision the model trains in (discussed in Appendix E.4).

E.2 ASSEMBLING THE GRAM MATRIX WITHOUT PER-SAMPLE GRADIENTS

Equation 49 is a sum of inner products of per-sample gradients, and the default Sven capture (`hooks`) obtains all of them from one forward and one backward pass. Forward hooks on every parameter-carrying layer l record that layer's input x_l and output z_l. A single backward pass then differentiates the *sum* of the residual rows with respect to the captured outputs,

$$g_l \;=\; \frac{\partial}{\partial z_l} \sum_\alpha \mathcal{R}^\alpha, \tag{50}$$

and because distinct conditions are decoupled — $\mathcal{R}^\alpha$ depends only on the activations of sample α — the slice of g_l belonging to sample α is exactly $\partial\mathcal{R}^\alpha/\partial z_l^\alpha$. One backward pass therefore yields all B per-sample output gradients, and it targets layer *outputs*, so no parameter gradient is ever formed. Because the rows enter only through Equation 50, the chain factors of the κ power, of the signed-residual row definition and of the micro-batch mean (Appendix E.5) are already inside g_l and need no special treatment.

For a layer applied once per sample with a two-dimensional input, the per-sample gradient with respect to the weight is the outer product $\partial\mathcal{R}^\alpha/\partial W_l = g_l^\alpha (x_l^\alpha)^T$, so its contribution to G factorizes into the Hadamard product of two Gram matrices,

$$G^{(l)} \;=\; \left(g_l g_l^T\right) \odot \left(x_l x_l^T\right) \;+\; \left(g_l g_l^T\right)\big|_{\text{bias}}, \qquad G \;=\; \sum_l G^{(l)}, \tag{51}$$

where the second term is present when the layer has a bias. Only $B \times B$ objects are built; the per-sample gradient itself is never stored. This factorization is standard in the per-example-gradient literature (Goodfellow, 2015); what is new here is only that the Gram matrix, and not the per-sample gradients, is what we need out of it. Three cases need more than Equation 51:

- **Layers applied at many positions per sample** (a projection inside a transformer block, a convolution). The per-sample gradient is then a sum of outer products over positions and no longer factorizes. The implementation forms the flattened per-sample gradient blocks explicitly — $\sum_{\text{pos}} g_l^\alpha \, (x_l^\alpha)^T$ for a projection, $\texttt{unfold}(x_l)$ contracted against g_l for a `Conv2d` — and contracts them into G immediately. Each block is $\mathcal{O}(B\, d_{\text{out}} d_{\text{in}})$ rather than $\mathcal{O}(BP)$, and the batch is processed in chunks whenever a block would exceed a fixed element budget, one side being recomputed for off-diagonal chunk pairs rather than held, so the transient is bounded independently of B.

- **Embeddings**, whose per-sample gradient is a sparse scatter of g_l into the rows the sample's tokens select. These are densely scattered into blocks and contracted under the same budget.

- **Normalization affine parameters**. With the statistics held fixed the per-sample gradients are $\sum_{\text{pos}} g_l \odot \hat{x}_l$ and $\sum_{\text{pos}} g_l$, each contributing AA^T with A the resulting (B, C) matrix. `LayerNorm` normalizes over trailing dimensions at each position, so it is per-sample by construction and $\hat{x}_l$ is recomputed from the captured input; layers that normalize with batch statistics are treated in Appendix E.3.

The capture therefore costs one forward and one backward pass plus the per-layer contractions. The memory cost is what an ordinary backward pass already holds plus the double-precision contractions and $B \times B$ accumulator; much smaller than the $B \times P$ Jacobian. A complete step is two forward–backward pairs (capture and $M^T w$) and one $B \times B$ eigendecomposition.

E.3 Architectural restrictions and fallbacks

Equation 50 is exact only where the batch elements are decoupled. The efficient implementation requires that every captured layer is applied exactly once per forward pass, that no parameters be tied between layers, and that nothing resample between the capture and the update. Train-mode dropout, repeated modules, tied weights, grouped or non-zero-padded convolutions, a normalization layer with no running statistics to hold fixed, and any parameter-carrying layer outside the supported set (linear, `Conv2d` with a single group, normalization, `LayerNorm`, embedding) are not supported.

The important case is BatchNorm, where activations of sample α depend on the whole batch. This couples batch elements, and what the hooks recover is not $\partial \mathcal{R}^\alpha / \partial \theta$. Freezing the statistics severely degrades performance for ResNet18 on CIFAR-10, so the ResNet18 experiments (Appendix L) use a different capture. Two exact fallbacks compute G from Jacobian blocks of bounded size, both by reverse-mode differentiation of the residual rows with respect to a *subset* of the parameters at a time:

- `full`: one group holding every parameter, i.e. the dense (B, P) Jacobian materialized once and contracted into G. It needs the fewest passes and the most memory. We use this form for the ResNet18 studies.

- `chunked`: the parameter tensors are partitioned into groups of at most a fixed number of elements; each group's (B, P_{grp}) Jacobian block is formed, contracted into G and discarded, and a single tensor larger than the budget forms its own group whose rows are themselves computed in batches. Peak memory is set by the largest group rather than by P, at the price of one differentiation pass per group. We do not use this form in this paper, but it would be necessary for very large models that could not use the efficient Gram capture.

E.4 Truncation and numerical resolution

The retained set $\mathcal{K}$ of Equation 48 is formed in the Gram eigenbasis, reproducing the truncation of Algorithm 1: the eigenvalues are sorted in descending order, $\sigma_i = \sqrt{\max(\lambda_i, 0)}$, the ones with $\sigma_i \leq \texttt{rtol} \cdot \sigma_1$ are dropped, and of those that survive the k largest are kept. An empty $\mathcal{K}$ means G is zero or non-finite, and raises an error (runs counted as "diverged"). Forming G squares the condition number, $\text{cond}(G) = \text{cond}(M)^2$, which sets the resolution of the spectrum. We thus accumulate G in float64 regardless of what the model uses.

Algorithm 2 Sven update step, Gram implementation. Lines 1–4 are the `hooks` capture; the `full` and `chunked` captures replace them by Jacobian blocks contracted into the same G (Appendix E.3).

Require: batch of conditions $\{\alpha\}$ with B rows, parameters θ, learning rate η, rank k, tolerance `rtol`, exponent κ

1: $\{x_l, z_l\}$, $\mathcal{R}^\alpha \leftarrow$ one forward pass, recording each layer's input and output ▷ $\mathcal{R}^\alpha = (\ell^\alpha)^{\kappa/2}$
2: $g_l \leftarrow \partial\big(\sum_\alpha \mathcal{R}^\alpha\big)/\partial z_l$ ▷ one backward; rows decoupled
3: $G \leftarrow \sum_l \left[(g_l g_l^T) \odot (x_l x_l^T) + (g_l g_l^T)|_{\text{bias}}\right]$ ▷ (B, B), float64
4: $\lambda, U \leftarrow \text{eigh}(G)$, sorted descending; $\sigma_i \leftarrow \sqrt{\max(\lambda_i, 0)}$ ▷ independent of k
5: $\mathcal{K} \leftarrow \{i \le k \,:\, \sigma_i > \texttt{rtol} \cdot \sigma_1\}$ ▷ Equation 53
6: **if** $\mathcal{K} = \emptyset$ **then**
7: **return** failure ▷ G zero or non-finite: the run has diverged
8: **end if**
9: $w \leftarrow \sum_{i \in \mathcal{K}} u_i \, \sigma_i^{-2} \, (u_i^T R)$
10: $\delta\theta \leftarrow -\eta \, \partial\big(\sum_\alpha w_\alpha \mathcal{R}^\alpha\big)/\partial\theta$ ▷ $= -\eta \, M^T w$; one forward, one backward
11: **return** $\theta + \delta\theta$

E.5 Micro-batches and parameter masks

To reduce memory usage in cases where the efficient Gram capture cannot be used, we introduce two options: micro-batching and parameter batching. We describe them here, and include experimental studies in Appendix P.

Micro-batches Instead of treating a full batch elementwise, we group it into "micro-batches" $\mathcal{D}_b$ of size $1 < \mu \le B$ and sum- or mean-aggregate sample-wise losses in a micro-batch into a single scalar loss. Mathematically,

$$L(\theta) = \sum_{i=1}^{B} L_i(\theta) = \sum_{b=1}^{B/\mu} \underbrace{\sum_{\alpha \in \mathcal{D}_b} \ell^\alpha(\theta)}_{\tilde{\ell}_b(\theta)}, \tag{52}$$

where the $\mathcal{D}_b$ are a partition of the batch. We then apply Sven at the level of micro-batches, using $\tilde{\ell}_b$ as the scalar losses. This requires instantiating only a $(B/\mu) \times P$ Jacobian, saving a factor of μ in memory. As the decomposition becomes coarser (fewer, larger micro-batches), our method converges to traditional SGD.

Batched parameters. While batching *data* is standard practice to reduce computational overhead and introduce beneficial stochasticity, batching *parameters* is rarely, if ever, considered. The batched parameter approach simply involves updating a randomly selected subset of parameters at each optimizer step, and is implemented using masks. In principle this should reduce memory overhead, but in practice we find that PyTorch and JAX still appear to instantiate full intermediate gradients "under the hood", and the memory savings are not realized. It is also worth noting that if we allow for sufficiently batched parameters, we can instead effectively learn in the underparametrized limit, and use the analytic form for the Moore-Penrose inverse in Equation 12.

E.6 The algorithm

Algorithm 2 summarizes the memory-optimized Sven step. Unlike Algorithm 1, the cost is independent of k since the whole spectrum of $G \in \mathbb{R}^{B \times B}$ comes out of the eigendecomposition.

E.7 Measured cost

Table 14 gives Sven's step time and peak memory for the four backends across the architectures of this paper (see Appendix R for a full profiling study against the full set of baseline optimizers). The quoted numbers come from the dedicated timing runs discussed in Appendix D. When available, the `hooks` Gram capture is fastest and costs only 2-3× as much time as an Adam step. For the small MLPs (toy 1D, polynomial) all three Gram capture methods use a similar amount of memory, but

Table 14: Step time and peak memory of Sven's four backends: `hooks`, `full` and `chunked` Gram captures and the classic explicit-Jacobian path with a randomized truncated SVD. Column 4 reports the step time ratio relative to Adam. The final column reports the spread of the step time across k. Missing rows are configurations that cannot be run: `hooks` is unavailable on ResNet18 because it cannot handle BatchNorm.

Architecture	Backend	Step (ms)	x Adam	Capture (ms)	Capture share	Solve+apply (ms)	Peak (MB)	k spread
Toy 1D MLP	Sven (Gram, hooks)	11.5	3.11	4.78	42%	6.72	18.86	1.02
	Sven (Gram, full J)	15.4	4.16	7.74	50%	7.67	18.86	1
	Sven (Gram, chunked)	159.7	43.2	152	95%	8.11	18.86	–
	Sven (classic, rand. SVD)	10.28	2.78	7.39	72%	2.89	21.41	1.01
Polynomial MLP	Sven (Gram, hooks)	11.71	3.15	4.86	41%	6.86	18.87	1.01
	Sven (Gram, full J)	15.39	4.15	7.72	50%	7.66	18.87	1
	Sven (Gram, chunked)	163.6	44.1	155	95%	8.08	18.87	–
	Sven (classic, rand. SVD)	10.5	2.83	7.46	71%	3.04	21.44	1.01
MNIST MLP	Sven (Gram, hooks)	10.28	2.62	3.79	37%	6.49	20.89	1
	Sven (Gram, full J)	14.31	3.65	6.64	46%	7.66	48.07	1.01
	Sven (Gram, chunked)	155.9	39.7	147	94%	9.09	45.53	–
	Sven (classic, rand. SVD)	10.13	2.58	6.69	66%	3.44	48.15	1.02
nanoGPT	Sven (Gram, hooks)	59.53	2.53	32.3	54%	27.3	614.2	1.02
	Sven (Gram, full J)	229.2	9.74	204	89%	25.6	2892	1
	Sven (Gram, chunked)	423.2	18	398	94%	25.7	2624	–
	Sven (classic, rand. SVD)	239.2	10.2	234	98%	5.05	2892	1.01
CIFAR-10 ResNet18	Sven (Gram, full J)	188	7.89	165	88%	22.8	22976	1
	Sven (Gram, chunked)	207.9	8.72	182	88%	25.9	6921	–
	Sven (classic, rand. SVD)	1041	43.7	907	87%	134	23030	1.11

for larger models the `hooks` capture is significantly lighter. In the case of CIFAR-10 + ResNet18, where the hooked capture is unavailable, the `chunked` capture saves a lot of memory relative to the full Jacobian instantiation. Chunked capture is also much less time efficient for the smaller models because the chunking introduces unnecessary overhead, but catches up to `full` for ResNet18. Finally, we observe that the "classic" algorithm – which computes a randomized SVD of the full $B \times P$ Jacobian instead of eigendecomposing its Gram matrix – is the clear loser in both memory and time efficiency for all of the larger models.

E.8 Remaining Challenges

The largest barriers to scaling Sven are the cases where efficient Gram capture is not possible. When it *is* possible, Sven is relatively close to parity with Adam. A capture that handles BatchNorm, weight tying, etc. without materializing blocks, or a training-time policy that removes the coupling without changing what is optimized, would be a natural direction to explore next.

F Additional Results

This appendix includes the full details of the scans presented in Sec. 4: every method's confirmation table, the ranking summary across all seven scans, the all-seed trajectories, and the 1D regression scan in full. The k and `rtol` sweeps at each scan's selected configuration are in Appendix G, next to the spectra that explain them.

All-seed trajectories. Fig. 4 redraws Fig. 1 with every method in each field rather than the five best. Fig. 5 shows similar results for the best configurations on the three *regression* tasks (1D, polynomial, MNIST), with a third column added for training loss. The 1D regression results are included here, rather than in the main text, because they do not differ significantly from the polynomial regression results (and have the largest between-seed variance).

Summary tables. Table 15 compactly summarizes the results of Sec. 4: Sven and seven baselines on the three headline tasks, with the cost columns from the standalone timing pass. The per-scan tables at the end of this appendix give the same for every method. Figure 6 gives every method's rank on every scan, on validation loss (which selection used) and on test loss (an outcome). Sven is consistently among the top 2 or 3 optimizers, except for the case of CIFAR-10 where it falls far behind the rest.

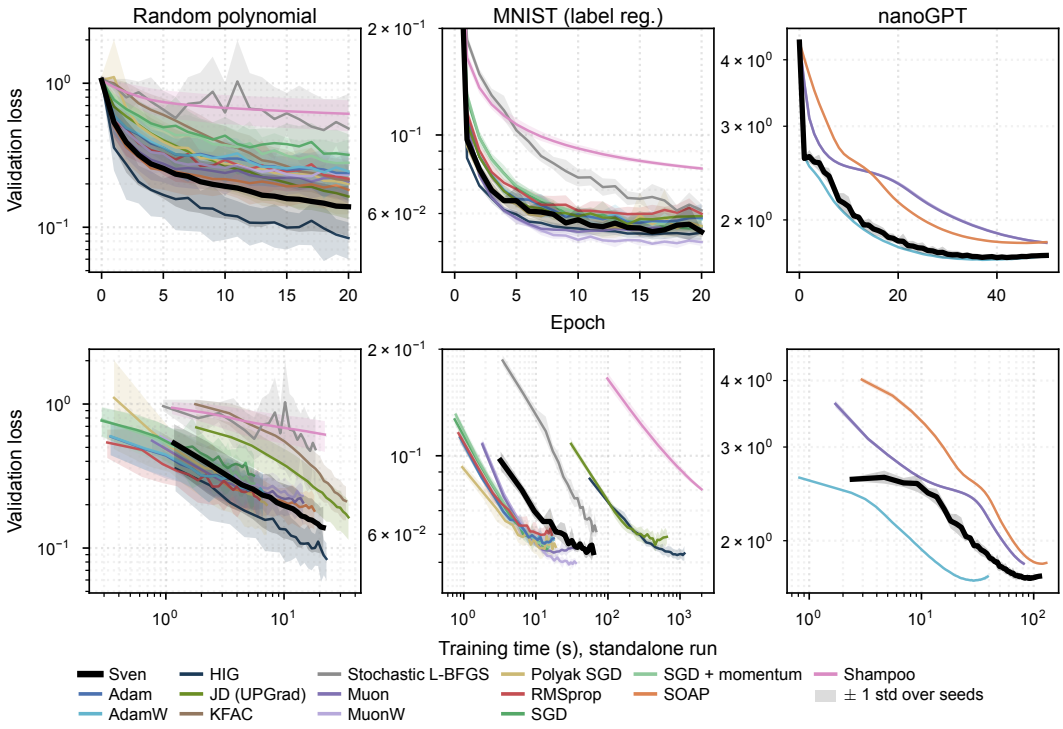

Figure 4: Reproduction of Fig. 1 with all baselines shown: validation loss against epoch (top) and against standalone wall time (bottom). Bands are ± 1 std over the 5 confirmation seeds. SOAP is omitted for MNIST because it does not converge well.

Data-seed replicates. The two synthetic tasks are each three different random problems, and the confirmation pass runs all three. Table 16 separates the two spreads: the standard deviation of the three per-instance means is variation of the target function, while the seed band within an instance is variation of the optimizer's luck. The ratio measures the between-problem variation against the within-problem variation (optimizer luck). For polynomial regression it is > 1, indicating that the problem instance dominates variance. In this paper we always quote the pooled losses across dataset and model seeds.

G Analyzing Singular-Value Spectra

Sven's approximation is the truncation controlled by k and `rtol`, so inspecting the spectrum of M during training illuminates how Sven is working. In this appendix we analyze spectra and their interaction with k / `rtol` in two settings: "online", which logs untruncated spectra periodically during training; and "probe" measurements, which use cached checkpoints and recompute Jacobians on a fixed held-out probe set. The online spectrum reflects states the optimizer saw during training, while the probe spectrum reflects the "general problem" – a held-out test set – rather than training instances. The results shown in this section expand upon Fig. 3 in the main text.

