Second-Order, First-Class: A Composable Stack for Curvature-Aware Training

Mikalai Korbit[1] and Mario Zanon[1]

[1]IMT School for Advanced Studies Lucca
Lucca, Italy

Abstract

Second-order methods promise improved stability and faster convergence, yet they remain underused due to implementation overhead, tuning brittleness, and the lack of composable APIs. We introduce Somax, a composable Optax-native stack that treats curvature-aware training as a single JIT-compiled step governed by a static plan. Somax exposes first-class modules – curvature operators, estimators, linear solvers, preconditioners, and damping policies – behind a single step interface and composes with Optax by applying standard gradient transformations (e.g., momentum, weight decay, schedules) to the computed direction. This design makes typically hidden choices explicit and swappable. Somax separates planning from execution: it derives a static plan (including cadences) from module requirements, then runs the step through a specialized execution path that reuses intermediate results across modules. We report system-oriented ablations showing that (i) composition choices materially affect scaling behavior and time-to-accuracy, and (ii) planning reduces per-step overhead relative to unplanned composition with redundant recomputation.

Code availability. Code corresponding to this paper is available at https://github.com/cor3bit/somax.

1 Introduction

Second-order optimizers have long promised improved stability and better time-to-accuracy in machine learning, from Hessian-free optimization [12, 19, 20] to Gauss-Newton methods [8, 32] and structured preconditioners such as K-FAC and Shampoo [9, 18]. Yet modern training pipelines frequently default to Adam [11] or AdamW [16]. A recurring reason is not a lack of algorithms, but rather fragility to implementation choices: curvature model, linear solver, damping rule, and preconditioning can dominate outcomes, with small changes altering convergence or robustness [19, 20, 37]. In practice, these choices are rarely isolated. They are embedded in step implementations that combine curvature construction, linear solves, parameter updates, and damping logic, making it difficult to attribute performance to a specific mechanism or to reproduce a configuration from a paper description.

We treat this as a systems problem. A practical second-order method is a multi-stage step pipeline: (i) construct a curvature snapshot (a linearization state), (ii) solve a regularized linear system to obtain a direction, (iii) apply the direction through an update transformation (e.g., direction momentum, clipping, weight decay), (iv) optionally compute post-update signals (e.g., gain ratio for damping control), and (v) update internal states for the next step. Multiple components in

this pipeline may request overlapping computations (e.g., loss values for trust-region updates and diagnostics, or solver statistics for stopping). Without explicit contracts and execution planning, implementations either recompute expensive quantities or couple modules through shared state and implicit assumptions. Both outcomes hinder experimentation and obscure the true cost profile of a step. Prior work suggests that seemingly small choices inside this pipeline, including damping updates, solver configuration, estimator cadence, and warm starts, can dominate stability and wall-clock time-to-accuracy. A broader related work discussion is provided in Appendix A.

We argue that two missing ingredients are *composability* and *execution planning*. Optimizer libraries such as Optax factor training as a chain of transformations [6], but typical second-order implementations expose functionality through solver-centric or framework-specific interfaces (e.g., KFAC-JAX [3], JAXopt [2]), making it difficult to swap a curvature operator or damping policy without rewriting the step logic. Moreover, second-order steps must coordinate pre-update and post-update computations: for instance, trust-region damping needs an actual decrease that is consistent with the update that was applied, which in turn depends on the exact update transformation and its internal state. This demands an execution model that retains the linearization state long enough to evaluate post-update signals, while avoiding redundant recomputation.

To address these issues, we introduce Somax, a composable stack for curvature-aware training in JAX with Optax-native post-direction step application. Somax assembles second-order methods from swappable modules with explicit contracts: curvature operators (e.g., Exact Hessian and Generalized Gauss-Newton), linear solvers (Conjugate Gradient and direct methods, in parameter or row space), preconditioners (direct diagonal and EMA-based methods), damping policies (constant, trust-region, and step-norm control), and estimator-based telemetry (diagonal, trace, and spectral summaries computed from matrix-vector products without an additional curvature snapshot) [10, 15, 21]. Somax separates static requirements (which statistics are needed, and at what cadence) from a physical execution plan (which computations are enabled, cadence-gated, or omitted). A planner merges module requirements into a static plan that fixes the execution lane, metric schema, and cadences, and a specialized executor performs the step with a single linearization and optional post-update policies without an additional curvature snapshot.

We evaluate Somax with controlled ablations and system-oriented metrics, following guidance on optimizer benchmarking [23, 33, 34]. In particular, we study how lane and estimator choice, solver configuration, damping, and preconditioning affect scaling behavior and time-to-accuracy under a fixed step contract. Lightweight telemetry, such as loss values, gain ratios, top eigenvalues, and diagonal estimates [5, 21, 39], is exposed as an explicit cadence-gated systems cost that can support solver diagnostics and damping control without being hidden inside optimizer-specific code.

Our contributions are as follows.

1. **End-to-end second-order step API.** We provide a step interface that constructs a single step-local curvature state, solves for a direction, applies an Optax transformation to that direction, and updates method and Optax state, with optional post-update signals.
2. **Composable module contracts.** We expose swappable curvature operators, solvers, damping policies, preconditioners, and estimator-backed telemetry, with explicit inputs/outputs and lane-specific constraints enforced at assembly time.
3. **Composition-aware execution planning.** We merge module requirements into a static step plan that fixes the execution lane, supports reuse of step-local curvature state, and cadence-gates optional computations.
4. **Empirical study of module interactions.** We report controlled ablations that isolate how solver, damping, preconditioning, and execution-lane choices behave under a common step interface.