G.1 The spectra

Figure 7 draws each scan's spectrum over training for the best-performing configuration, highlighting the broad differences in shape. On the 1D task it falls steeply and lies mostly below the `rtol` floor, becoming slightly less steep over the course of training. The polynomial spectrum is less steep and retains its shape. MNIST CE and label regression (LR) become steeper over training, but the best `rtol` setting means the spectrum is heavily truncated for CE and barely truncated for LR. A similar trend is true for CIFAR, though its CE spectrum is relatively stable. This suggests that

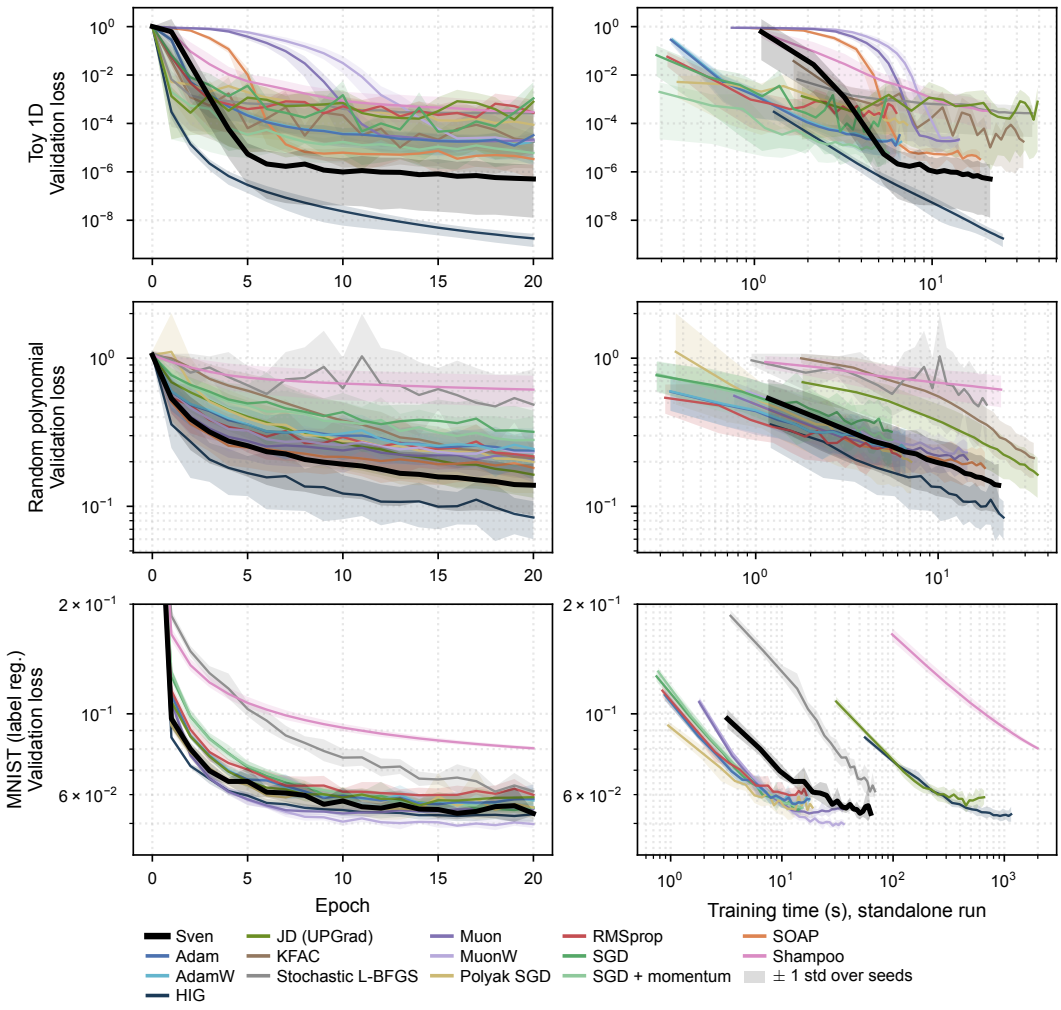

Figure 5: Confirmation trajectories for 1D regression (top), polynomial regression (middle) and MNIST label regression (bottom). We plot validation loss against epoch (left) and against standalone training time (right). Bands are mean ± 1 std over the 5 confirmation seeds. SOAP is omitted from MNIST because it does not converge well.

retaining directions with lower SVs is more important for performance on regression problems than on cross-entropy, but it is unclear why.

G.2 Effect of k and rtol

Truncation. Two knobs restrict the update: k and `rtol`. Implemented as in Appendix E.4, the number of directions the step inverts is

$$r_{\text{used}} \;=\; \min\big(k,\; \#\{i : \sigma_i > \texttt{rtol} \cdot \sigma_1\}\big). \tag{53}$$

It follows that the best-performing k cannot be read as a count of significant directions, because the spectra in Fig. 7 indicate it is often `rtol` that controls r_{used} (see Table 19 for a detailed accounting). Whether k or `rtol` truncates can also change during training: on CIFAR-10 label regression neither does at step 0 and `rtol` does at the end. A corollary for the tuning grids: a grid point whose binding cut is k produces the same trajectory as every tighter `rtol` at the same (k, η), and one whose binding cut is `rtol` the same as every larger k, so Sven's grid holds fewer distinct trajectories than points — 47.5% to 66.4% of the nominal grid across the four MLP scans — where a single-knob baseline has one trajectory per point.

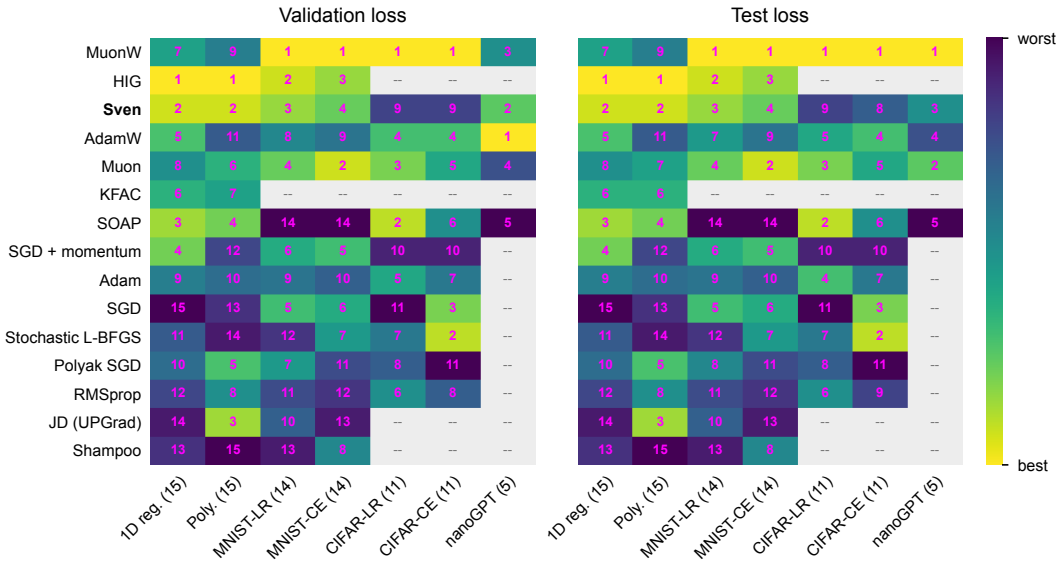

Figure 6: Ranking of the best-performing configurations of each optimizer on each scan using validation loss (left) and test loss (right). Numbers indicate rank, and colors are normalized by the number of contenders for each scan (indicated in the bottom row). A gray cell means the method was not run on that scan or produced no finite confirmation value.

Table 15: Headline results on the confirmation seeds: final validation and test loss (mean ± 1 std over seeds), test accuracy where defined, finished/attempted runs, seconds per epoch and peak memory. Losses are from the confirmation pass; the two cost columns are the standalone timing pass.

Method	Configuration	Val. loss	Val. (tune)	Test loss	Test acc.	fin./att.	s/ep.	MB
Random Polynomial								
Sven	η=0.5, k=16, rtol=0.03	**0.1388 ± 0.055**	0.1095	0.136 ± 0.055	–	15/15	1.33	18.86
HIG	η=0.1, tau=1e-08	**0.08406 ± 0.024**	0.07149	0.08028 ± 0.02	–	15/15	1.4	21.59
MuonW	η=0.01, wd=0.1	0.2313 ± 0.07	0.2061	0.2226 ± 0.06	–	15/15	0.967	18.44
Muon	η=0.01	0.207 ± 0.062	0.1651	0.202 ± 0.058	–	15/15	0.978	18.44
SOAP	η=0.01	0.1818 ± 0.05	0.1496	0.1756 ± 0.042	–	15/15	1.15	21.29
AdamW	η=0.01, wd=0.01	0.2442 ± 0.092	0.2054	0.2403 ± 0.09	–	15/15	0.58	18.44
Adam	η=0.01	0.2377 ± 0.087	0.1852	0.2283 ± 0.079	–	15/15	0.58	18.44
SGD	η=0.1	0.3182 ± 0.12	0.2263	0.315 ± 0.13	–	15/15	0.528	18.43
MNIST (label reg.)								
Sven	η=0.5, k=64, rtol=0.001	0.05328 ± 0.0007	0.05215	0.05363 ± 0.0024	96.7%	5/5	3.74	31.36
HIG	η=0.05, tau=0.03	0.05298 ± 0.0026	0.05076	0.05324 ± 0.0023	96.6%	5/5	58.3	404.3
MuonW	η=0.001, wd=0.1	**0.04982 ± 0.0013**	0.04999	0.04959 ± 0.0025	97.0%	5/5	2.4	31.59
Muon	η=0.001	0.05473 ± 0.0017	0.0558	0.05424 ± 0.0038	96.7%	5/5	2.34	31.59
SOAP	η=0.01	0.5808 ± 0.44	0.5708	0.5732 ± 0.45	44.5%	5/5	3.34	41.83
AdamW	η=0.001, wd=0.01	0.05784 ± 0.0048	0.05686	0.05613 ± 0.004	96.7%	5/5	1.46	31.7
Adam	η=0.001	0.05844 ± 0.0055	0.05683	0.057 ± 0.004	96.6%	5/5	1.48	31.7
SGD	η=0.1	0.05524 ± 0.0025	0.05338	0.05439 ± 0.0019	96.9%	5/5	1.33	31.48
nanoGPT (tiny-shakespeare)								
Sven	η=0.1, k=64, rtol=0.001	**1.715 ± 0.008**	1.724	1.977 ± 0.024	–	5/5	2.4	615.4
MuonW	η=0.0001, wd=0.1	1.81 ± 0.0025	1.818	1.973 ± 0.0061	–	5/5	1.75	326.5
Muon	η=0.0001	1.812 ± 0.0023	1.819	1.976 ± 0.0058	–	5/5	1.76	326.5
SOAP	η=0.0001	1.817 ± 0.0062	1.822	2.033 ± 0.0092	–	5/5	2.73	352.3
AdamW	η=0.0003, wd=0.01	**1.713 ± 0.018**	1.714	1.978 ± 0.02	–	5/5	0.914	329.7

Sweeps Fig. 8 sweeps k with η and `rtol` pinned at each scan's selected values, and Fig. 9 shows the seed-mean final loss over k at fixed `rtol` and over `rtol` at fixed k. Improvement saturates

Table 16: Data-seed replicates on the two synthetic tasks. Each data seed is a different random problem, so the standard deviation of the three per-instance means is variation of the target and not of the optimizer. A ratio above one says the choice of instance moves the answer more than the seed does.

Method	Val. (pooled)	Val. (inst. mean)	Betw.-inst. std	Within-inst. std	Ratio
Toy 1D					
HIG	1.775×10^{-9}	1.775×10^{-9}	6.39×10^{-10}	8.08×10^{-10}	0.791
Sven	5.086×10^{-7}	5.086×10^{-7}	6.45×10^{-7}	9.45×10^{-7}	0.682
SOAP	3.372×10^{-6}	3.372×10^{-6}	9.89×10^{-7}	3.11×10^{-6}	0.318
SGD + momentum	6.027×10^{-6}	6.027×10^{-6}	2.06×10^{-6}	3.17×10^{-6}	0.652
AdamW	1.735×10^{-5}	1.735×10^{-5}	9.65×10^{-6}	1.47×10^{-5}	0.657
KFAC	1.802×10^{-5}	1.802×10^{-5}	1.30×10^{-5}	1.48×10^{-5}	0.878
MuonW	1.928×10^{-5}	1.928×10^{-5}	1.82×10^{-6}	9.34×10^{-6}	0.195
Muon	2.159×10^{-5}	2.159×10^{-5}	8.43×10^{-6}	1.70×10^{-5}	0.495
Adam	3.264×10^{-5}	3.264×10^{-5}	2.66×10^{-5}	4.53×10^{-5}	0.587
Polyak SGD	8.792×10^{-5}	8.792×10^{-5}	2.68×10^{-5}	0.000104	0.257
Stochastic L-BFGS	0.0002659	0.0002644	6.93×10^{-5}	0.000305	0.227
RMSprop	0.0002746	0.0002746	0.000262	0.000529	0.495
Shampoo	0.0002956	0.0002956	0.000155	0.000346	0.449
JD (UPGrad)	0.0007893	0.0007893	0.00103	0.00153	0.675
SGD	0.001084	0.001084	0.00177	0.00233	0.761
Random Polynomial					
HIG	0.08406	0.08406	0.0268	0.00816	3.28
Sven	0.1388	0.1388	0.057	0.0259	2.2
JD (UPGrad)	0.1635	0.1635	0.0524	0.0218	2.4
SOAP	0.1818	0.1818	0.0507	0.0243	2.09
Polyak SGD	0.1997	0.1997	0.0954	0.0333	2.87
Muon	0.207	0.207	0.0682	0.0237	2.88
KFAC	0.2118	0.2118	0.0605	0.0241	2.51
RMSprop	0.2173	0.2173	0.0608	0.0225	2.71
MuonW	0.2313	0.2313	0.0782	0.022	3.56
Adam	0.2377	0.2377	0.0912	0.0373	2.45
AdamW	0.2442	0.2442	0.0808	0.0579	1.4
SGD + momentum	0.2805	0.2805	0.137	0.0447	3.07
SGD	0.3182	0.3182	0.133	0.0518	2.56
Stochastic L-BFGS	0.4855	0.4855	0.359	0	–
Shampoo	0.6137	0.6137	0.166	0.0531	3.12

at $k = 16$ (1D), 16 (polynomial), 32 (MNIST label regression) and 8 (MNIST cross-entropy): a different fraction of B on each scan corresponding to the point where `rtol` starts to truncate, past which raising k changes nothing. The `rtol` panels read the same way against the spectra: on the 1D task, whose spectrum is steepest, the loss moves by orders of magnitude across the `rtol` grid; on MNIST LR, whose spectrum is flatter, most `rtol` lines coincide. On the polynomial task a looser cut *helps*, up to the selected `rtol`, and the apparent preference for a large k at a tight `rtol` is the interaction of the two cuts rather than the value of additional directions.

Fig. 10 extends the same measurement to the whole tuning grid: at each $(k, \texttt{rtol})$ it shows the used rank as a fraction of k at the selected learning rate, which makes the region where k truncates and the region where `rtol` truncates visible at once.

G.3 Effect of truncation

For each logged batch we compute the fraction of the residuals $\|\tilde{U}^T\mathcal{R}\|^2$ carried by the directions the truncation *keeps*, and track it over training: Fig. 11(a) draws it along with the used rank (c), and Table 19 gives the first- and last-step values. The fraction is consistently 100% on MNIST LR, CIFAR LR, and nanoGPT, indicating that the discarded directions carry essentially none of the residual. MNIST LR and nanoGPT use the entire spectrum throughout training, while for the others `rtol` does reduce the rank. The truncation thus discards something tangible on some scans, but with an

Table 17: Every method's selected configuration on the confirmation seeds, 1D regression (top) and random-polynomial regression (bottom). Rows are sorted by validation loss, so the row order is the ranking; the band is ± 1 std over seeds.

Method	Selected configuration	Val. loss	Test loss	Train (eval)
HIG	lr=0.005, tau=1e-06	$(1.775 \pm 0.99) \times 10^{-9}$	$(1.886 \pm 1.1) \times 10^{-9}$	$(1.811 \pm 1.0) \times 10^{-9}$
Sven	k=32, lr=0.01, rtol=0.0001	$(5.086 \pm 15) \times 10^{-7}$	$(5.358 \pm 16) \times 10^{-7}$	$(5.075 \pm 15) \times 10^{-7}$
SOAP	lr=0.001	$(3.372 \pm 3.1) \times 10^{-6}$	$(3.347 \pm 3.1) \times 10^{-6}$	$(3.370 \pm 3.1) \times 10^{-6}$
SGD + momentum	lr=0.1	$(6.027 \pm 3.8) \times 10^{-6}$	$(6.040 \pm 3.7) \times 10^{-6}$	$(6.040 \pm 3.8) \times 10^{-6}$
AdamW	lr=0.001, weight_decay=0.01	$(1.735 \pm 1.7) \times 10^{-5}$	$(1.736 \pm 1.8) \times 10^{-5}$	$(1.738 \pm 1.7) \times 10^{-5}$
KFAC	lr=0.001	$(1.802 \pm 1.9) \times 10^{-5}$	$(1.795 \pm 1.9) \times 10^{-5}$	$(1.798 \pm 1.9) \times 10^{-5}$
MuonW	lr=0.001, weight_decay=0.1	$(1.928 \pm 0.90) \times 10^{-5}$	$(1.919 \pm 0.89) \times 10^{-5}$	$(1.923 \pm 0.90) \times 10^{-5}$
Muon	lr=0.001	$(2.159 \pm 2.0) \times 10^{-5}$	$(2.178 \pm 2.1) \times 10^{-5}$	$(2.165 \pm 2.0) \times 10^{-5}$
Adam	lr=0.001	$(3.264 \pm 5.8) \times 10^{-5}$	$(3.320 \pm 6.0) \times 10^{-5}$	$(3.290 \pm 5.9) \times 10^{-5}$
Polyak SGD	f_star=0, max_lr=1, eps=1e-08	$(8.792 \pm 11) \times 10^{-5}$	$(8.806 \pm 11) \times 10^{-5}$	$(8.748 \pm 11) \times 10^{-5}$
Stochastic L-BFGS	lr=0.03, max_iter=3, history_size=10	$(2.659 \pm 2.9) \times 10^{-4}$	$(2.667 \pm 2.9) \times 10^{-4}$	$(2.670 \pm 3.0) \times 10^{-4}$
RMSprop	lr=0.001	$(2.746 \pm 6.6) \times 10^{-4}$	$(2.750 \pm 6.6) \times 10^{-4}$	$(2.759 \pm 6.6) \times 10^{-4}$
Shampoo	lr=0.1	$(2.956 \pm 4.5) \times 10^{-4}$	$(3.016 \pm 4.7) \times 10^{-4}$	$(2.991 \pm 4.7) \times 10^{-4}$
JD (UPGrad)	lr=0.01, aggregator=UPGrad, inner_optimizer=Adam	$(7.893 \pm 24) \times 10^{-4}$	$(7.829 \pm 23) \times 10^{-4}$	$(7.922 \pm 24) \times 10^{-4}$
SGD	lr=0.3	$(1.084 \pm 4.0) \times 10^{-3}$	$(1.083 \pm 3.9) \times 10^{-3}$	$(1.095 \pm 4.0) \times 10^{-3}$

Method	Selected configuration	Val. loss	Test loss	Train (eval)
HIG	lr=0.1, tau=1e-08	0.08406 ± 0.024	0.08028 ± 0.02	0.05783 ± 0.014
Sven	k=16, lr=0.5, rtol=0.03	0.1388 ± 0.055	0.136 ± 0.055	0.1106 ± 0.042
JD (UPGrad)	lr=0.001, aggregator=UPGrad, inner_optimizer=Adam	0.1635 ± 0.049	0.1529 ± 0.044	0.1234 ± 0.038
SOAP	lr=0.01	0.1818 ± 0.05	0.1756 ± 0.042	0.1283 ± 0.031
Polyak SGD	f_star=0, max_lr=1, eps=1e-08	0.1997 ± 0.089	0.1924 ± 0.081	0.1457 ± 0.067
Muon	lr=0.01	0.207 ± 0.062	0.202 ± 0.058	0.1474 ± 0.04
KFAC	lr=1e-05	0.2118 ± 0.056	0.2013 ± 0.048	0.1626 ± 0.041
RMSprop	lr=0.01	0.2173 ± 0.056	0.2108 ± 0.047	0.1539 ± 0.032
MuonW	lr=0.01, weight_decay=0.1	0.2313 ± 0.07	0.2226 ± 0.06	0.1873 ± 0.051
Adam	lr=0.01	0.2377 ± 0.087	0.2283 ± 0.079	0.1696 ± 0.061
AdamW	lr=0.01, weight_decay=0.01	0.2442 ± 0.092	0.2403 ± 0.09	0.1872 ± 0.078
SGD + momentum	lr=0.01	0.2805 ± 0.13	0.2723 ± 0.11	0.2154 ± 0.1
SGD	lr=0.1	0.3182 ± 0.12	0.315 ± 0.13	0.259 ± 0.12
Stochastic L-BFGS	lr=1, max_iter=1, history_size=2	0.4855 ± 0.36	0.4485 ± 0.33	0.4123 ± 0.34
Shampoo	lr=0.1	0.6137 ± 0.15	0.6064 ± 0.13	0.5699 ± 0.14

inconsistent effect on the fraction of retained residuals. It *rises* on the polynomial scan (92.3% to 99.6%) and MNIST CE (31% to 95.2%), i.e. the residual concentrates into the directions that survive the truncation. It falls marginally on the 1D scan (92.1% to 91.5%) and substantially on CIFAR CE (90.1% to 76.7%), where by the end of training roughly a quarter of the batch residual lies in directions the update throws away. This effect is clearly dataset-dependent, and the scan where we truncate the highest residual fraction is where Sven performs the worst (CIFAR-10).

Table 18: Every method's selected configuration on the confirmation seeds, MNIST label regression (top) and MNIST cross-entropy (bottom). Conventions as in Table 17. K-FAC is absent from both: it has no eligible configuration on either scan (Appendix Q).

Method	Selected configuration	Val. loss	Test loss	Test acc.	Train (eval)
MuonW	lr=0.001, weight_decay=0.1	0.04982 ± 0.0013	0.04959 ± 0.0025	97.03%	0.02206 ± 0.00041
HIG	lr=0.05, tau=0.03	0.05298 ± 0.0026	0.05324 ± 0.0023	96.60%	0.02293 ± 0.0019
Sven	k=64, lr=0.5, rtol=0.001	0.05328 ± 0.0007	0.05363 ± 0.0024	96.72%	0.02804 ± 0.0012
Muon	lr=0.001	0.05473 ± 0.0017	0.05424 ± 0.0038	96.72%	0.01652 ± 0.0011
SGD	lr=0.1	0.05524 ± 0.0025	0.05439 ± 0.0019	96.89%	0.02968 ± 0.0019
SGD + momentum	lr=0.01	0.05524 ± 0.0015	0.05454 ± 0.0017	96.95%	0.02863 ± 0.0011
Polyak SGD	f_star=0, max_lr=1, eps=1e-08	0.05532 ± 0.004	0.05694 ± 0.0031	96.59%	0.02319 ± 0.0026
AdamW	lr=0.001, weight_decay=0.01	0.05784 ± 0.0048	0.05613 ± 0.004	96.69%	0.02526 ± 0.0029
Adam	lr=0.001	0.05844 ± 0.0055	0.057 ± 0.004	96.59%	0.02432 ± 0.0033
JD (UPGrad)	lr=0.001, aggregator=UPGrad, inner_optimizer=Adam	0.05902 ± 0.0025	0.05738 ± 0.0042	96.46%	0.02461 ± 0.0029
RMSprop	lr=0.001	0.05989 ± 0.0015	0.05897 ± 0.0018	96.43%	0.02781 ± 0.0024
Stochastic L-BFGS	lr=0.1, max_iter=3, history_size=2	0.06127 ± 0.0025	0.0609 ± 0.0026	96.64%	0.04508 ± 0.0025
Shampoo	lr=0.1	0.0804 ± 0.00099	0.08082 ± 0.0012	95.79%	0.0612 ± 0.0018
SOAP	lr=0.01	0.5808 ± 0.44	0.5732 ± 0.45	44.45%	0.555 ± 0.47

Method	Selected configuration	Val. loss	Test loss	Test acc.	Train (eval)
MuonW	lr=0.0001, weight_decay=0.1	0.1046 ± 0.0038	0.1032 ± 0.0049	96.90%	0.06315 ± 0.0024
Muon	lr=0.0001	0.1067 ± 0.004	0.1054 ± 0.0052	96.88%	0.05821 ± 0.0023
HIG	lr=0.05, tau=0.03	0.1119 ± 0.0031	0.1094 ± 0.0039	96.73%	0.04896 ± 0.0032
Sven	k=32, lr=0.5, rtol=0.3	0.1134 ± 0.0051	0.1128 ± 0.0045	96.64%	0.05013 ± 0.0057
SGD + momentum	lr=0.001	0.1249 ± 0.0076	0.1206 ± 0.0052	96.22%	0.08399 ± 0.0071
SGD	lr=0.01	0.1258 ± 0.013	0.1219 ± 0.014	96.24%	0.08522 ± 0.014
Stochastic L-BFGS	lr=0.5, max_iter=1, history_size=2	0.1307 ± 0.0088	0.125 ± 0.0056	96.36%	0.08887 ± 0.012
Shampoo	lr=0.1	0.1317 ± 0.0031	0.1271 ± 0.0063	96.13%	0.06502 ± 0.005
AdamW	lr=0.001, weight_decay=0.01	0.1368 ± 0.0027	0.1285 ± 0.0063	96.64%	0.03307 ± 0.0032
Adam	lr=0.0001	0.1456 ± 0.0068	0.145 ± 0.0081	95.73%	0.1221 ± 0.0063
Polyak SGD	f_star=0, max_lr=1, eps=1e-08	0.1471 ± 0.0015	0.145 ± 0.0088	96.81%	0.02024 ± 0.003
RMSprop	lr=0.0001	0.153 ± 0.01	0.1496 ± 0.0093	95.51%	0.1301 ± 0.01
JD (UPGrad)	lr=0.0001, aggregator=UPGrad, inner_optimizer=Adam	0.1591 ± 0.011	0.1568 ± 0.011	95.62%	0.122 ± 0.0086
SOAP	lr=0.0001	0.646 ± 1.1	0.6406 ± 1.1	77.88%	0.6019 ± 1.1

G.4 Probe-Set Analysis

As discussed above, we save checkpoints from the training trajectories of four optimizers (Sven, Adam, MuonW, and HIG), and compute double-precision Jacobians on a probe dataset and analyze their singular value spectra. The spectrum width is $\min(\text{probe rows}, P)$, and the settings for the four probed datasets are shown in Table 20.

Probe-set Residuals Fig. 13 and Table 21 summarize the probe results for Sven: the fraction of the reachable probe residual that lies in the leading directions of the probe Jacobian, with the selected k marked. On the 1D task and both MNIST scans the top-k directions carry nearly all of it; on the polynomial task they carry around 50%. In the online measurement, which measured the kept fraction of a *batch-level* residual, we saw that the selected k kept essentially 100% of the residual. The probe measurement answers a different question: how much of the whole problem's residual any rank-k

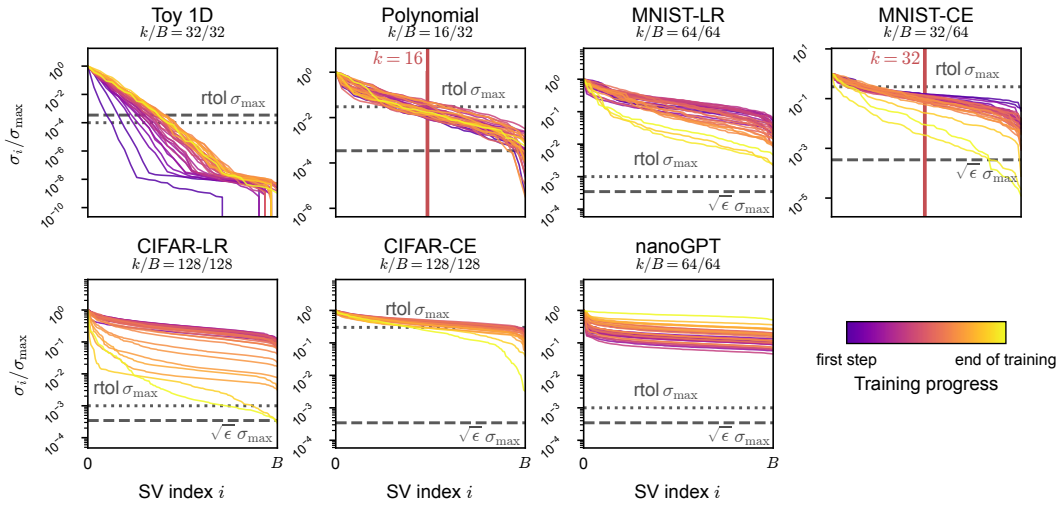

Figure 7: Full-width online Gram spectra over training (one model seed per scan). Each line is from one logged batch and normalized by its σ_1; the vertical line indicates k when $k < B$, the dotted line the `rtol` cut, and the dashed line the float32 round-off floor.

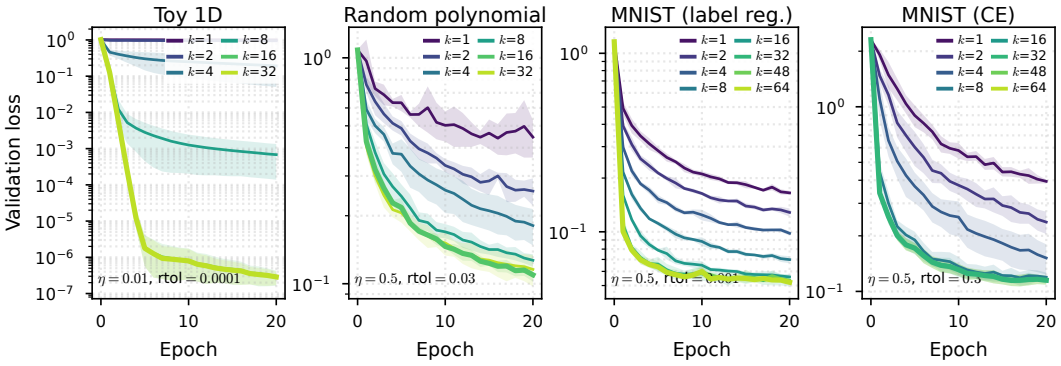

Figure 8: Validation loss against epoch over the rank cutoff k, with η and `rtol` pinned at each scan's selected values; the thick curve is the selected k.

update can address. In this case, the batch size is the upper truncation limit, as the step lives in the batch's row space, so even at $k = B$ it can reach no more than the curve's value at B. The top-k directions of the probe Jacobian are also the best k for the *whole* set, not the directions one might obtain from a randomly sampled batch. The curve in Fig. 13 at k is thus an upper bound on what a step addresses. What the probe measurement establishes is therefore not that the truncation discards signal at each step but that a rank-k, batch-B update is a local correction to a residual that generally needs more directions to describe. This is most evident on the polynomial task, which is also where Sven has a larger margin over the first-order optimizers.

Other optimizers Fig. 14 shows the probe spectra along the trajectories of Sven, Adam, MuonW and HIG on every cached scan, one named seed per scan. Fig. 15 plots the resolved rank, the effective rank, the condition number and σ_B/σ_1 against training step for every cached scan, and Fig. 16 the distance from initialization and the parameter norm. Two notions of rank appear in these figures and tables. The *resolved rank* is the number of singular values above $10^{-12}\,\sigma_{\max}$, the limit for a float64 spectrum computation; it is a count at a threshold, and the condition number is taken over the resolved part only. The *effective rank* is the entropy rank $\exp(-\sum_i p_i \log p_i)$ with $p_i = \sigma_i / \sum_j \sigma_j$: it equals r for a flat spectrum of r equal singular values and falls smoothly as the spectrum concentrates. For a very hierarchical spectrum, it is far below the resolved rank.

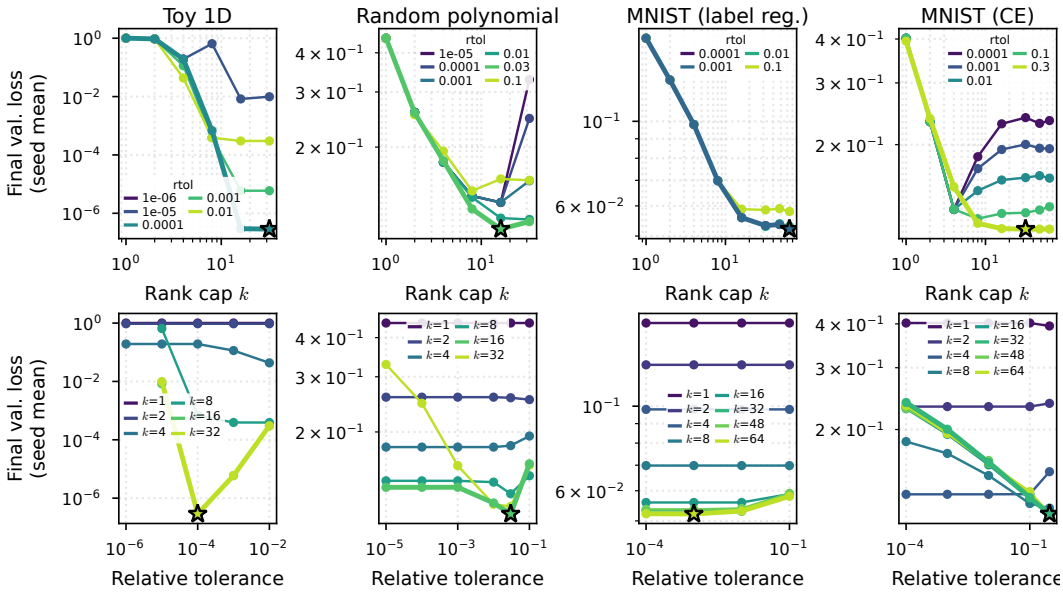

Figure 9: Mean final validation loss over k at fixed `rtol` (top) and over `rtol` at fixed k (bottom), both at each scan's selected learning rate; the star is the selection of record and ineligible configurations are omitted.

Table 19: What the truncation does, per scan. "Used" is the number of directions actually inverted, $\min(k, \texttt{rtol}\text{-rank})$, at the first and last logged step; "Kept" is the fraction of that batch's residual they account for. "Binds" is read at the last logged step. Seed means over the diagnostics pass.

Scan	k / B	**rtol**	**Binds**	**Used, first → final**	**Kept, first → final**
Toy 1D	32 / 32	10^{-4}	rtol	4.2 → 14.4	92.08% → 91.53%
Polynomial	16 / 32	0.03	rtol	7.0 → 13.4	92.25% → 99.59%
MNIST-LR	64 / 64	10^{-3}	neither ($k = B$)	64.0 → 63.6	100.00% → 100.00%
MNIST-CE	32 / 64	0.3	rtol	10.0 → 2.8	31.01% → 95.17%
CIFAR-LR	128 / 128	10^{-3}	rtol	128.0 → 81.6	100.00% → 100.00%
CIFAR-CE	128 / 128	0.3	rtol	106.0 → 36.6	90.13% → 76.69%
nanoGPT	64 / 64	10^{-3}	neither ($k = B$)	64.0 → 64.0	100.00% → 100.00%

We observe that for the toy 1D dataset, the resolved rank is much smaller than $\min(\text{probe rows}, P)$ and that the condition number and σ_B/σ_1 get very near the float64 cutoff. This reflects the problem's strongly hierarchical spectrum. For the other datasets, the resolved rank typically equals $\min(\text{probe rows}, P)$. In all cases the effective rank is significantly smaller, reflecting varying degrees of hierarchy in the spectra. The trends differ for 1D/polynomial versus MNIST, however: the effective rank trends upwards in the synthetic task, indicating a flattening spectrum, while it trends down for both MNIST tasks, reflecting steepening. The overall trends, measured by beginning and endpoint values, are remarkably similar for all the optimizers.

Fig. 16 examines the impact of Sven's update being the minimum-norm solution of the linearized system at each step, which invites the conjecture that it might stay closer to its initialization, or reach a smaller-norm solution than a first-order method. This does not appear to be true in general, with Sven's parameter norm ending larger than Adam's in some cases and smaller in others. Sven's parameter norm does tend to grow faster early in training, however. Table 22 compares Sven's parameter norm with other baselines at the last saved checkpoint. It lists Sven's mean parameter norm over seeds, compares it to the best (smallest) norm obtained by other optimizers, and shows the mean norm difference between Sven and the best other baseline. As in Fig. 16, the results are inconclusive, but it is clear that Sven does not generally result in a lower parameter norm.

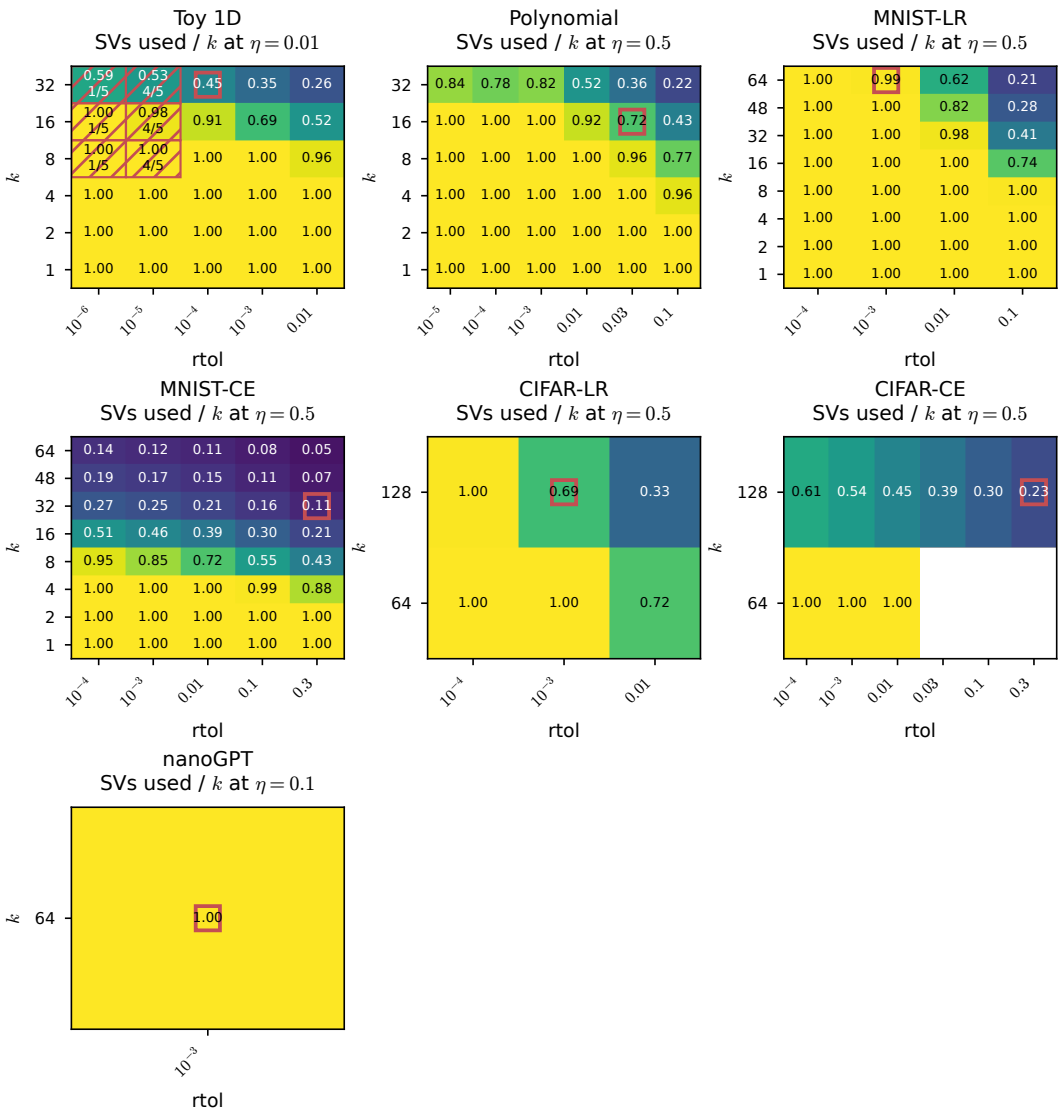

Figure 10: The rank the update uses as a fraction of the cap k, over the tuning $(k, \texttt{rtol})$ grid at each scan's selected learning rate, final epoch. A cell at 1 is a cell where k truncates; anything below it is a cell where `rtol` truncates. The red square is the selection of record and a hatched cell indicates fewer seeds than the grid intended (failed runs).