2 Background: Curvature-Aware Training

This section connects the mathematical formulation of second-order updates with their computational realization in modern training systems. We define the learning problem and a first-order baseline, then introduce curvature-scaled directions computed by a damped solve. Finally, we explain why the same subproblem admits three compute regimes (diagonal, parameter-space, and row-space) and how Somax makes this choice explicit.

2.1 Notation

Scalars are plain (e.g., a), vectors are bold lowercase (e.g., $\boldsymbol{a}$), and matrices are bold uppercase (e.g., $\boldsymbol{A}$). The Euclidean norm is $\|\cdot\|$ and $\odot$ denotes element-wise products. The identity matrix is $\boldsymbol{I}$. Indices i and t denote, respectively, data points and iterations. Curvature operators are denoted by $\boldsymbol{H}_t$ and may be exact or approximate; Somax accesses $\boldsymbol{H}_t$ through matrix-free primitives. Key symbols are summarized in Table 1.

Table 1: Notation (excerpt).

Symbol	Meaning
$\mathcal{D}$, n	Dataset and size
$\mathcal{B}_t$, b	Mini-batch at step t and size
$\boldsymbol{w} \in \mathbb{R}^d$	Model parameters (weights)
$\mathcal{L}_{\mathcal{B}_t}(\boldsymbol{w})$	Mini-batch loss at step t
$\boldsymbol{g}_t$	Mini-batch gradient $\nabla \mathcal{L}_{\mathcal{B}_t}(\boldsymbol{w}_t)$
$\boldsymbol{H}_t$	Curvature operator (exact or approximate)
λ_t	Damping for the linear subproblem
$\boldsymbol{s}_t$	Scaled direction (solution of the subproblem)

2.2 Curvature-Aware Training

Given data $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^n$ and a model $f_{\boldsymbol{w}}$ with per-example loss $\ell_i(\boldsymbol{w}) = \ell(f_{\boldsymbol{w}}(x_i), y_i)$, we minimize the empirical risk

$$\mathcal{L}_{\mathcal{D}}(\boldsymbol{w}) \;=\; \frac{1}{n} \sum_{i=1}^{n} \ell_i(\boldsymbol{w}). \tag{2.1}$$

This empirical risk minimization objective is optimized using stochastic mini-batches. At step t, a mini-batch $\mathcal{B}_t$ of size b yields the batch loss and gradient

$$\mathcal{L}_{\mathcal{B}_t}(\boldsymbol{w}) \;=\; \frac{1}{b} \sum_{i \in \mathcal{B}_t} \ell_i(\boldsymbol{w}), \qquad \boldsymbol{g}_t \;=\; \nabla \mathcal{L}_{\mathcal{B}_t}(\boldsymbol{w}_t). \tag{2.2}$$

A first-order update applies a post-processing transform to $\boldsymbol{g}_t$ (e.g., momentum, clipping, or weight decay) and updates weights as

$$\boldsymbol{w}_{t+1} \;=\; \boldsymbol{w}_t \;-\; \alpha_t \, \boldsymbol{u}_t, \tag{2.3}$$

where $\boldsymbol{u}_t$ is the final direction after the chosen first-order transform.

Many optimizers can be viewed as applying a (possibly stateful) linear operator (or preconditioner) to the gradient, $\boldsymbol{u}_t = \boldsymbol{P}_t \boldsymbol{g}_t$. First-order methods choose $\boldsymbol{P}_t$ to be cheap (e.g., diagonal scaling and

Table 2: Selected second-order methods for large-scale stochastic optimization.

Method	**Curv.**	**Lane**	**Solver**	**Estimation / preconditioning**
AdaHessian [40]	Hessian	diag.	explicit	Hutchinson diag. estimator + diag. EMA preconditioner
Sophia-G [15]	GGN (CE)	diag.	explicit	GNB diag. estimator + diag. EMA preconditioner + clipping
Sophia-H [15]	Hessian	diag.	explicit	Hutchinson diag. estimator + diag. EMA preconditioner + clipping
Newton-CG [20]	Hessian	param.	CG/PCG (HVP)	Optional diag. precond.; damping
SGN [8]	GGN (MSE/CE)	param.	CG/PCG (GGN matvec)	Optional diag. precond.; damping
EGN [14]	GGN (MSE/CE)	row	row-Cholesky / row-CG	Row-space solve with internal damping conversion

momentum acceleration), while second-order methods approximate an inverse-curvature operator, $\boldsymbol{P}_t \approx (\boldsymbol{H}_t + \lambda_t \boldsymbol{I})^{-1}$, leading to the damped linear subproblem.

A curvature-aware method modifies the direction computation by incorporating a local curvature model. This replaces a purely gradient-based update with a curvature-scaled direction whose computation is dominated by (i) how curvature is represented and (ii) how the resulting linear system is solved. This viewpoint explains both the algorithmic diversity of second-order methods and the diversity of their implementations. Different curvature structures and solvers induce different dominant primitives and cost profiles. Table 2 previews the selected methods implemented in Somax and used throughout the paper.