Table 20: The cached probe sets. The spectrum width is $\min(\text{probe rows}, P)$ and only one of the two bounds is ever active. "Resolved rank" counts the singular values above the double-precision threshold on Sven's final checkpoint, averaged over 5 model seeds.

Scan	**Probe rows**	P	**Spectrum width**	**Resolved rank, final step**
Toy 1D	10,000	593	593	45.0
Random Polynomial	10,000	673	673	673.0
MNIST (label reg.)	512	27,562	512	512.0
MNIST (CE)	512	27,562	512	512.0

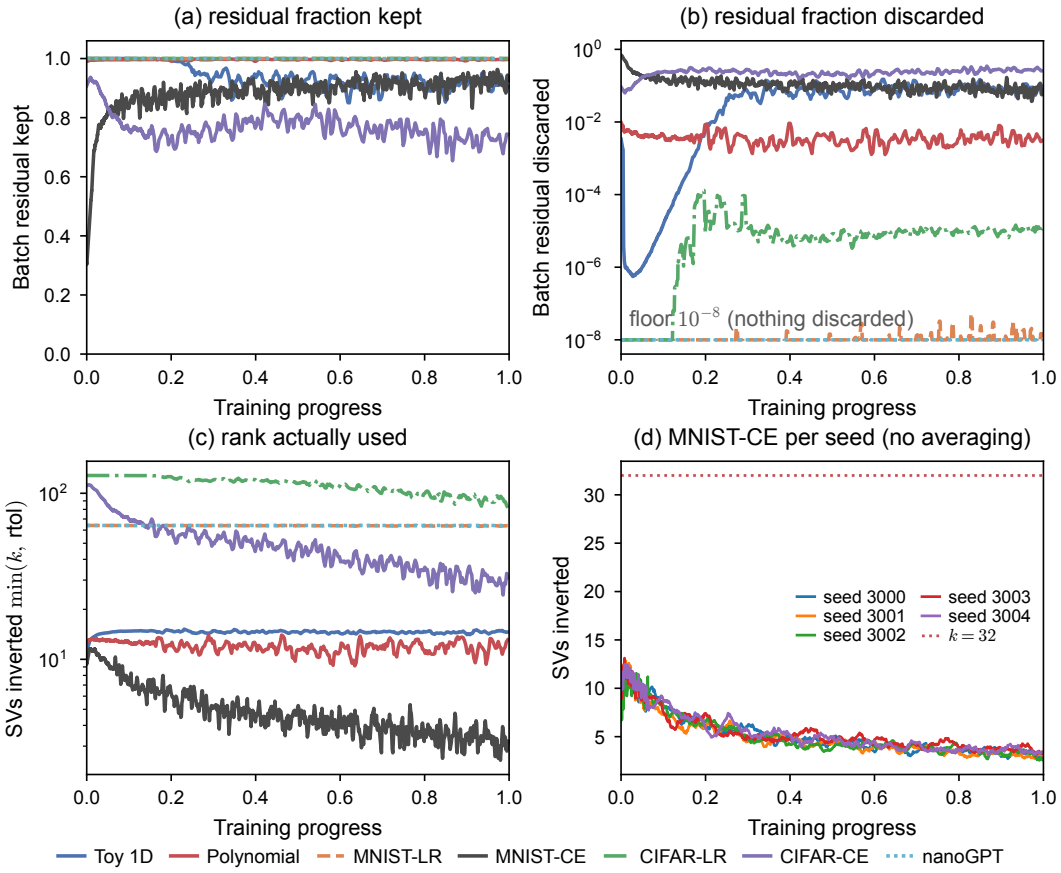

Figure 11: (a)–(c) Kept and discarded fraction of the batch residual and the rank actually inverted, against training progress, seed mean over the diagnostics seeds with a 60-step moving average. (d) The five seeds of the MNIST cross-entropy scan drawn separately: the collapse to ~ 3 directions happens on every seed.

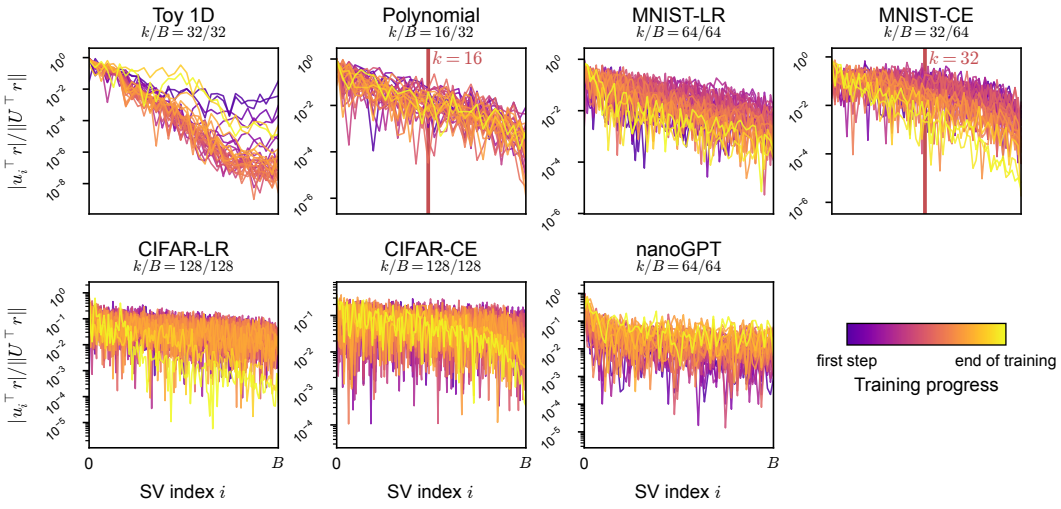

Figure 12: Where the batch residual sits: $|u_i \cdot R|/\|U^T R\|$ against singular-value index over training, one seed per panel.

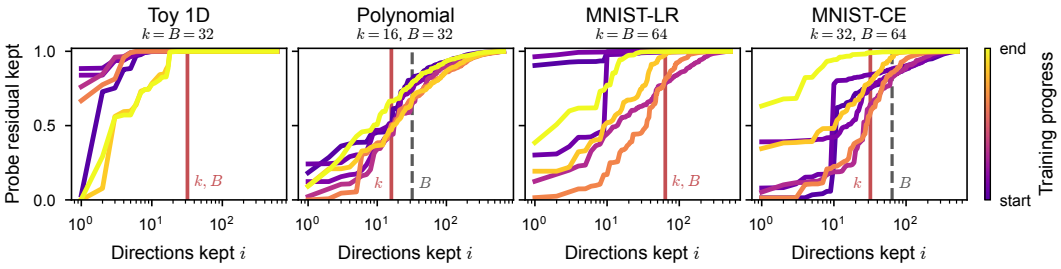

Figure 13: Cumulative fraction of the reachable probe residual against the number of leading probe directions kept, one seed per scan, colored by checkpoint. Two cuts are marked: k is the rank a step inverts, B the row space a step lives in, so even an untruncated step reaches no more than the curve's value at B. The x axis runs to $\min(\text{probe rows}, P)$.

Table 21: Residual fraction inside the top-k probe directions, offline and in double precision, averaged over 5 model seeds; fractions are of the part of the probe residual the full Jacobian column space can reach, which is $\geq 99.5\%$ on every scan. An asterisk marks the selected k.

Scan	**Checkpoint**	k	**Top-k fraction (%)**	**Seed std (%)**
Toy 1D	first	8	99.6	0.2
Toy 1D	first	16	100.0	0.0
Toy 1D	first	32 *	100.0	0.0
Toy 1D	last	8	64.2	13.0
Toy 1D	last	16	76.9	10.6
Toy 1D	last	32 *	100.0	0.0
Random Polynomial	first	8	36.9	4.3
Random Polynomial	first	16 *	55.1	5.9
Random Polynomial	first	32	79.1	1.2
Random Polynomial	last	8	43.6	13.8
Random Polynomial	last	16 *	58.8	10.2
Random Polynomial	last	32	73.7	5.9
MNIST (label reg.)	first	8	91.8	2.7
MNIST (label reg.)	first	16	99.7	0.2
MNIST (label reg.)	first	32	99.9	0.1
MNIST (label reg.)	first	64 *	99.9	0.0
MNIST (label reg.)	last	8	60.9	11.7
MNIST (label reg.)	last	16	78.6	14.7
MNIST (label reg.)	last	32	93.8	8.0
MNIST (label reg.)	last	64 *	99.9	0.1
MNIST (CE)	first	8	4.7	1.5
MNIST (CE)	first	16	64.9	6.1
MNIST (CE)	first	32 *	76.5	1.8
MNIST (CE)	first	64	83.9	1.2
MNIST (CE)	last	8	79.8	9.9
MNIST (CE)	last	16	89.0	7.5
MNIST (CE)	last	32 *	96.1	3.2
MNIST (CE)	last	64	99.6	0.2

H Paired Comparisons

This appendix reports the per-seed paired differences between Sven and baseline optimizers. Every optimizer at a given model seed starts from the same initialization and sees the same batches in the

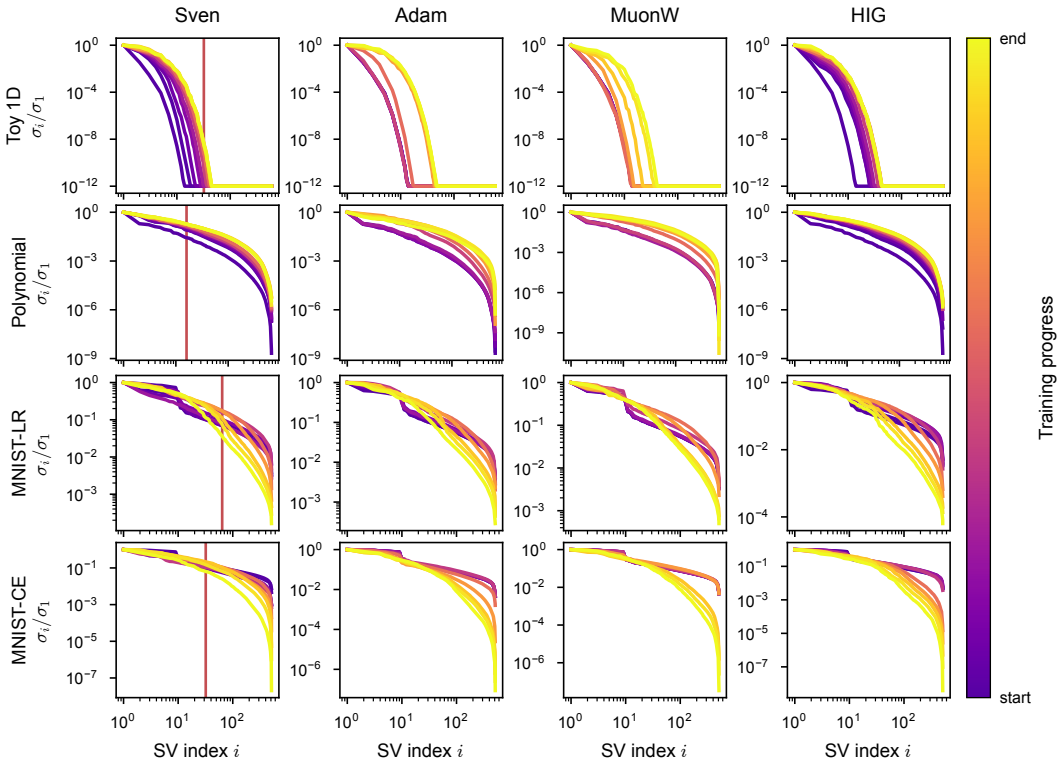

Figure 14: Probe-set spectra in double precision along the trajectories of four optimizers (columns) on every cached scan (rows), using the same probe rows at every checkpoint/optimizer. The red line is Sven's k. The spectrum width is $\min(\text{probe rows}, P)$.

Table 22: Parameter norm change (top panel) and absolute value (bottom) for Sven and the lowest of the other three optimizers, paired by model seed. A bold paired difference is one whose 95% interval excludes zero.

Scan	Sven	Best other mean	Rank	Paired diff.	95% interval	Seeds lower
$\|\theta_t - \theta_0\|$						
Toy 1D	7.82 ± 1.3	6.259 (MuonW)	3 of 4	1.56 ± 1.4	$[-0.196, 3.32]$	0/5
Random Polynomial	7.524 ± 0.71	8.12 (HIG)	1 of 4	-0.597 ± 0.58	$[-1.32, 0.127]$	4/5
MNIST (label reg.)	12.48 ± 0.27	12.52 (HIG)	1 of 4	-0.0423 ± 0.16	$[-0.239, 0.155]$	3/5
MNIST (CE)	9.475 ± 0.77	7.69 (HIG)	3 of 4	$\mathbf{1.79 \pm 0.8}$	$[0.786, 2.78]$	0/5
$\|\theta_t\|$						
Toy 1D	9.316 ± 1.1	7.62 (MuonW)	3 of 4	$\mathbf{1.7 \pm 1.2}$	$[0.263, 3.13]$	0/5
Random Polynomial	8.545 ± 0.37	9.134 (HIG)	1 of 4	$\mathbf{-0.59 \pm 0.27}$	$[-0.923, -0.256]$	5/5
MNIST (label reg.)	13.33 ± 0.21	13.24 (HIG)	2 of 4	0.0848 ± 0.18	$[-0.142, 0.311]$	1/5
MNIST (CE)	11.8 ± 0.53	10.59 (Adam)	3 of 4	$\mathbf{1.21 \pm 0.57}$	$[0.501, 1.92]$	0/5

same order, so the difference between two methods can be taken within a seed, which removes the seed-to-seed variation. Fig. 17 shows the mean paired difference in final validation between Sven and the baseline optimizers, along with its 95% t-interval. This elucidates where Sven truly is a better or worse choice than the baselines, and supports the observations in the main text from Figs. 1 and 2.

I Dataset-Level Overparametrization

Section 2 distinguishes two notions of overparametrization: $P > B$, which holds everywhere, and $P > N$, a property of the dataset. The theory that motivates Sven concerns the singularity of the natural-gradient metric, which is a P versus N statement. In this appendix we explicitly vary P/N, where N is the size of the training set (the validation and test set fixed).

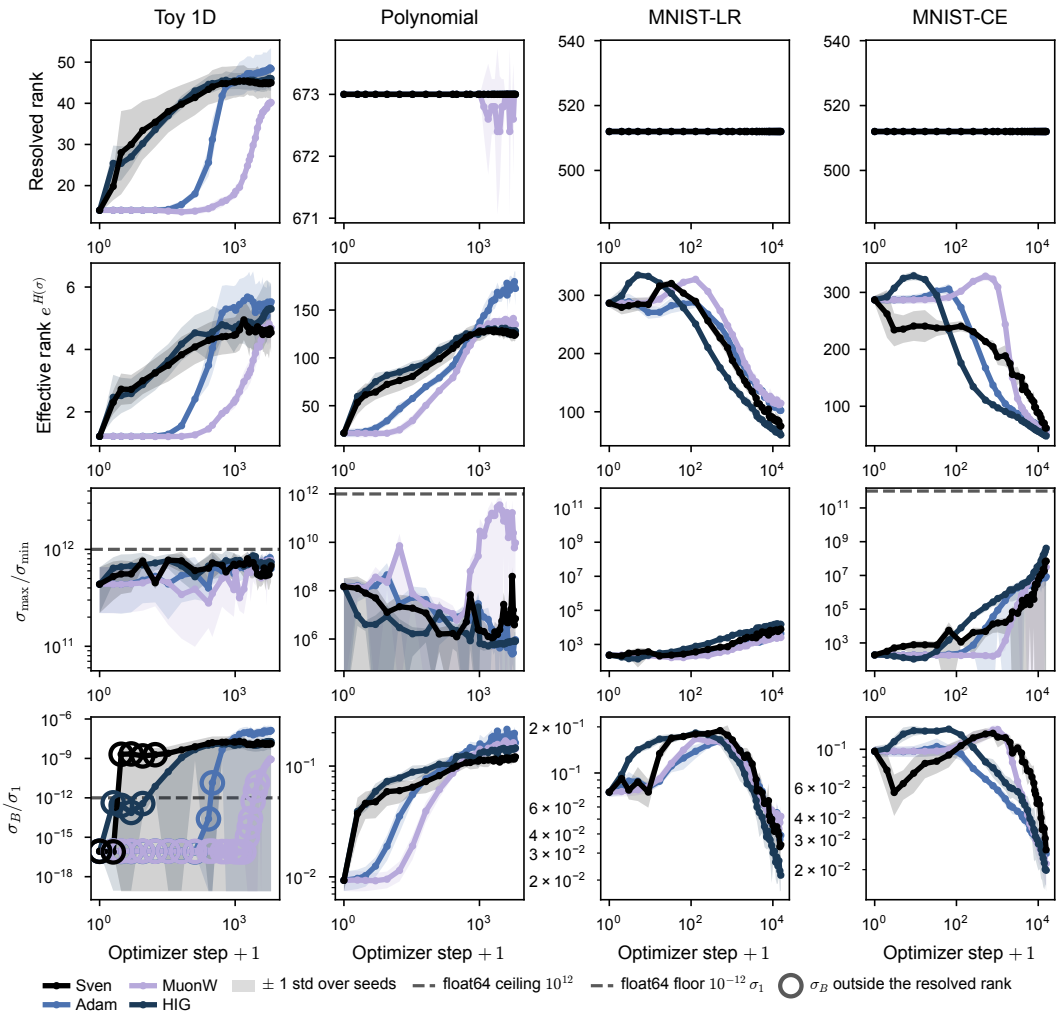

Figure 15: Resolved rank, effective rank, condition number and σ_B/σ_1 of the probe Jacobian along the trajectories of Sven, Adam, MuonW and HIG, seed mean $\pm$ 1 std over five cached seeds. The condition number is taken over the resolved part of the spectrum and cannot exceed 10^{12}, the inverse of the float64 threshold; a hollow marker means index B lies outside the resolved rank, so the ratio beside it is a threshold artifact. On the 1D task only 14 of 593 directions are resolved at initialization, which is why that scan's condition number and σ_B/σ_1 are artifacts throughout.

We study 1D regression, polynomial regression, and MNIST label regression at varying levels of P/N spanning under- and overparametrized. The synthetic datasets are trained full-batch, while MNIST subsamples the training set and keeps its batch size from earlier experiments[13]. Thirteen optimizers[14] are trained for 200 epochs on a similar hyperparameter grid to the studies in the main text, with best configurations selected via validation loss.

Fig. 18 plots train, validation, and test loss versus P/N for the best-performing configuration of each optimizer. Sven generally follows the same patterns as the baselines and more or less maintains its relative rank; there is no clear evidence that it has an advantage in dataset-level overparametrized settings.

[13]The MNIST MLP is much larger than the MLP for the synthetic datasets, so full-batch training is not feasible in e.g. the overparametrized setting.

[14]Jacobian Descent and K-FAC are omitted due to previously observed training instability.

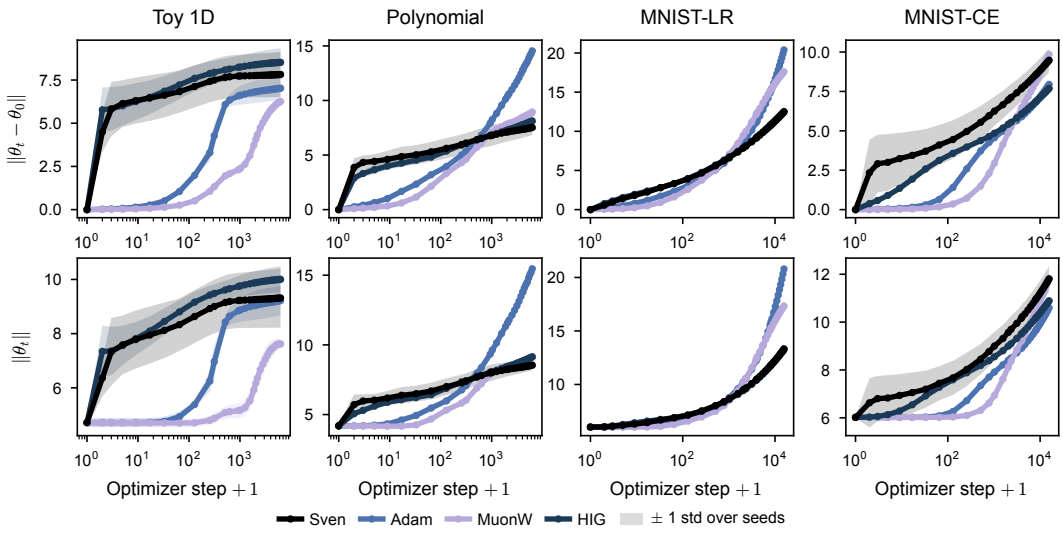

Figure 16: Distance from initialization and parameter norm along the four cached trajectories, seed mean $\pm$ 1 std.

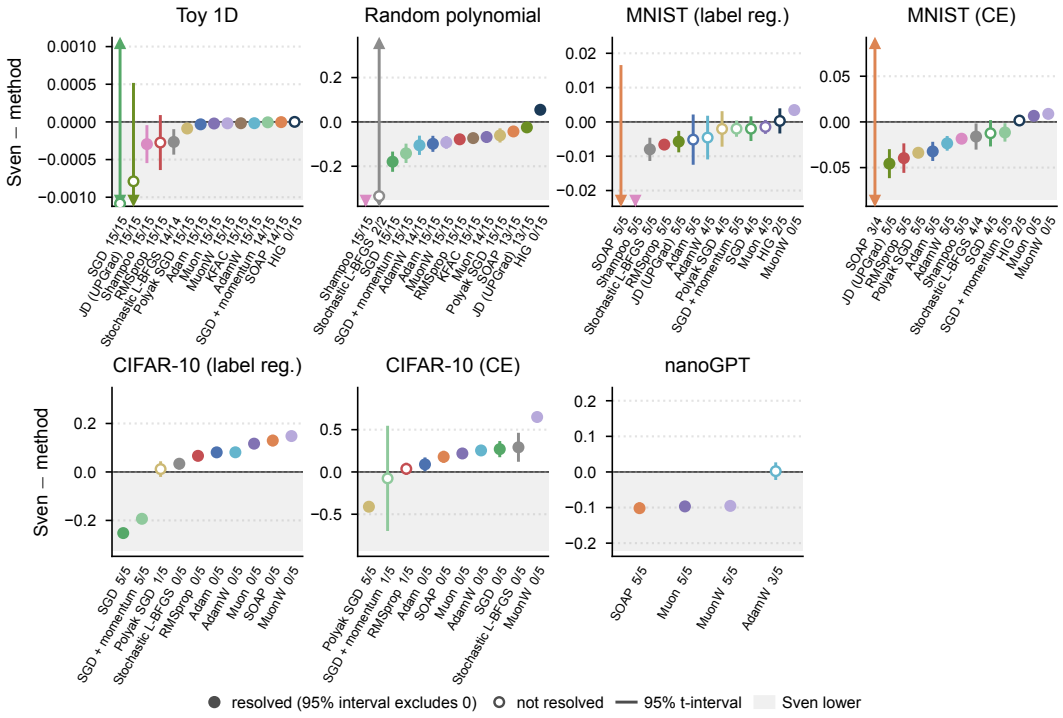

Figure 17: Paired per-seed differences in final validation loss. Differences are taken within a pair (same model seed, and data seed on the two synthetic tasks). Error bars are Student-t 95% intervals on the mean difference and a filled marker is an interval that excludes zero. The count beside each name is the pairs Sven won out of the pairs that finished. An interval wider than the panel is clipped and its end drawn as an arrow.

J Batch-Size Sensitivity

Sven is clearly sensitive to batch size, as it controls the rank of the Jacobian used in computing the step. In Appendix G we observed that the batch-size-limited rank is sometimes significantly smaller

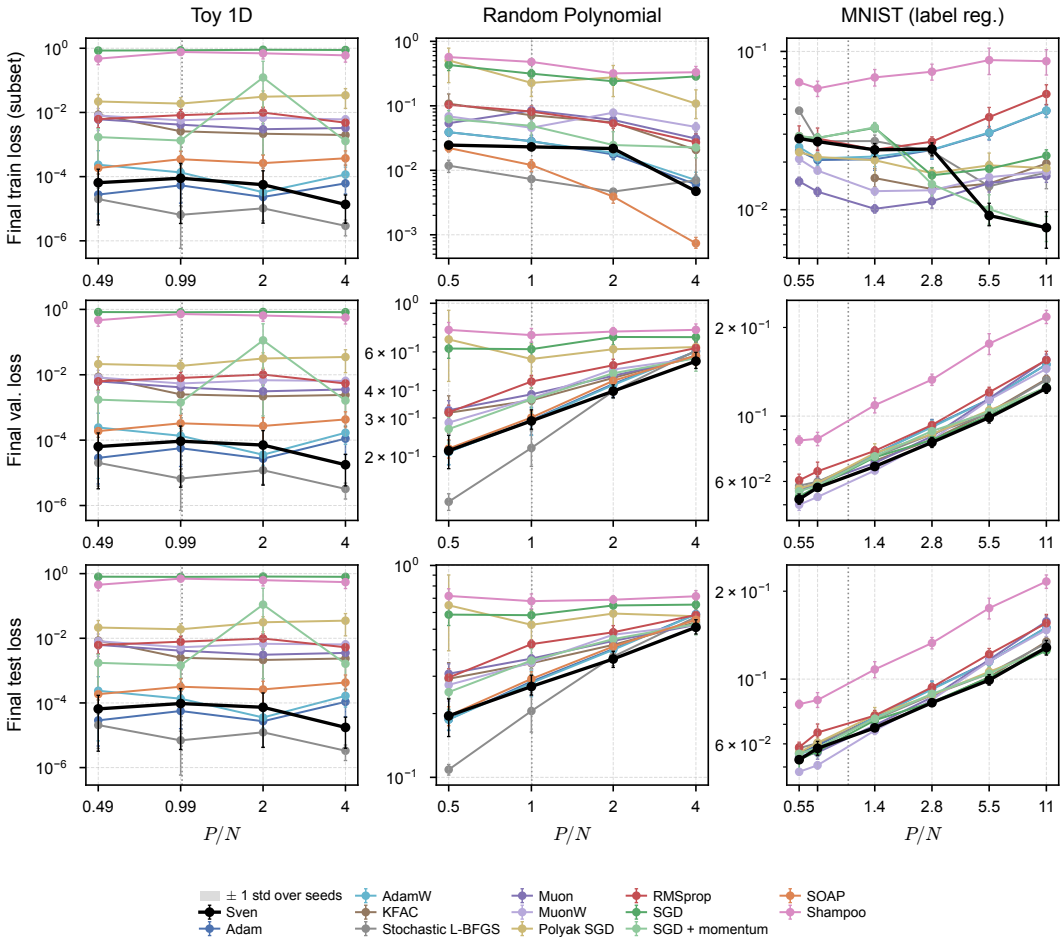

Figure 18: Final loss on the training subset (top), validation (middle) and test (bottom) against P/N for thirteen methods on the 1D, polynomial and MNIST label-regression tasks, with the configuration selected inside each arm on validation loss and a band of ± 1 std over seeds. SOAP is omitted from the MNIST panels, as in Fig. 1.

than the rank of the residuals evaluated on a large test set, so it may be beneficial to run Sven on as large a batch size as possible. In this appendix we study this dependence, sweeping batch size and comparing Sven to other baselines at each setting. We focus on polynomial regression and scan over $B = 8, 16, 32, 64, 128, 256$ at a fixed 20 training epochs, tuning baselines at each B as before. We omit HIG and JD here, focusing mainly on Sven's performance relative to the more standard baselines.

Figure 19 shows the best-performing validation loss versus batch size for Sven and the other baselines. Almost no optimizers have a monotonic trend, Sven included. This suggests that batch size is not particularly influential in overall performance for the polynomial dataset, despite what Fig. 13 suggests about the rank gap between the batch-constrained residuals and the full-dataset residuals.

K Classification with Cross-Entropy

The headline MNIST experiment in Sec. 4 uses a label-regression loss, $L(\mathbf{x}_i; \theta) = \|f_\theta(\mathbf{x}_i) - \mathbf{y}_i\|^2$, which puts it on the same footing as the synthetic datasets and is consistent with the mean-squared-error derivation of Sec. 2. This appendix examines performance using cross-entropy loss (still with $\kappa = 2$). We run optimizer hyperparameter scans using the same models and protocols as the main experiments, described in Appendix D.

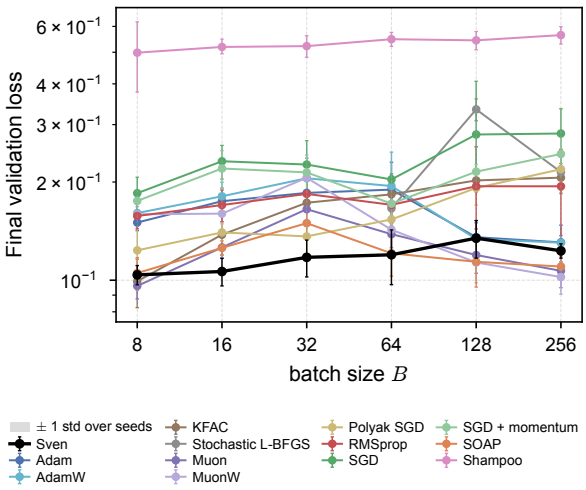

Figure 19: Batch-size sweep on random-polynomial regression: selected final validation loss against B for. Error bars are ± 1 std over seeds.

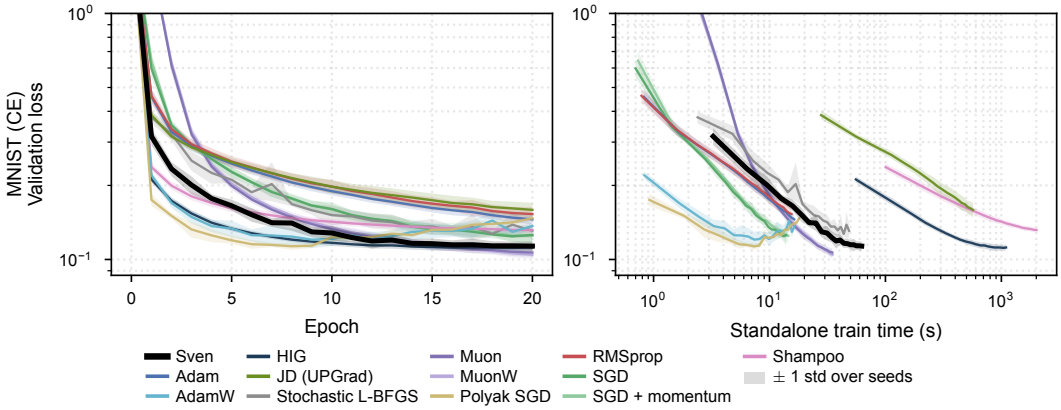

Figure 20: Confirmation-seed trajectories on MNIST with cross-entropy: validation loss against epoch (left) and against synchronized standalone training time (right); bands of ± 1 std over seeds. SOAP is omitted, as in Fig. 1.

Figure 20 plots validation loss versus epoch (left) and wall-time (right) for the best-performing optimizer settings, with wall-time again measured from a dedicated run. Sven performs strongly, outperforming most baselines except HIG and Muon, with HIG taking more than 10 times longer due to its heavier Jacobian computation. Sven thus appears to be competitive with popular baselines in the cross-entropy setting, and is not restricted to regression problems (this was already suggested by the nanoGPT results in Fig. 1). Referring back to Fig. 7, we can analyze MNIST cross-entropy vs label regression by looking at the Jacobian SV spectrum shapes during training (Fig. 3(b) also overlays them at the end of training). The two roughly resemble each other at initialization, but over the course of training the CE spectrum becomes sharply hierarchical while the LR spectrum remains flatter (though it does become steeper). The CE spectrum is also aggressively truncated by the selected (best-performing) `rtol`, while the LR spectrum remains un-truncated throughout training. Since the two scans share everything but the loss, these differences are a property of the objective. A plausible conjecture is softmax saturation: once the model is confident, almost all per-sample gradients become small and nearly parallel, leaving few independent conditions in a batch. We do not study this in detail here, and leave it for future investigation.

Table 23: CIFAR-10 summary table for label regression (top) and cross-entropy (bottom).

Objective	Method	Configuration	Val	Test	Val acc	Test acc	Train (eval)
CIFAR-10 (label reg.)	MuonW	lr=0.01, weight_decay=0.1	0.3357 ± 0.005	0.3437 ± 0.0046	76.8%	76.3%	0.2069
	SOAP	lr=0.01	0.3542 ± 0.017	0.3625 ± 0.0094	78.3%	77.9%	0.03345
	Muon	lr=0.01	0.367 ± 0.013	0.3808 ± 0.015	78.0%	77.3%	0.03828
	AdamW	lr=0.01, weight_decay=0.01	0.4027 ± 0.0075	0.4015 ± 0.011	72.6%	72.8%	0.1456
	Adam	lr=0.01	0.403 ± 0.015	0.4005 ± 0.011	73.1%	73.4%	0.1189
	RMSprop	lr=0.001	0.4173 ± 0.019	0.4165 ± 0.013	74.2%	74.2%	0.07515
	Stochastic L-BFGS	lr=2, max_iter=3, history_size=2	0.4499 ± 0.026	0.4509 ± 0.026	71.4%	71.1%	0.2051
	Polyak SGD	f_star=0, max_lr=1, eps=1e-08	0.472 ± 0.026	0.4687 ± 0.013	69.9%	69.6%	0.04596
	Sven	k=128, lr=0.5, rtol=0.001	0.4838 ± 0.011	0.4938 ± 0.0095	69.9%	69.4%	0.06737
	SGD + momentum	lr=0.001	0.6771 ± 0.013	0.6814 ± 0.0079	54.4%	54.5%	0.1879
	SGD	lr=0.001	0.7361 ± 0.0099	0.7336 ± 0.0091	44.9%	45.6%	0.6141
CIFAR-10 (CE)	MuonW	lr=0.01, weight_decay=0.1	0.7054 ± 0.025	0.7073 ± 0.025	76.9%	76.8%	0.3106
	Stochastic L-BFGS	lr=2, max_iter=2, history_size=5	1.063 ± 0.13	1.09 ± 0.12	73.6%	73.7%	0.1519
	SGD	lr=3	1.085 ± 0.08	1.107 ± 0.074	68.9%	68.6%	0.3555
	AdamW	lr=0.01, weight_decay=0.01	1.101 ± 0.05	1.108 ± 0.026	73.6%	73.5%	0.09618
	Muon	lr=0.01	1.137 ± 0.024	1.18 ± 0.034	78.2%	77.9%	0.03619
	SOAP	lr=0.01	1.176 ± 0.063	1.249 ± 0.062	78.5%	77.6%	0.02897
	Adam	lr=0.001	1.266 ± 0.058	1.304 ± 0.044	75.0%	74.6%	0.05469
	RMSprop	lr=0.001	1.318 ± 0.049	1.366 ± 0.045	74.2%	74.2%	0.06282
	Sven	k=128, lr=0.5, rtol=0.3	1.355 ± 0.016	1.339 ± 0.016	57.9%	58.2%	0.1586
	SGD + momentum	lr=0.3	1.431 ± 0.49	1.454 ± 0.48	56.8%	56.8%	0.6511
	Polyak SGD	f_star=0, max_lr=1, eps=1e-08	1.766 ± 0.041	1.779 ± 0.025	63.1%	63.2%	0.004623

L CIFAR-10 EXPERIMENTS

In this appendix we present results training Sven on CIFAR-10 (Krizhevsky & Hinton, 2009) with a convolutional neural network classifier, where we find that it performs quite poorly. We train ResNet18 (He et al., 2016) under both cross-entropy and label regression losses, and compare against 10 baselines[15]. We train for 20 epochs with $B = 128$ and the splits described in Appendix D.1. BatchNorm was found to be necessary to achieve good performance with *any* optimizer, so we enable it for these studies. This prevents us from using the memory efficient Gram matrix capture for Sven, so we fall back to the costly full Jacobian capture described in Appendix E. Sven is therefore much slower and more memory-hungry on CIFAR than on other baselines.

Table 23 summarizes results for the best-performing optimizer variants on label regression and cross-entropy. Fig. 21 summarizes validation loss/accuracy as a function of epoch, while Fig. 22 plots them against wall time. Sven performs poorly in both settings, especially so for CE, though it has comparatively very low cross-seed spread in this case. Curiously, most methods overfit strongly for CE while they do not for LR. As expected, the wall-time plots show that Sven takes *much* longer to train due to its inefficient Jacobian capture. Fig. 23 plots epoch wall-time and peak GPU memory, confirming this. Looking back to Fig. 11 in Appendix G, we recall that CIFAR-10 CE appears to discard a substantial fraction of the residual norm by the end of training, which may partially explain the degraded performance here[16].

M PARAMETER MASKING AT RESNET SCALE

In this appendix we explore parameter masking, introduced in Appendix E.5, as a theoretical option for reducing memory overhead for large models like ResNet18 where the efficient Gram capture cannot be used. This does not yet work in practice because, as discussed above, autograd backends still instantiate full intermediate gradient tensors even when the requested fraction $f < 1$. We can still study how $f < 1$ affects optimization performance, which we explore here.

We train CIFAR-10 + ResNet18 with label regression at $f \in \{0.05, 0.1, 0.25, 0.5, 1\}$, running three model seeds each. The parameter mask is an elementwise Bernoulli and is resampled at every training step; we choose a *uniformly* random subset of parameters, not sub-units of the network (e.g. full weight matrices). All points use the best-performing Sven configuration from Appendix L. Fig. 24 shows validation loss and test accuracy as a function of the parameter fraction f. Surprisingly, performance is roughly the same at $f = 0.5$, and only modestly degraded in validation loss at $f = 0.25$ (accuracy remains flat). Accuracy drops fast below $f = 0.25$, but the results suggest that performance can be maintained with 75% reduction in memory overhead.