A curvature-aware step chooses a (possibly approximate) curvature operator $\boldsymbol{H}_t$ and defines a scaled direction $\boldsymbol{s}_t$ via the damped solve

$$(\boldsymbol{H}_t + \lambda_t \boldsymbol{I})\, \boldsymbol{s}_t = \boldsymbol{g}_t, \qquad \lambda_t \geq 0. \tag{2.4}$$

Damping λ_t regularizes the solve and acts as a control variable for step selection. In practice, (2.4) is often solved by conjugate gradient (CG), or by preconditioned conjugate gradient (PCG) when an explicit preconditioner is available. When solved inexactly, one typically uses the relative residual criterion

$$\|(\boldsymbol{H}_t + \lambda_t \boldsymbol{I})\boldsymbol{s}_t - \boldsymbol{g}_t\| \;\leq\; \tau\, \|\boldsymbol{g}_t\|\,. \tag{2.5}$$

The applied update may still include the same post-direction transforms used by first-order training; the key difference is how $\boldsymbol{s}_t$ is computed.

Second-order methods in modern ML span a range of curvature structures and computational strategies. Diagonal methods retain only per-parameter curvature information and apply explicit scaling, keeping overhead relatively close to first-order training; examples include AdaHessian [40] and Sophia-style updates [15]. Matrix-free methods instead access curvature through matrix-vector products and solve (2.4) iteratively, as in Hessian-free optimization [12, 19, 20] and stochastic Gauss-Newton methods [8]. For Gauss-Newton and generalized Gauss-Newton operators, the same

subproblem can also be realized in row space, leading to a different scaling regime [13, 14, 32]. Across these families, performance often depends on interacting choices around the solve, damping, estimation cadence, and preconditioning, rather than on any single ingredient in isolation.

2.3 Mapping the Theory to System Design

Somax targets the compute regimes in Table 2 by standardizing two contracts: (a) curvature access through matrix-free primitives, and (b) solve execution through one of three lanes. We describe the operator families used in this paper and then summarize how each lane realizes the same damped solve in Equation (2.4).

Two operator families are used throughout the paper.

Exact Hessian (matrix-free). The mini-batch Hessian $\nabla^2 \mathcal{L}_{\mathcal{B}_t}(\boldsymbol{w}_t)$ is applied by Hessian-vector products (HVPs), computed efficiently via Jacobian-vector product (JVP) and vector-Jacobian product (VJP) compositions [28]. The raw Hessian can be indefinite, damping in (2.4) is therefore part of the definition of the computed direction.

Generalized Gauss-Newton (GGN). For losses of the form $\ell(z, y)$ with $z = f_{\boldsymbol{w}}(x)$, the GGN takes the form

$$\boldsymbol{H}_{\mathrm{GGN}}(\boldsymbol{w}_t) = \frac{1}{b} \sum_{i \in \mathcal{B}_t} \boldsymbol{J}_i^\top \boldsymbol{H}_{z,i} \boldsymbol{J}_i, \tag{2.6}$$

where $\boldsymbol{J}_i = \nabla_{\boldsymbol{w}} f_{\boldsymbol{w}}(x_i)$ and $\boldsymbol{H}_{z,i} = \nabla_z^2 \ell(z, y_i)\big|_{z=f_{\boldsymbol{w}}(x_i)}$. Somax instantiates GGN operators for mean squared error (MSE) and cross-entropy (CE) objectives and exposes both matvec and, when available, row-operator primitives.

The same damped subproblem in (2.4) can be realized through different execution regimes, which differ in the linear-algebra object represented explicitly and in the dominant computational primitive (element-wise scaling, matvecs, or row-space solves). In Somax, we refer to these execution regimes as *lanes*.

Diagonal lane. The diagonal lane constructs a diagonal approximation $\boldsymbol{D}_t \approx \boldsymbol{H}_t + \lambda_t \boldsymbol{I}$ (often estimated and smoothed) and applies an explicit inverse:

$$\boldsymbol{s}_t = \boldsymbol{D}_t^{-1} \boldsymbol{g}_t. \tag{2.7}$$

This lane covers diagonal-curvature methods in which curvature is represented as per-parameter scaling.

Parameter-space solver lane. The parameter-space lane defines a matrix-free matvec $v \mapsto (\boldsymbol{H}_t + \lambda_t \boldsymbol{I})v$ and solves (2.5) with CG (or PCG with a supplied preconditioner). This lane covers Hessian-free / truncated-Newton and stochastic Gauss-Newton variants when the dominant primitive is a curvature-vector product.

Row-space solver lane. When $\boldsymbol{H}_t$ admits a row-operator form, as in GN/GGN methods, the system can solve in row space and backproject:

$$(\boldsymbol{J}\boldsymbol{J}^\top + \mu_t \boldsymbol{I})\, \boldsymbol{v}_t = \boldsymbol{r}, \qquad \boldsymbol{s}_t = \boldsymbol{J}^\top \boldsymbol{v}_t. \tag{2.8}$$

This lane exposes a different scaling regime: the linear system dimension is b rather than d. For mean-reduction objectives, we set $\mu_t = b\,\lambda_t$ so that row-space regularization matches the parameter-space damping convention in (2.4).

The remainder of the paper focuses on how this step is realized as a system. We next describe the Somax planner-executor design and lane-specialized execution.

3 Somax: A Composable Stack

Figure 1 summarizes the Somax architecture. The central design choice is to treat curvature-aware optimization as a *planned step pipeline* rather than as an Optax `GradientTransformation`. For first-order methods, returning a transformed direction is often sufficient. For curvature-aware methods, this abstraction is too weak: post-update control signals such as actual decrease and gain ratio must be computed from the *applied* update, which depends on the Optax transform and its internal state. Somax therefore executes curvature construction, damped solve, update application, and optional post-update control within one lane-specialized planned step.

Somax targets single-accelerator JAX execution under JIT. Its step contract enforces five invariants: one call to `step` performs one optimization step; one curvature snapshot is constructed per step; the execution lane is fixed at assembly time, with no runtime lane branching; `StepInfo` has a fixed structure even when probes are disabled or cadence-gated; and post-update control signals are computed from the applied update. These invariants make lane choice, solver configuration, damping, and estimator cadence configurable under one stable `init/step` API. Multi-device execution introduces additional placement and communication issues and is left to future work.

3.1 Assembly and Static Planning

Somax separates method description from method execution. Users specify curvature, estimator, solver, preconditioning, damping, telemetry, and Optax post-processing declaratively. During assembly, defaults are resolved and incompatible combinations are rejected, so the compiled step contains no runtime compatibility logic. Assembly also enforces lane-specific constraints, for example when row-space execution requires row primitives or disallows incompatible solver features.

The planner then converts the assembled method into a static execution plan. This plan records: (i) the execution lane implied by the operator and solver family, (ii) a fixed `StepInfo` schema, and (iii) cadence gates for optional work such as loss-after evaluation, gain-ratio bundles, and estimator probes. The planner is deliberately narrow: it does not search over methods, but specializes a user-specified configuration into a lane-specific, JAX-compatible step path. A component-level contract table is provided in Appendix B.

This plan links the mathematical subproblem in Section 2 to the system behavior studied in Section 4. By making lane, telemetry, and control paths explicit, it turns their cost and interactions into measurable properties of one compiled step.

3.2 Lane-Specialized Compiled Execution

At runtime, Somax executes exactly one lane-specialized step function selected by the static plan. Each step is organized around a single linearization: the curvature operator is initialized once, the damped subproblem is solved in the selected lane, the resulting direction is passed through an internal Optax transformation to obtain the applied update, and optional post-update probes are evaluated against that update. This preserves the semantics required by damping control while keeping the outer training loop unchanged.

Somax exposes the three execution lanes introduced in Section 2: (i) diagonal scaling with no iterative solve, (ii) parameter-space CG/PCG on matrix-free matvecs, and (iii) row-space solves, when row primitives are available, followed by backprojection. These lanes share the same external step interface, but differ in their dominant linear-algebra object and therefore in their scaling behavior. In particular, the row-space lane is a distinct execution regime, not a minor implementation variant.

This distinction is what enables the lane-switching study in Section 4.2, which compares different physical realizations of the same damped subproblem under one interface.