[15] JD and HIG require output Jacobians and were omitted because they cannot scale to ResNet-18; K-FAC and Shampoo were omitted due to cost

[16] The same is not true of CIFAR label regression, so we hesitate to make a strong causal claim

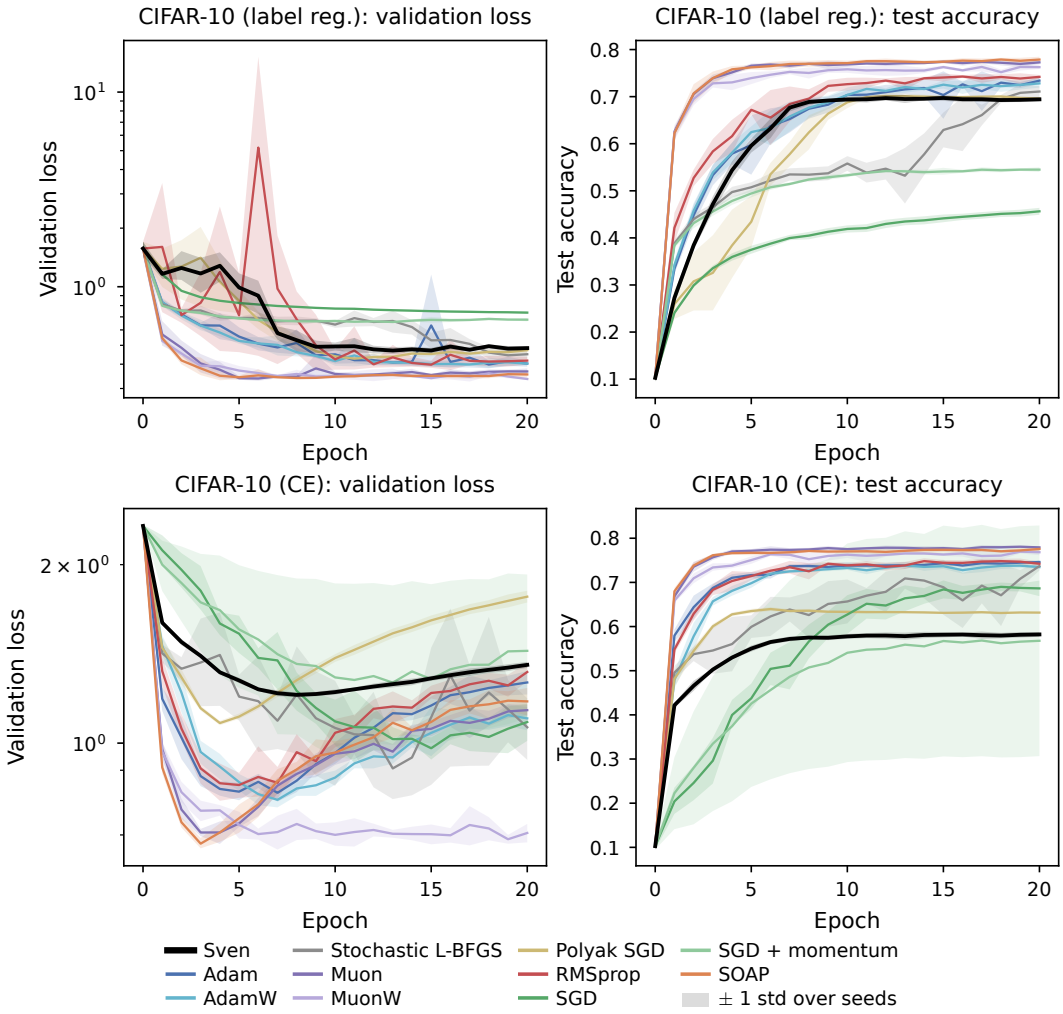

Figure 21: Validation loss (left) and test accuracy (right) for CIFAR-10 / ResNet18 with label regression (top) and cross-entropy (bottom).

Fig. 25 plots peak memory usage and step time as a function of f. Peak memory does fall with f, but by less than 50% and not following the theoretical $1/f$ profile. As stated above, this is because of how the Gram capture works here: ResNet18 must use the full capture, which is one reverse-mode pass over all P parameters that materializes the $B \times P$ Jacobian before the mask is applied. The peak therefore has two parts: the backward pass component (activations + the Jacobian block) which is fixed at every f, and the mask-selected block that is gathered out and which scales with f. The former is the $\sim$12 GB floor the memory curve flattens onto. In the efficient capture with hooks, the floor does not exist and memory is proportional to the active fraction. Interestingly, step time *increases* as f is decreased.

N Language Modeling with Transformers

This appendix studies Sven's performance on language modeling tasks: the nanoGPT tiny-shakespeare runs presented in the main paper, and a trial large-scale run on GPT-2-small. We observe poor performance on GPT-2-small, but with the major caveat that we did not have sufficient time or compute resources to run sensible hyperparameter sweeps or multiple seeds.

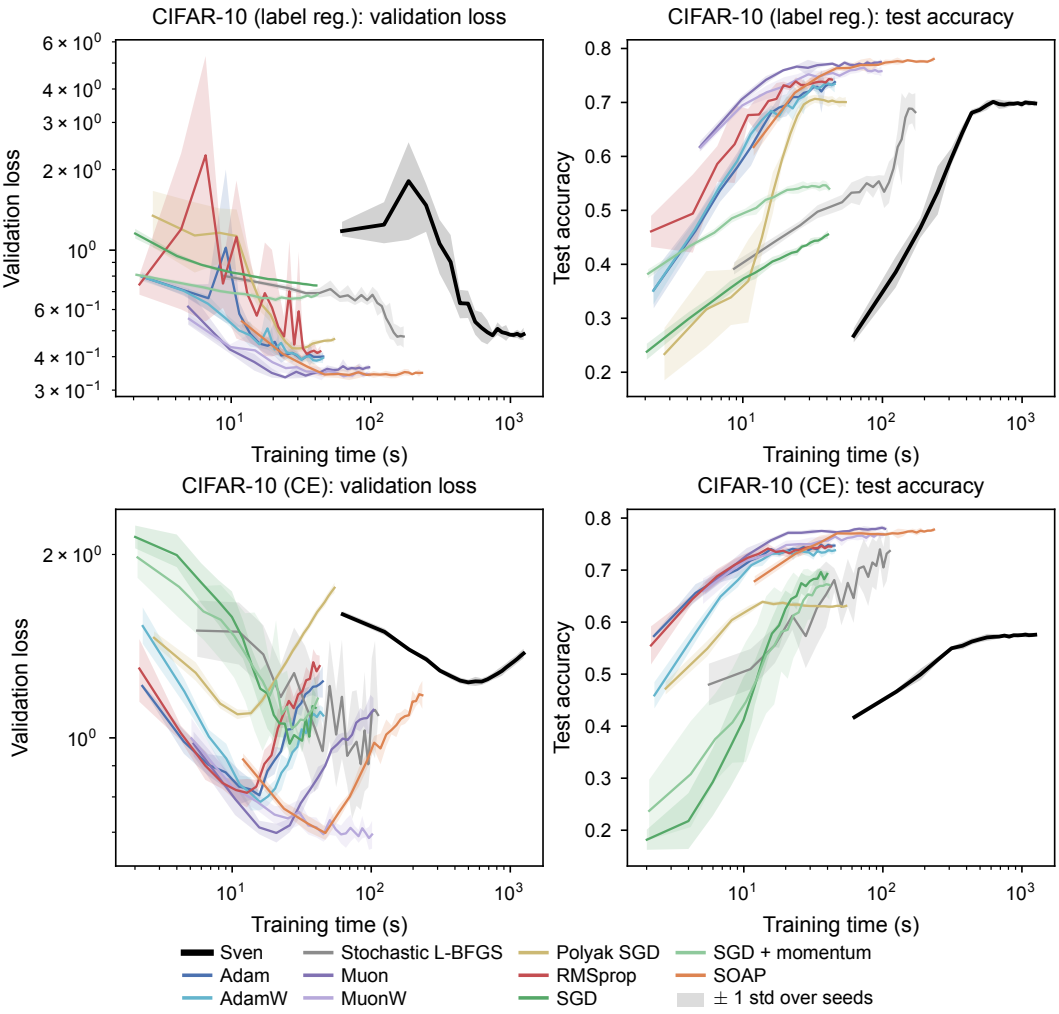

Figure 22: Validation loss (left) and test accuracy (right) plotted against training wall-time for CIFAR-10 / ResNet18 with label regression (top) and cross-entropy (bottom).

N.1 NANOGPT ON CHARACTER-LEVEL SHAKESPEARE

The training setup for the nanoGPT studies is described in Appendix D. We compare five methods: AdamW, Muon, MuonW, SOAP, and Sven[17], scanning hyperparameters for each and choosing the best-performing configuration. Sven is able to use the efficient `hooks` capture in this case. The results are summarized in Fig. 26, which shows Sven as a strong contender: it performs as well as AdamW and better than Muon or SOAP, while being about as time-efficient as Muon/SOAP. One significant caveat is that we only scan Sven's learning rate in this experiment (Fig. 26, right) due to resource constraints. It is possible that Sven performs even better under appropriate tuning, but this was beyond the scope of this study.

N.2 GPT-2-SMALL ON FINEWEB-EDU

In this study we train GPT-2-small (Radford et al., 2019) for one epoch on FineWeb-edu (Penedo et al., 2024) with $B = 16$ (see Appendix D). We are again unable to perform a full hyperparameter scan due to resource constraints, so these results should be read as preliminary and subject to (potentially significant) change under re-tuning. Fig. 27 draws the trajectories and Table 24 gives each method's

[17]We use a limited set of mainstream baselines for this study, as it is more computationally demanding.

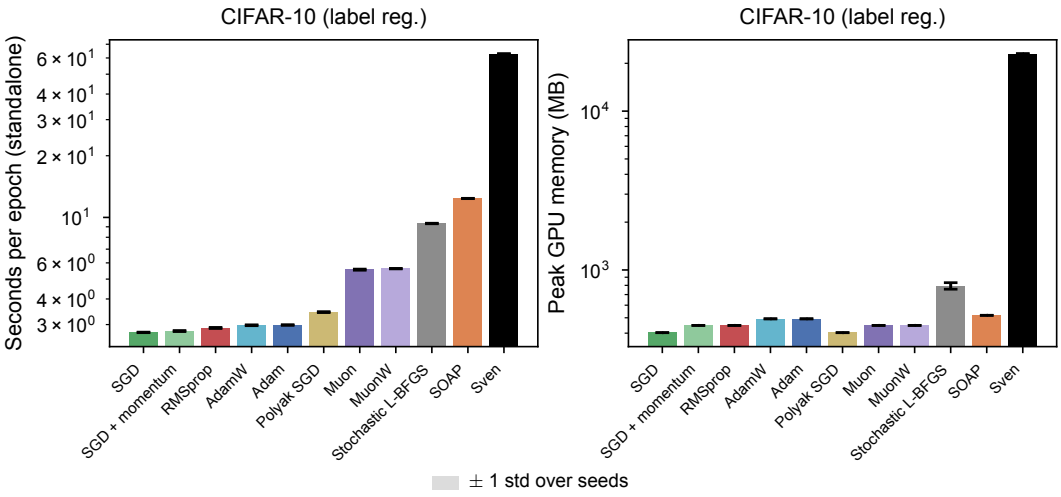

Figure 23: Seconds per epoch and peak GPU memory of each method's selected configuration on the label-regression scan.

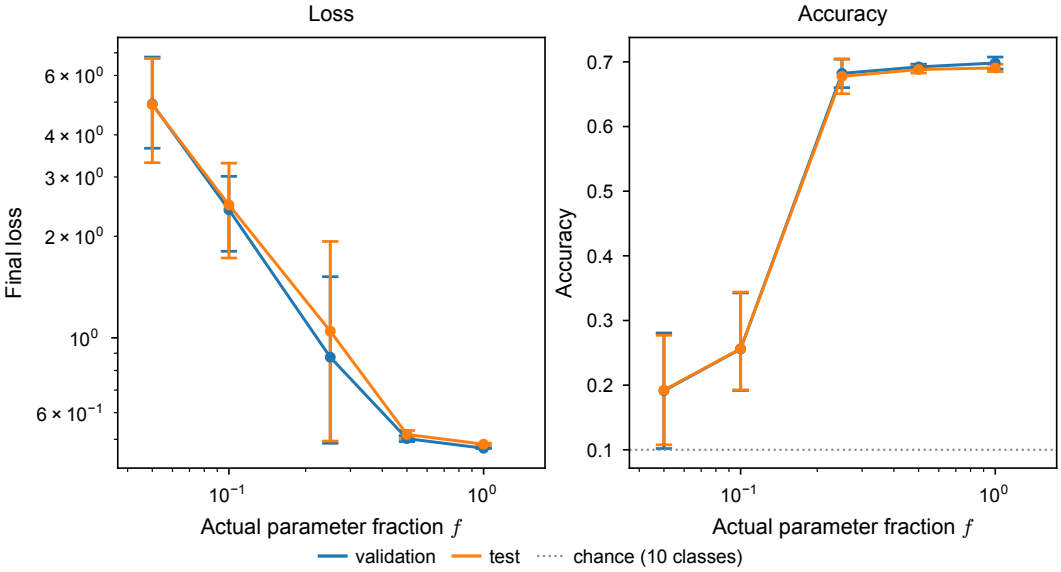

Figure 24: Validation/test loss (left) and accuracy (right) against the actual parameter fraction on ResNet18 / CIFAR-10 with label regression. Quality holds to $f \approx 0.25$ and collapses at $f \leq 0.1$.

best run and its cost. Sven is clearly the worst performer, but it is unknown if this is due to poor hyperparameter settings. Further study is needed to truly assess its performance on large-scale language modeling.

Interestingly, Sven's wall-time cost is *significantly* higher on this task due to the structure of the language modeling task. The hooks capture of Appendix E.2 adds each linear layer's contribution to the Gram as $(gg^\top) \odot (xx^\top)$, which never forms a per-sample gradient — but only when the layer's input has one row per sample. For a sequence model, the input is $B \times L \times d$ and the per-sample gradient is a sum over the L positions. Our current implementation falls back to forming the flattened $B \times (d_{\text{out}} d_{\text{in}})$ per-sample gradient block explicitly and contracting it in chunks. This block is modestly sized for attention heads, but enormous for the output head (`lm_head`): $d_{\text{out}} d_{\text{in}} = 50{,}304 \times 768 \approx 3.9 \times 10^7$. The head is thus captured one sample at a time and the off-diagonal Gram entries recompute their blocks ($\mathcal{O}(B^2)$ block formations). This sets the long step/epoch time

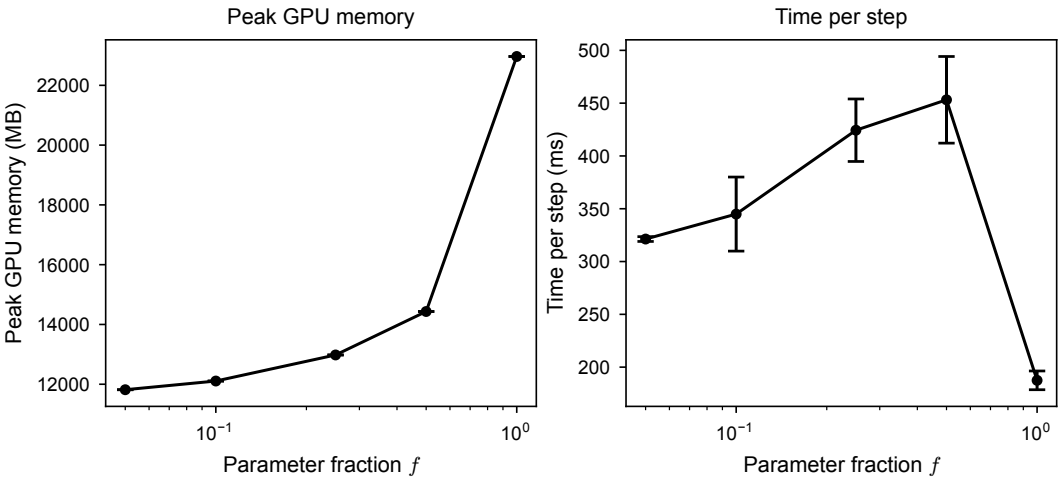

Figure 25: Peak memory (left) and time per optimizer step (right) against the actual parameter fraction; error bars are ± 1 std over seeds.

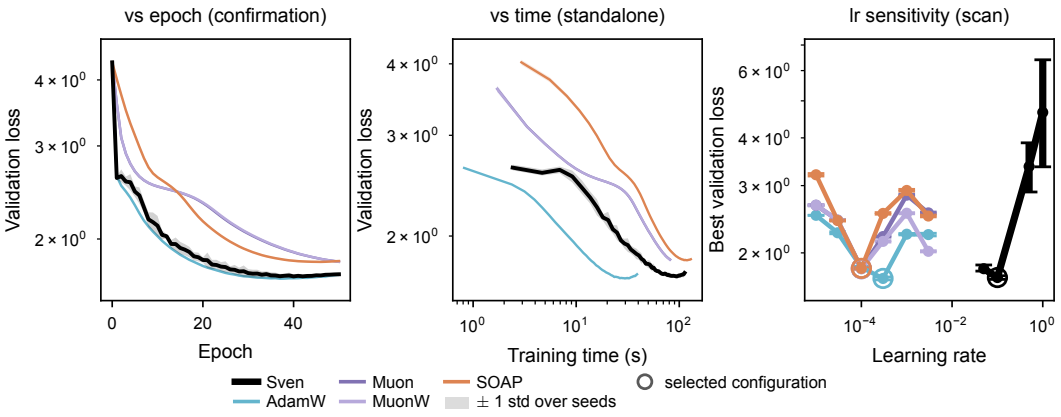

Figure 26: nanoGPT on character-level Shakespeare. Validation loss against epoch (left) and against training time (center), ± 1 std over 5 seeds; final validation loss against learning rate (right).

and larger peak memory of Table 24. The nanoGPT head is much smaller (128×65) and does not cause the same problem. The fix is known but not used: the head's Gram factorizes over positions exactly as the 2D case does, $K_{ab} = \sum_{t,t'} (g_a[t] \cdot g_b[t'])(x_a[t] \cdot x_b[t'])$, and an implementation that contracts over token pairs without materializing the blocks removes the time and the memory cost. This was found to be slower at smaller model sizes and thus disabled, but will likely speed things up at GPT-2 scale.

O VARYING THE κ PARAMETER

In Section 2.2 we introduced the parameter κ, which defines the effective residual $\mathcal{R}^{\alpha}_{\text{eff}} = (\ell^{\alpha})^{\kappa/2}$. In this framing the Sven update step (Equation 7) is directly analogous to the linear least-squares solution when $\kappa = 1$, and $\kappa = 2$ makes the effective residual the per-sample loss itself. We use $\kappa = 2$ as a default throughout the paper because it does not require taking fractional powers of the loss. In this appendix we explore different values of κ and address some subtleties around using $\kappa \neq 2$.

First, we note that using the unsigned effective residual with $\kappa = 1$ is unstable, as it inevitably produces non-finite values within a few epochs. With $\mathcal{R}_{\text{eff}} = \ell^{\kappa/2}$ and a scalar mean-squared-error loss, the row of M belonging to a sample is proportional to $\ell^{\kappa/2-1}\, \partial \ell / \partial \theta$, which diverges as $\ell \to 0$ for $\kappa < 2$. Taking the residual with its sign, $\mathcal{R}_{\text{eff}} = \operatorname{sign}(r)\, |r|^{\kappa}$ for a scalar residual r, removes

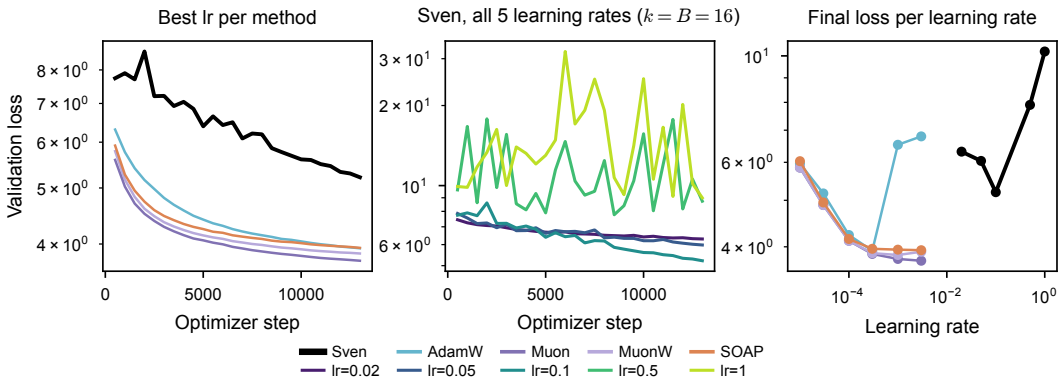

Figure 27: GPT-2-small on FineWeb-edu: validation loss against optimization step for the best learning rate of each method (left) and for each of Sven's learning rates (center), and the final validation loss against the learning rate for every method (right). Resource constraints prevented a full hyperparameter scan.

Table 24: Results summary table for GPT-2-small. Results use just one model seed due to resource constraints.

Method	Best lr	Val	Test	Wall (h)	ms/step	Peak MB
Muon	0.003	3.741	3.632	2.95	793	24650
MuonW	0.001	3.846	3.742	2.95	793	24650
AdamW	0.0003	3.925	3.826	2.89	776	24990
SOAP	0.003	3.935	3.825	4.15	1120	27391
Sven	0.1	5.198	5.107	9.25	2517	36050

the singularity: the row is proportional to $|r|^{\kappa-1}$, which is finite for $\kappa \geq 1$, and the sign cancels between M^+ and $\mathcal{R}$ so the update is unchanged wherever the old form was well defined. With signed residuals, we find that training with $\kappa = 1$ is stable.

Second, we note that κ acts on M as a per-sample row scaling: with $g_\alpha = \partial \ell^\alpha / \partial \theta$ the ordinary per-sample gradient, row α of M is $(\kappa/2)\,(\ell^\alpha)^{\kappa/2-1}\, g_\alpha$, so $M = DG$ for a diagonal D that depends on κ and on each sample's own loss. Wherever G has full row rank — the untruncated overparametrized solve — the pseudoinverse undoes any such scaling exactly, $(DG)^+ = G^+ D^{-1}$, and the factor combines with $\mathcal{R}^\alpha_{\text{eff}} = (\ell^\alpha)^{\kappa/2}$ to give $\delta\theta = -\eta\,(2/\kappa)\,G^+\ell$: the κ update is $2/\kappa$ times the $\kappa = 2$ update, for any per-sample loss, scalar or vector target alike. κ can therefore only act through the truncation, which is applied to the singular values of DG and so sees the reweighting. Comparing κ values at a common η therefore compares effective step sizes. We instead compare at matched *effective step* $2\eta/\kappa$. The shared learning-rate list realizes 3 common effective steps, $\{0.25, 0.5, 1\}$, at all three κ, in two arms: $k = 32$, where the truncation is active, and $k = 64 = B$, where it is not.

Fig. 28 compares various settings of κ on MNIST label regression. In the untruncated arm the scaling argument is confirmed to the precision of the experiment: below the largest effective step size the three κ values agree with a small spread. In the truncated arm they do not agree, showing a significantly larger spread. So κ acts only through the truncation, which is what Sec. 2.2 predicts, since κ changes the row space of M and therefore which directions survive the cut.

P Decreasing Memory Requirements

Appendix E.5 defines two knobs for reducing Sven's memory: micro-batching and parameter batching. In principle, both cut down the size of the Jacobian that must be instantiated, though the efficient

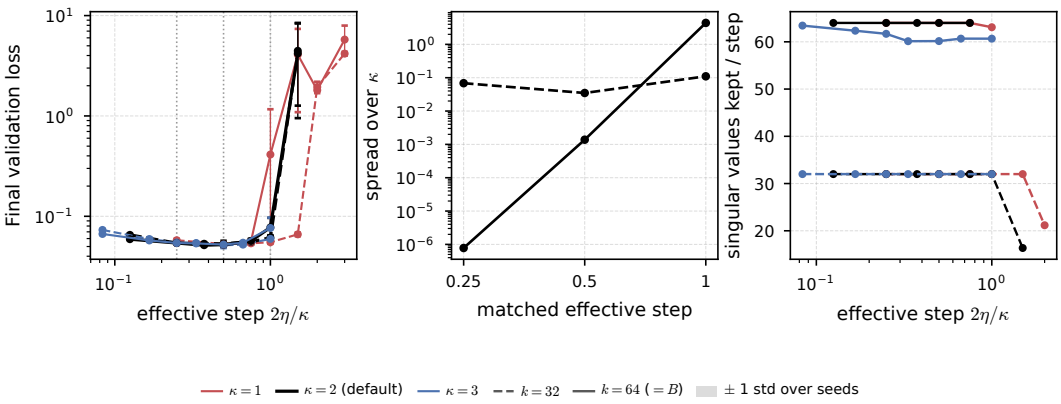

Figure 28: $\kappa \in \{1, 2, 3\}$ on MNIST label regression, plotted against the *effective* step $2\eta/\kappa$ rather than against η, at $k = 32$ (truncation active) and $k = 64 = B$ (inactive). Left: final validation loss. Center: the spread over κ at each matched step. Right: the number of singular values the update inverts against effective step. Bands are ± 1 std over 5 model seeds.

Gram capture makes this cost saving less relevant[18]. This appendix shows micro-batching and parameter batching results on a few of the small-scale problems (toy 1D, polynomial, and MNIST).

We vary the micro-batch size μB between 1 and B, and the parameter fraction $f \in \{0.1, 0.25, 0.5, 0.75, 1\}$, using Sven's selected best configuration from the large hyperparameter scans. Fig. 29 summarizes the results, showing a generally modest impact on performance as μB and f are varied[19]. The notable exception is the 1D regression, which clearly begins to diverge outside of the default settings. This may be related to its strongly hierarchical singular value spectrum observed in earlier appendices. Memory-wise and timing-wise, micro-batching has little impact, as expected. Parameter batching also has little impact on memory but a modest impact on step time. This is consistent with what we observed in Appendix M, and is likely related to how autograd handles parameter masking.

Q Robustness and Stability

Sven's truncation (`rtol` and k) is designed to avoid inverting tiny singular values, but at tight `rtol` or large learning rates it can still become unstable, as can most of the baselines at large learning rates. Here we count failures on the four headline tuning grids (toy 1D, polynomial, MNIST label regression, MNIST cross-entropy), and show Sven's failures on the P/N sweeps of Appendix I and the batch-size sweep of Appendix J beside them as examples of where the diverged fraction gets high. A run is diverged if the runner recorded a numerical failure, or if its final validation loss is non-finite or exceeds ten times its initial value.

Fig. 30 (left) shows Sven's diverged fraction against `rtol` and Fig. 31 its distribution over the full $(\texttt{rtol}, \eta)$ plane, per grid. On the headline grids the rates are relatively small, with failures concentrating at low `rtol` and large η. This is especially pronounced on toy 1D, where the failure rate hits $\sim$50% at the lowest `rtol`. The failure rates are larger for the P/N sweeps, likely because they probe unusual settings; the pattern in $(\texttt{rtol}, \eta)$ is broadly the same.

Fig. 30 (right) gives the diverged fraction per method on the four tuning grids; the denominators differ, since Sven sweeps more hyperparameter configurations. L-BFGS and K-FAC diverge on a substantial number of runs, with K-FAC failing on *all* MNIST runs[20].

[18] It is still important when the efficient capture cannot be used, e.g. the CIFAR-10 studies with ResNet18 in Appendix L.

[19] In Appendix M we found that CIFAR-10 was also fairly robust to reducing parameter fraction down to $f \sim 0.25$, but not below this.

[20] This is due to failures inside the eigendecomposition of its Kronecker-factored Fisher, which is rank-deficient in this regime (Sec. 3)

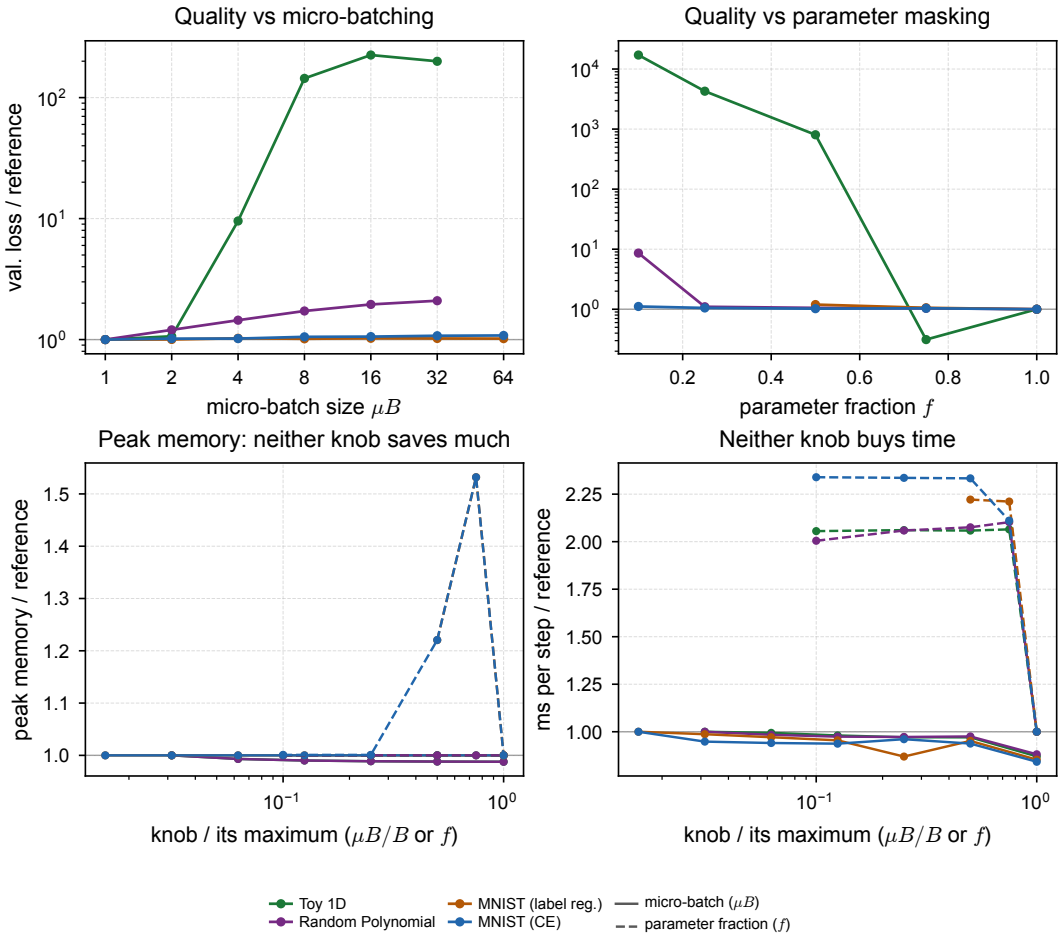

Figure 29: The two memory-mitigation knobs on the four MLP scans, each normalized by its own reference setting. Top: final validation loss against micro-batch size and against parameter fraction. Bottom: peak memory (left) and step time (right) against each knob scaled to its own maximum, relative to the reference setting (solid for micro-batching, dashed for parameter batching).

R Memory and Time Profiling

This appendix measures Sven's memory and wall-time cost relative to the baseline optimizers. The cost of a Sven step splits into the *capture* (retrieving the Jacobian and/or Jacobian Gram matrix) and the *solve* (eigendecomposition of the Gram matrix or full pseudoinverse of the Jacobian, plus the application of the update). We use the memory-efficient implementation (denoted `hooks`) described in Appendix E by default, which is generally the most time- and memory-efficient option. In this appendix we analyze that implementation alongside three others: full Jacobian capture + randomized pseudoinverse (`classic`), full Jacobian capture + Gram eigendecomposition (`full`), and chunked Gram accumulation where the full Jacobian cannot be instantiated in memory (`chunked`).

The `classic` backend is a naive implementation, which costs $\mathcal{O}(kPB)$ due to the explicit $B \times P$ pseudoinverse computation. This can be done with one of three techniques: a full SVD followed by truncation, or two randomized algorithms. The full SVD explicitly decomposes the Jacobian using `torch.svd` and truncates to the top k singular values; it is the most expensive option, though for very small models and reasonable batch sizes we found it fast and well within memory. The two randomized algorithms share the same Halko–Martinsson–Tropp range-finding approach (Halko et al., 2011) but differ in how they extract singular values and right singular vectors from the projected matrix. Both target rank k, oversample by p, and perform q power iterations (in practice, we use $p = 5$ and $q = 1$).

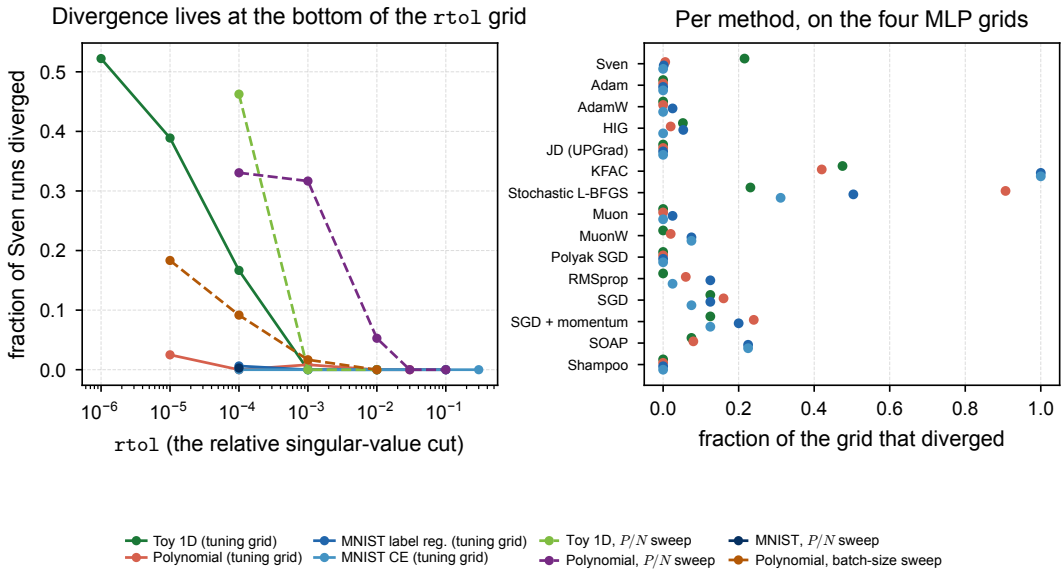

Figure 30: Left: fraction of Sven runs that diverged versus `rtol`, on the four headline tuning grids (solid) and on the P/N and batch-size sweeps (dashed). Right: fraction of each grid that diverged, per method. Sven's grid includes the `rtol` axis, so the denominators are not equal across methods.

Algorithm 3 Randomized truncated SVD (v1) — range-finder + dense SVD.

Require: $A \in \mathbb{R}^{m \times n}$, target rank k, oversampling p, power iterations q

$r \leftarrow \min(k + p,\ \min(m, n))$
Draw $\Omega \in \mathbb{R}^{n \times r}$ with i.i.d. $\mathcal{N}(0, 1)$ entries
$Y \leftarrow A\,\Omega$ ▷ (m, r) random sketch
for $i = 1, \ldots, q$ **do**
    $Y \leftarrow A\,(A^\top Y)$ ▷ Power iteration
end for
$Q, _ \leftarrow \mathrm{QR}(Y)$ ▷ $Q \in \mathbb{R}^{m \times r}$
$B \leftarrow Q^\top A$ ▷ (r, n) projected matrix
$U_B, S, V_h \leftarrow \mathrm{SVD}(B)$
$U \leftarrow Q\,U_B$
return $U_{:,:k},\ S_{:k},\ (V_h)_{:k,:}$

Algorithms 3 and 4 describe the two randomized algorithms[21]. Given a matrix $A \in \mathbb{R}^{m \times n}$, the first variant (Algorithm 3) is simply the standard recipe: build an $m \times r$ orthonormal basis Q of the dominant range of A via QR, form the small projected matrix $B = Q^\top A \in \mathbb{R}^{r \times n}$, and call a dense SVD on B. This is numerically stable — in particular, the singular values of B inherit the conditioning of A rather than its square — but it requires the SVD backend to handle B, which has many more columns than rows. The second variant (Algorithm 4) was introduced as a workaround for cuSolver's `gesvdj` failing with `CUSOLVER_STATUS_INVALID_VALUE` when n is very large (e.g. a large ResNet). After forming the same B, it instead constructs the $r \times r$ Gram matrix $G = BB^\top$, recovers (S, U) from a symmetric eigendecomposition of G via `torch.linalg.eigh`, and obtains the right singular vectors through the rectangular matmul $V_h = S^{-1}U^\top A$. The price for this robustness is the explicit squaring of singular values: the condition number of G is the square of that of B, so the smallest retained singular values lose roughly half their significant digits[22]. For our truncated regime, in which k is chosen so that the retained spectrum is well separated from the discarded tail, this loss is acceptable, and this algorithm is the recommended choice whenever P exceeds a few million.

[21] In these algorithms, the symbol B is the *projected matrix* $Q^\top A$, not the batch size

[22] The memory-optimized Gram formulation has the same limitation

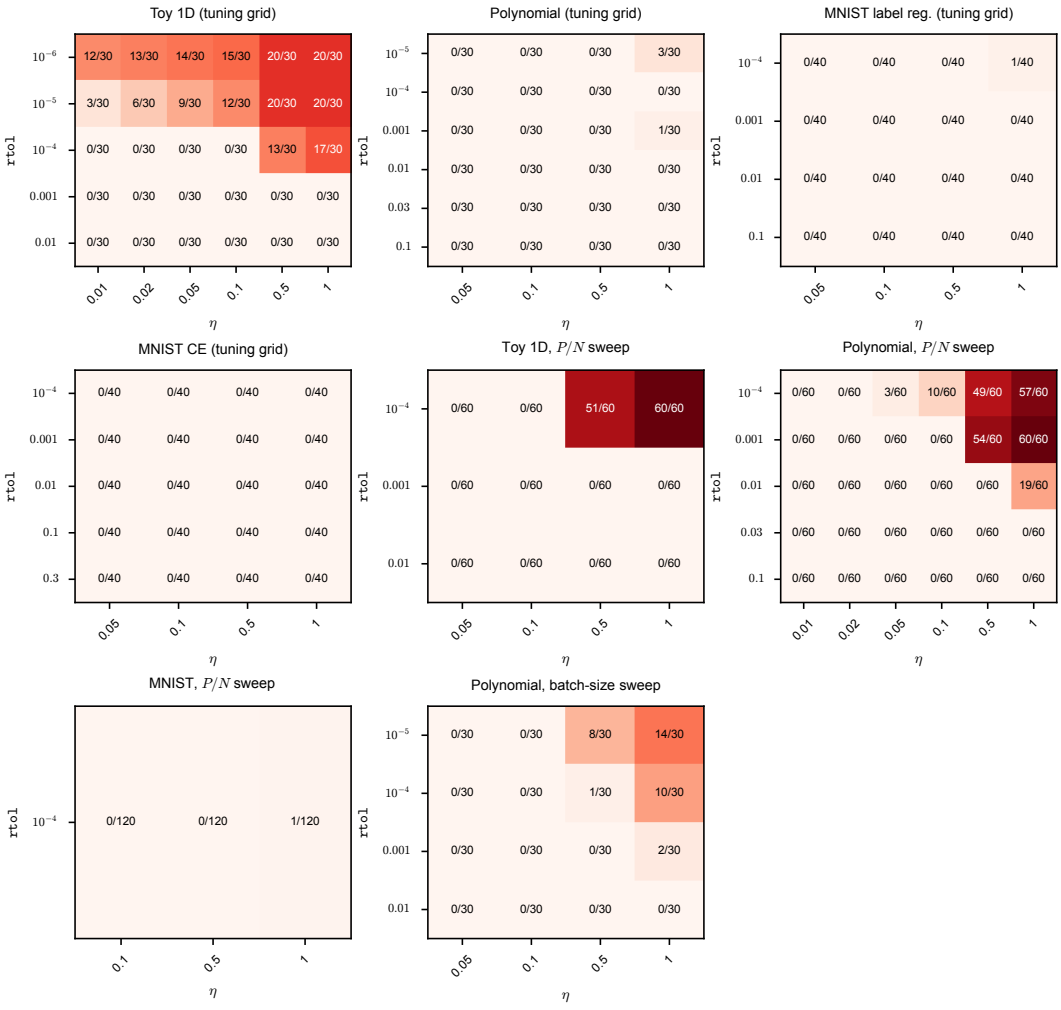

Figure 31: Sven's diverged runs over the $(\texttt{rtol}, \eta)$ plane, as diverged/attempted per cell: the four headline grids, then the three P/N sweeps and the batch-size sweep.