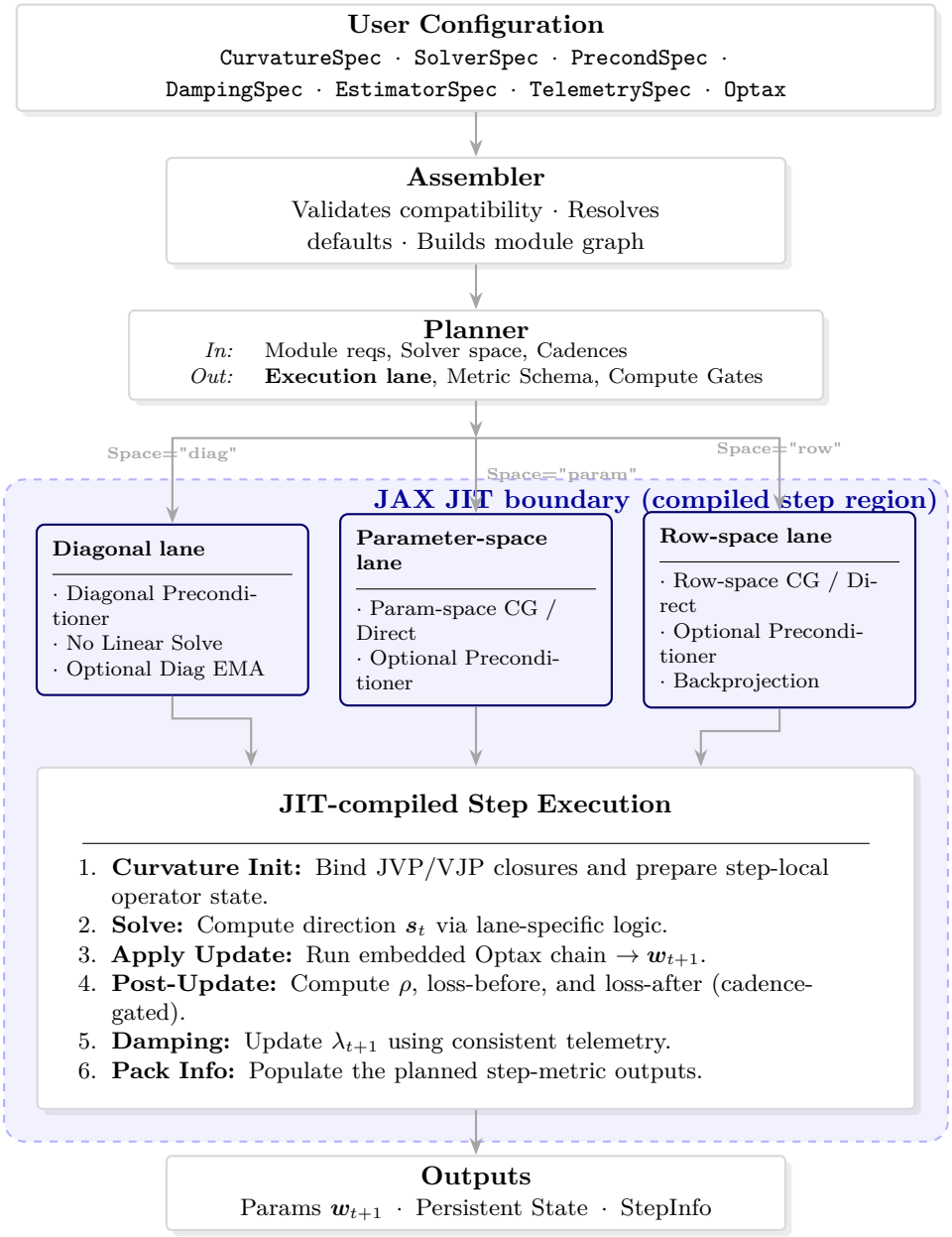

Figure 1: Somax system architecture. The assembler and planner resolve a user configuration into a static execution plan. This plan records one lane-specific execution path (diagonal, parameter-space, or row-space), the step-metric outputs, and cadence gates; the resulting step is then JIT-compiled. The executor manages the full update lifecycle – from linearization to Optax application and optional post-update control signals – ensuring that post-update metrics are consistent with the applied update.

3.3 Planned Telemetry and Cadence-Gated Probes

Somax treats telemetry as part of the step contract rather than as passive logging. Some signals arise naturally during execution, such as solver iteration counts or convergence flags. Others require additional work, such as loss-after evaluation or gain-ratio bundles for damping control. These optional probes are explicitly planned and cadence-gated.

This design serves two purposes. First, it preserves semantic consistency: enabled probes are computed from the same applied update and, when required, the retained linearization state. Second, it preserves JAX stability across ablations: when a probe is disabled or skipped by cadence, Somax still returns the corresponding `StepInfo` field and fills it with a default sentinel. This keeps the output structure static, avoids recompilation, and simplifies downstream logging across method variants.

Telemetry is therefore modeled as optional in-step work with explicit cadence and explicit cost, rather than as free instrumentation added after execution. This makes probe overhead a controlled design variable, which we isolate in Section 3.4.

3.4 Design Validation: Cadence-Gated Probe Overhead

Execution planning is a systems claim, not merely a software abstraction. One concrete consequence is that optional probes should incur cost only when enabled, and that cost should vary with cadence. We therefore report a single-device microbenchmark that isolates cadence-gated post-update probes under a fixed step contract. This benchmark validates one planner-visible mechanism, rather than the full benefit of planning.

The benchmark runs on a single NVIDIA RTX A4000 (16GB) with the JAX GPU backend on a synthetic regression workload with static shapes. The model is a two-hidden-layer ReLU MLP of width 1024, with scalar output, input dimension 512, and batch size 256. We time the JIT-compiled step after warmup and report steady-state median and p90 step time.

We instantiate a Newton-CG configuration in the parameter-space lane with constant damping and fixed solver settings, and vary only the cadence of the post-update *rho* bundle, which computes actual and predicted decrease and therefore requires additional post-update work. All other components are held fixed. The benchmark sweeps `rho_every_k` over `{-1, 10, 5, 2, 1}`, where `-1` disables the probe entirely.

Table 3: Microbench: steady-state overhead of cadence-gated rho probes. Overhead is reported relative to probes disabled.

Setting (`rho_every_k`)	Median (ms)	p90 (ms)	Overhead vs. off
off (`-1`)	0.84	0.87	0.0%
every 10 steps (`10`)	0.95	1.16	12.4%
every 5 steps (`5`)	0.96	1.33	13.8%
every 2 steps (`2`)	1.30	1.40	57.2%
every step (`1`)	1.33	1.34	53.7%

Table 3 shows that post-update probes are a material systems cost rather than "free logging". Sparse cadences increase steady-state median step time by about 12-14% relative to probes disabled, whereas dense cadences increase it by more than 50%. In particular, enabling the rho bundle at every step raises the median from 0.84 ms to 1.33 ms.

This is exactly the contract the planner is meant to expose: disabled probes incur no cost, sparse probes incur bounded cost, and dense probes become a dominant part of the step. Section 4 builds on this execution model to study lane choice, solver-control interactions, and estimator cadence under controlled configuration changes.

4 Experiments

4.1 Experimental Setup

We evaluate Somax as a systems stack for curvature-aware training rather than as a universal optimizer ranking. The experiments comprise three case studies: lane-dependent scaling on synthetic regression, solver-damping interactions on Fashion-MNIST, and estimator-driven diagonal methods on CIFAR-10 with ResNet-20. All experiments use single-accelerator JAX execution under JIT. Additional implementation and protocol details are provided in Appendix C.

4.2 Somax Exposes Structure-Aware Execution Regimes

For structured curvature operators, execution lane is a first-class systems choice. Generalized Gauss-Newton (GGN) is a representative case: the same curvature family admits either parameter-space solves or structured row-space solves. Somax exposes this choice explicitly while preserving the same optimizer-facing interface.

This distinction appears concretely in classical second-order methods. Stochastic Gauss-Newton (SGN) [8] executes the damped linear solve in parameter space, whereas Exact Gauss-Newton (EGN) [14] transfers the computation to a row-space system defined by Jacobian structure. These methods are therefore a natural pair for isolating the systems consequences of lane choice: they share the same curvature family, but their dominant linear-algebra objects scale differently.

We evaluate this effect on a controlled synthetic regression workload designed to vary the two quantities that most directly determine the relative cost of the two lanes: parameter dimension and batch size. We compare two matched second-order configurations that differ only in execution lane, namely a parameter-space CG configuration and a row-space CG configuration. The loss, outer update semantics, learning rate, damping policy, and inner-solver budget are held fixed.

Figure 2 reports steady-state step time under two sweeps. In the left panel, we fix batch size and increase model size, thereby increasing parameter dimension while keeping the row-space dimension fixed. In the right panel, we fix model size and increase batch size, thereby increasing the row-space dimension while keeping parameter dimension fixed. The resulting curves expose distinct scaling signatures. As model size grows, the parameter lane becomes substantially more expensive, while the row lane grows more gradually. As batch size grows, both lanes slow down, but the row lane retains a consistent advantage throughout the tested range.

The key observation is not that one regime universally dominates. Rather, when curvature structure can be exploited, Somax turns that opportunity into an explicit execution choice with its own scaling law. Changing the execution regime changes the dominant linear-algebra object and therefore the cost regime of the second-order step, even though the outer `init/step` interface remains unchanged.

4.3 Interactions Between Modules Shape Performance

Prior work on practical second-order optimization has repeatedly shown that performance is governed not only by the curvature model itself, but also by the interaction between linear solves, damping,

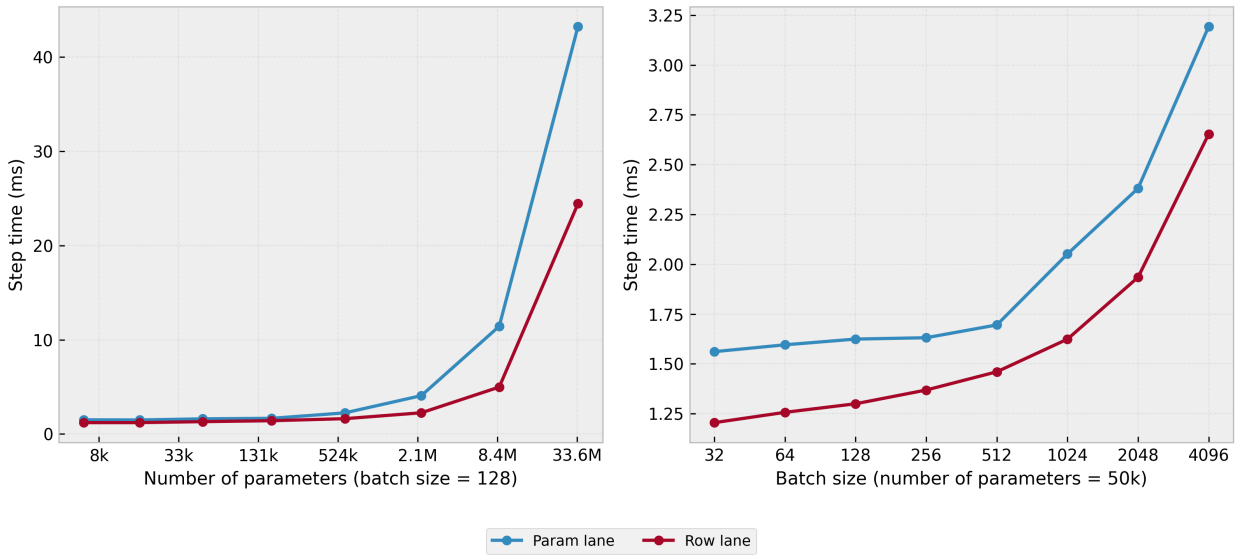

Figure 2: Steady-state step time for matched parameter-space and row-space configurations under a shared training interface. Left: increasing model size at fixed batch size. Right: increasing batch size at fixed model size.

and preconditioning [20, 37]. Somax is designed around this observation: rather than treating a second-order method as a monolithic optimizer, it exposes the main control components as composable modules under a shared step interface.

This case study fixes the parameter-space SGN setting on Fashion-MNIST and varies three modules: the damping policy, the CG budget, and the use of diagonal preconditioning. All runs share the same dataset, model, batch size, and training loop. For each configuration, we perform a learning rate search, selecting the best rate, and then report results averaged over 5 seeds. We include SGD and Adam for context, but the primary purpose of the experiment is to compare second-order module combinations under fixed lane and curvature choices. We use two predefined CG budgets: the *light* budget uses a small iteration cap and loose tolerance, while the *heavy* budget uses a larger cap and tighter tolerance. Exact model and protocol details, including the model architecture and the exact definitions of the light and heavy CG budgets, are given in Appendix C.

Table 4 summarizes the resulting interaction study. Several interaction effects are immediately visible. First, damping strongly changes the usefulness of a given CG budget. With constant damping, the light budget reaches the target faster than the heavy budget while achieving similar final accuracy. With trust-region damping, both configurations are slower, but the heavy budget attains the best final accuracy among the SGN variants. Second, increased solve effort does not uniformly improve performance. Heavier CG budgets consistently increase wall-clock time-to-target, and under constant damping they do so without improving final accuracy. Third, diagonal preconditioning is not uniformly beneficial in this setting. With constant damping it behaves similarly to the non-preconditioned baseline, with nearly identical time-to-target and final accuracy. Under trust-region damping, however, diagonal preconditioning performs poorly: the light and heavy configurations reach only 19.46% and 9.03% final accuracy, respectively. This suggests that module compatibility cannot be assumed, as the chosen diagonal preconditioner interacts unfavorably with trust-region acceptance dynamics.

We show that practical second-order behavior within a fixed lane emerges from module interactions rather than isolated knobs. The goal is not to identify a universally best method, but to

Table 4: Interactions between solver modules shape performance on Fashion-MNIST.