Our profiling runs follow a standard form: initialize the model, the data and the optimizer, move 8 pre-loaded batches to the device, record a resident memory baseline, take 10 warm-up steps and then measure 50 optimizer steps. We report the peak memory during each step and, as the step time, the mean over the last 80% of them. The profiler measures 5 architectures: three MLPs (1D, polynomial, MNIST), nanoGPT and ResNet18. We profile each of Sven's four backends as well as twelve baseline optimizers. For L-BFGS we fix `max_iter` $= 3$ throughout.

Figure 32 summarizes step time and peak memory of each method's selected configuration against parameter count, over five architectures. Sven uses `hooks` capture in all cases except for ResNet18, where it uses `full`. Sven uses nearly the same amount of memory as Adam for the MLPs, increasing somewhat for nanoGPT (for reasons discussed in Appendix N) and far exceeding it for ResNet18 due to the full Jacobian capture. Sven is roughly 2-3 times slower than Adam for the MLPs, becoming slower again for nanoGPT (same reasons) and ResNet18.

Figure 33 gives step time relative to Adam and peak memory relative to SGD for every method at every architecture. The `hooks` capture is the most efficient, with all other Sven backends using more memory and/or time per step across the board. When the `hooks` capture is unavailable for ResNet18, the `chunked` capture saves significant memory while sacrificing a little step time relative to a full capture. The `classic` backend is suboptimal for any reasonably large model, with the MLPs less affected due to their small parameter count.

Algorithm 4 Randomized truncated SVD (v2) — Gram-matrix eigendecomposition.

Require: $A \in \mathbb{R}^{m \times n}$, target rank k, oversampling p, power iterations q

$r \leftarrow \min(k + p,\ \min(m, n))$
Draw $\Omega \in \mathbb{R}^{n \times r}$ with i.i.d. $\mathcal{N}(0, 1)$ entries
$Y \leftarrow A\,\Omega$
for $i = 1, \ldots, q$ **do**
    $Y \leftarrow A\,(A^\top Y)$
end for
$Q, _ \leftarrow \mathrm{QR}(Y)$ ▷ $Q \in \mathbb{R}^{m \times r}$
$B \leftarrow Q^\top A$ ▷ (r, n)
$G \leftarrow B\,B^\top$ ▷ (r, r) Gram matrix
$\Lambda, W \leftarrow \mathrm{eigh}(G)$ ▷ Sort descending
$S \leftarrow \sqrt{\max(\Lambda, 0)}$
$U \leftarrow Q\,W$
$V_h \leftarrow \mathrm{diag}(S^{-1})\,U^\top A$ ▷ Single matmul; safe for any n
return $U_{:,:k},\ S_{:k},\ (V_h)_{:k,:}$

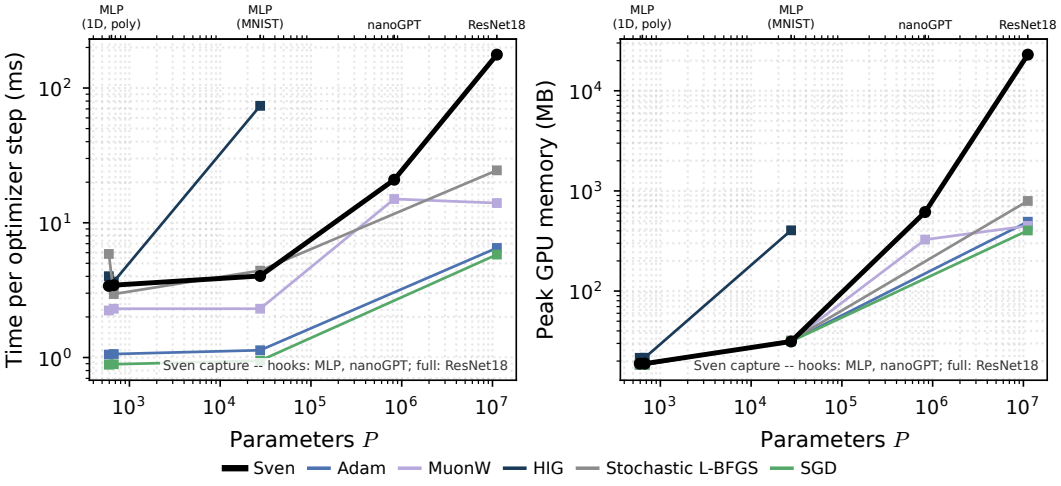

Figure 32: Time per optimizer step (left) and peak GPU memory (right) against parameter count over 5 model scales spanning 593 to 11,181,642 parameters (ResNet18), at each method's selected configuration.

Figure 34 measures step time as a function of k across the different Sven backends. Aside from large time differences between backends, none varies substantially with k. Figure 35 splits each Sven step into the capture and the solve-plus-apply across backends. The capture is cheapest for `hooks`, as expected, and most expensive for the `chunked` capture which serially processes portions of the full architecture. The `full` and `classic` backends both materialize the Jacobian, and their captures cost the same; `full`'s smaller capture *fraction* comes from a larger solve. At this scale a whole step is $\sim$10 ms and the Gram solve is dominated by fixed latencies: a float64 `eigh` through cuSolver costs a few ms however small the matrix, and recovering $J^\top w$ takes one extra forward and backward pass, whereas the classic path already holds J and applies it as one matmul. The Gram solve is therefore near-constant on MLPs, whichever capture fed it. The split only measures compute once the model is large enough for the solve to have real work in it: on ResNet18 the classic solve is a randomized SVD over a $128 \times 11,181,642$ Jacobian at 134 ms against 22.8 ms for the Gram solve, and the two backends' capture fractions agree.

Figure 36 sweeps the batch size, which is directly proportional to the Jacobian/Gram matrix dimensions; the dotted line on the memory plots shows the size of the dense Jacobian. Figure 37 sweeps the parameter count over two model families: an MLP that runs past ResNet18's parameter count, and nanoGPT. The `hooks` step time is constant for the MLP while rising modestly for nanoGPT, whereas

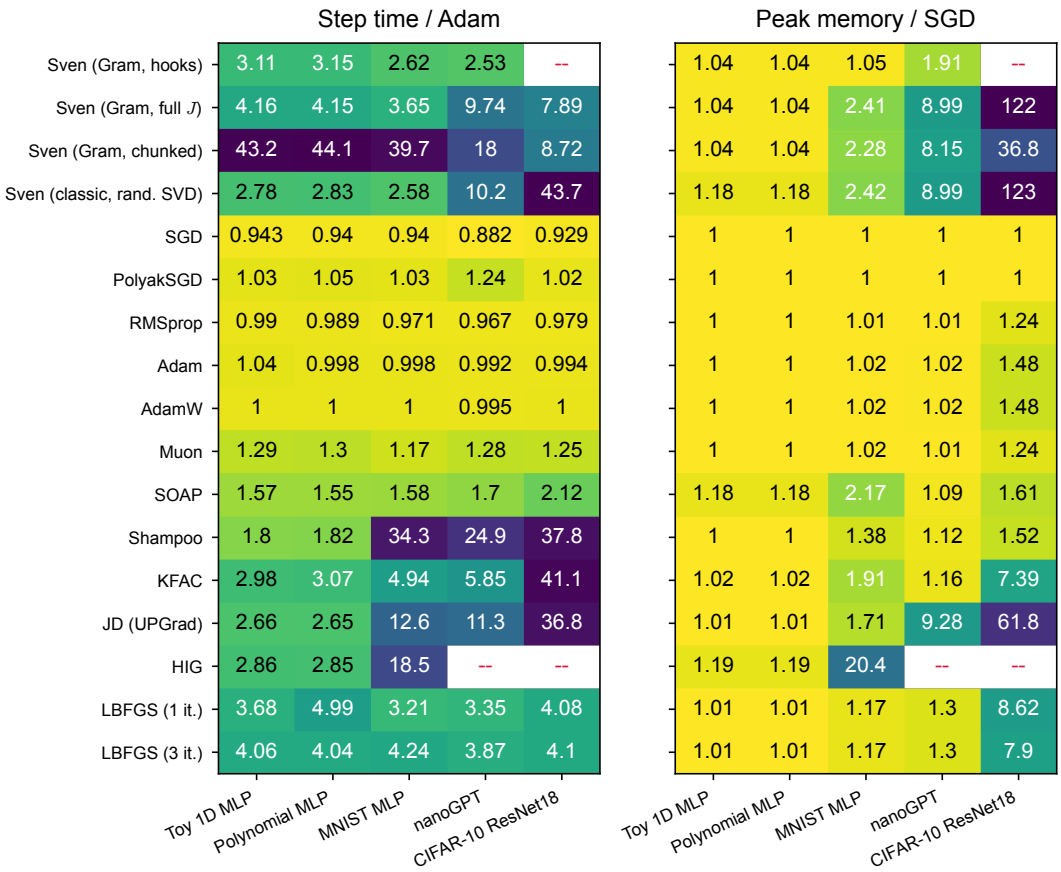

Figure 33: Step time relative to Adam (left) and peak memory relative to SGD (right) for every method at every architecture, at each architecture's profile set point.

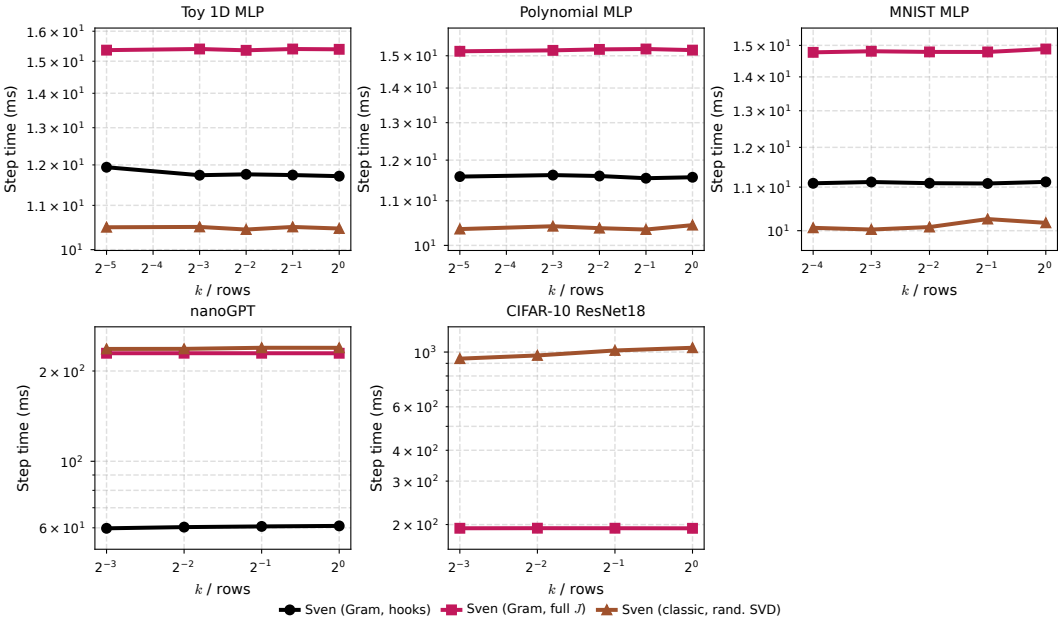

Figure 34: Step time against the rank cap k, per architecture and backend.

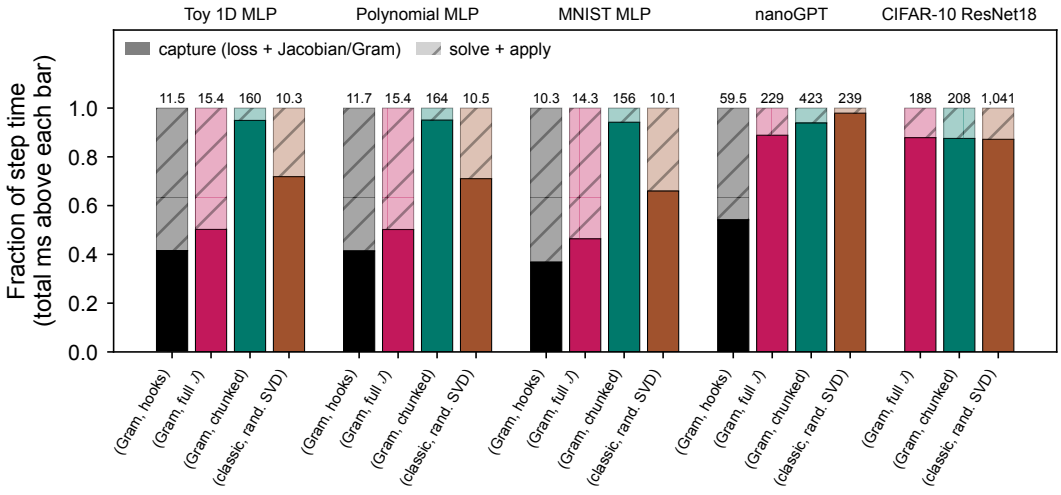

Figure 35: Share of a Sven step spent in the capture (solid) against the solve-plus-apply (hatched), per backend and architecture, with the total step time in ms above each bar.

the `full` and `classic` backends steadily increase. The `hooks` memory usage closely tracks Adam for the MLP, and scales identically to Adam but at a slightly higher baseline for nanoGPT.

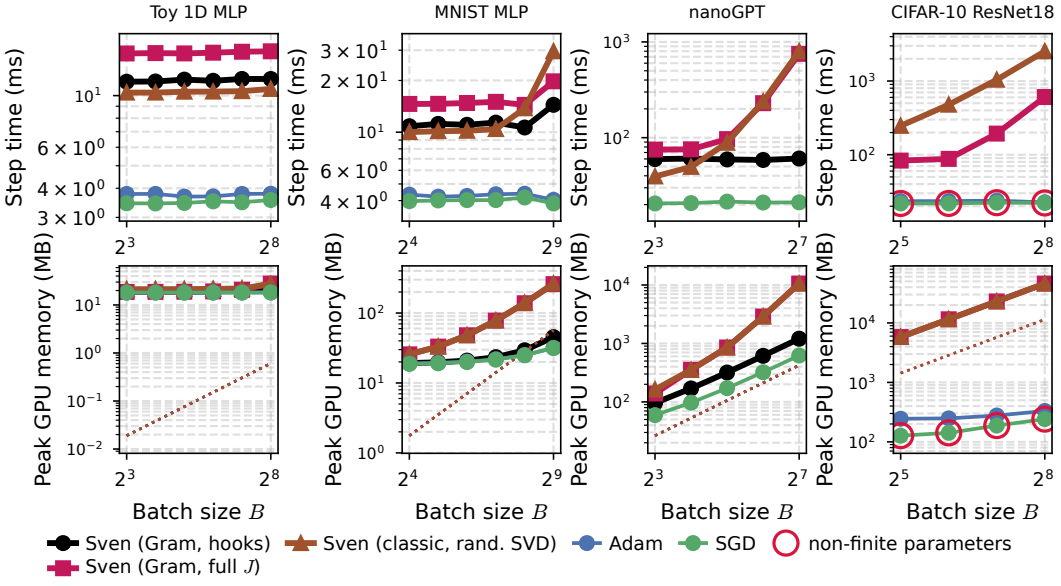

Figure 36: Step time (top) and peak memory (bottom) against the batch size B, per architecture. The dotted line on the memory panels is the memory size of the dense Jacobian.

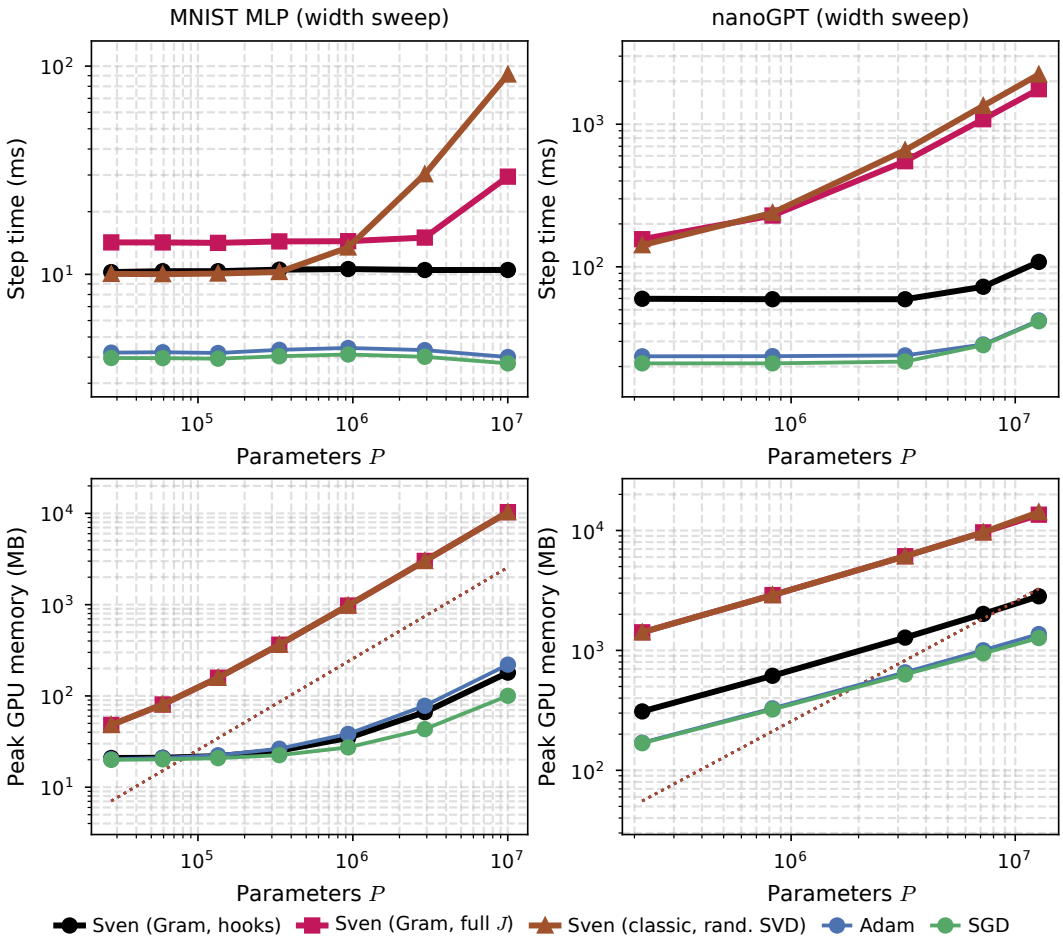

Figure 37: Step time and peak memory against parameter count over two width families: a perceptron sweep that runs past ResNet18's parameter count, and a nanoGPT sweep.