Solver	Precond.	Damping	CG budget	Time to 85% (s)	Final acc (%)
SGD	–	–	–	1.60	90.58
Adam	–	–	–	1.23	90.39
SGN	–	const	light	3.52	90.69
SGN	–	const	heavy	4.79	90.09
SGN	–	trust region	light	7.33	89.69
SGN	–	trust region	heavy	9.13	91.14
SGN	sq grad	const	light	3.54	90.81
SGN	sq grad	const	heavy	5.48	90.27
SGN	sq grad	trust region	light	–	19.46
SGN	sq grad	trust region	heavy	–	9.03

show that changing a small set of modules can materially affect performance. Somax makes these interactions efficient, explicit, swappable, and measurable under a shared execution contract, turning solver design from ad hoc optimizer rewriting to auditable configuration.

4.4 Composability Enables New Optimizer Regimes

Somax treats second-order optimizers as compositions of modular components rather than fixed algorithms. Under this view, Sophia-style methods can be expressed as particular choices of curvature operator, diagonal estimator, refresh cadence, and outer update logic. For example, Sophia-H corresponds to an Exact Hessian curvature operator combined with Hutchinson diagonal estimation and diagonal EMA preconditioning, while Sophia-G replaces the curvature-estimation pair with GGN curvature and the Gauss-Newton-Bartlett estimator [15]. This factorization exposes a broader design space in which related diagonal second-order regimes can be assembled under the same training interface.

To illustrate this design space, we construct a new Sophia-style regime on CIFAR-10 with ResNet-20 by combining GGN curvature with Hutchinson diagonal estimation. We denote this configuration as Sophia-N. The point is not to claim algorithmic novelty in isolation, but to show that Somax makes such recombinations straightforward to express and evaluate within a shared training pipeline. All methods use the same model and surrounding training loop, with SGDM included as a first-order reference.

Figure 3 reports test accuracy against wall-clock time. SGDM achieves the strongest final performance within the common wall-clock budget, reaching 92.21 ± 0.15 test accuracy. Among the Sophia-style methods, Sophia-G and Sophia-N perform similarly, reaching 91.99 ± 0.33 and 91.83 ± 0.13, respectively, while Sophia-H trails at 90.74 ± 0.29. Sophia-N converges competitively with established Sophia variants, showing that modular recombination can produce new configurations without requiring monolithic optimizer implementations. This is the main systems point of the experiment: once the optimizer is factorized into interchangeable modules, new optimizer regimes become concrete, testable assemblies rather than one-off optimizer implementations.

5 Conclusions and Future Work

We introduced Somax, a composable Optax-native stack that makes curvature-aware training a planned, compiled step pipeline. Each step constructs a single step-local curvature state, runs a

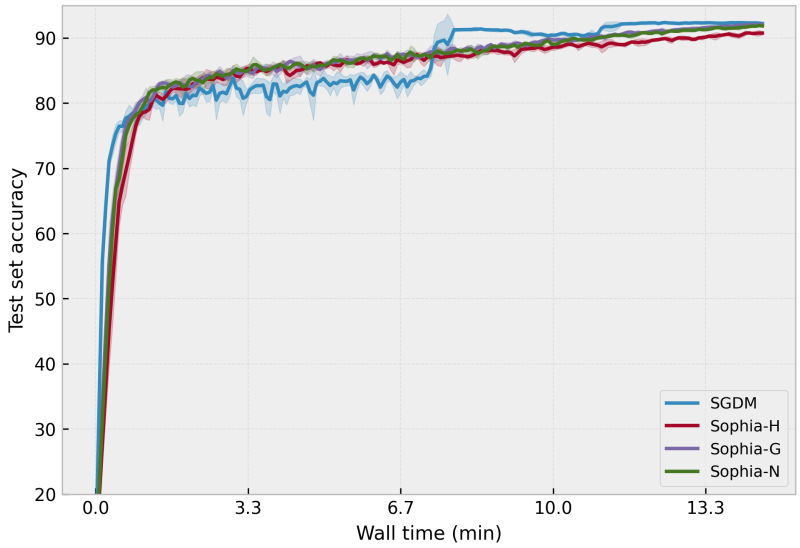

Figure 3: Wall-clock performance comparison of SGDM and Sophia-style optimizer regimes on CIFAR-10 with ResNet-20. Thick lines show mean test accuracy across 5 seeds; shaded bands indicate ±1 standard deviation.

lane-specialized solve (diagonal, parameter-space, or row-space), applies the direction through an embedded Optax update transformation, and updates auxiliary method state. A planner records the execution lane, the fixed output structure for step metrics, and cadences for optional computations, yielding a stable step interface for controlled second-order experiments.

Our experiments support three design claims. First, execution lane is a first-class architectural choice: parameter-space and row-space realizations have different cost models and scaling regimes under the same step contract. Second, solver behavior is coupled to damping and preconditioning; Somax makes this interaction measurable by computing post-update control signals from the applied update. Third, the same contract can enable new optimizer regimes through modular recombination.

Future work. Somax is designed to admit further curvature-aware optimizers as contract-preserving extensions. Natural next steps include structured preconditioners, such as Kronecker-factored [18] or tensor-axis methods [9]. Quasi-Newton methods [1, 4, 38] can be integrated as memory-bounded preconditioners, including diagonal-lane variants. Randomized and sketch-based curvature approximations [17, 29] offer another axis, enabling rank-controlled operator primitives while preserving the same step interface.

The main systems extension is multi-device execution. Distributed curvature-aware training introduces additional concerns, including parameter and optimizer-state sharding, communication-aware lane placement, and collective operations for curvature statistics.

References

[1] Albert S Berahas, Jorge Nocedal, and Martin Takác. A multi-batch L-BFGS method for machine learning. *Advances in Neural Information Processing Systems*, 29, 2016.

[2] Mathieu Blondel, Quentin Berthet, Marco Cuturi, Roy Frostig, Stephan Hoyer, Felipe Llinares-

López, Fabian Pedregosa, and Jean-Philippe Vert. Efficient and modular implicit differentiation. *arXiv preprint arXiv:2105.15183*, 2021.

[3] Aleksandar Botev and James Martens. KFAC-JAX, 2022. URL https://github.com/google-deepmind/kfac-jax.

[4] Richard H Byrd, Samantha L Hansen, Jorge Nocedal, and Yoram Singer. A stochastic quasi-Newton method for large-scale optimization. *SIAM Journal on Optimization*, 26(2):1008–1031, 2016.

[5] Felix Dangel, Frederik Kunstner, and Philipp Hennig. Backpack: Packing more into backprop. *arXiv preprint arXiv:1912.10985*, 2019.

[6] DeepMind, Igor Babuschkin, Kate Baumli, Alison Bell, Surya Bhupatiraju, Jake Bruce, Peter Buchlovsky, David Budden, Trevor Cai, Aidan Clark, Ivo Danihelka, Antoine Dedieu, Claudio Fantacci, Jonathan Godwin, Chris Jones, Ross Hemsley, Tom Hennigan, Matteo Hessel, Shaobo Hou, Steven Kapturowski, Thomas Keck, Iurii Kemaev, Michael King, Markus Kunesch, Lena Martens, Hamza Merzic, Vladimir Mikulik, Tamara Norman, George Papamakarios, John Quan, Roman Ring, Francisco Ruiz, Alvaro Sanchez, Laurent Sartran, Rosalia Schneider, Eren Sezener, Stephen Spencer, Srivatsan Srinivasan, Miloš Stanojević, Wojciech Stokowiec, Luyu Wang, Guangyao Zhou, and Fabio Viola. The DeepMind JAX Ecosystem, 2020. URL http://github.com/google-deepmind.

[7] Sai Surya Duvvuri, Fnu Devvrit, Rohan Anil, Cho-Jui Hsieh, and Inderjit Dhillon. Caspr: Combining axes preconditioners through kronecker approximation for deep learning. In *Forty-first International Conference on Machine Learning*, 2024.

[8] Matilde Gargiani, Andrea Zanelli, Moritz Diehl, and Frank Hutter. On the promise of the stochastic generalized Gauss-Newton method for training DNNs. *arXiv preprint arXiv:2006.02409*, 2020.

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

[10] Michael F Hutchinson. A stochastic estimator of the trace of the influence matrix for laplacian smoothing splines. *Communications in Statistics-Simulation and Computation*, 18(3):1059–1076, 1989.

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

[12] Ryan Kiros. Training neural networks with stochastic Hessian-free optimization. *arXiv preprint arXiv:1301.3641*, 2013.

[13] Mikalai Korbit and Mario Zanon. Incremental gauss-newton descent for machine learning. *arXiv preprint arXiv:2408.05560*, 2024.

[14] Mikalai Korbit, Adeyemi D Adeoye, Alberto Bemporad, and Mario Zanon. Exact gauss-newton optimization for training deep neural networks. *Neurocomputing*, page 131738, 2025.

[15] Hong Liu, Zhiyuan Li, David Hall, Percy Liang, and Tengyu Ma. Sophia: A scalable stochastic second-order optimizer for language model pre-training. *arXiv preprint arXiv:2305.14342*, 2023.

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

[17] Michael W Mahoney et al. Randomized algorithms for matrices and data. *Foundations and Trends® in Machine Learning*, 3(2):123–224, 2011.

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

[19] James Martens and Ilya Sutskever. Learning recurrent neural networks with Hessian-free optimization. In *Proceedings of the 28th international conference on machine learning (ICML-11)*, pages 1033–1040, 2011.

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

[21] Raphael A Meyer, Cameron Musco, Christopher Musco, and David P Woodruff. Hutch++: Optimal stochastic trace estimation. In *Symposium on Simplicity in Algorithms (SOSA)*, pages 142–155. SIAM, 2021.

[22] Raphael A Meyer, Cameron Musco, Christopher Musco, and David P Woodruff. Hutch++: Optimal stochastic trace estimation. In *Symposium on Simplicity in Algorithms (SOSA)*, pages 142–155. SIAM, 2021.

[23] Thomas Moreau, Mathurin Massias, Alexandre Gramfort, Pierre Ablin, Pierre-Antoine Bannier, Benjamin Charlier, Mathieu Dagréou, Tom Dupre la Tour, Ghislain Durif, Cassio F Dantas, et al. Benchopt: Reproducible, efficient and collaborative optimization benchmarks. *Advances in Neural Information Processing Systems*, 35:25404–25421, 2022.

[24] Kazuki Osawa, Yohei Tsuji, Yuichiro Ueno, Akira Naruse, Rio Yokota, and Satoshi Matsuoka. Large-scale distributed second-order optimization using kronecker-factored approximate curvature for deep convolutional neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 12359–12367, 2019.

[25] Kazuki Osawa, Yohei Tsuji, Yuichiro Ueno, Akira Naruse, Chuan-Sheng Foo, and Rio Yokota. Scalable and practical natural gradient for large-scale deep learning. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(1):404–415, 2020.

[26] Vardan Papyan. The full spectrum of deep net Hessians at scale: Dynamics with sample size. *arXiv preprint arXiv:1811.07062*, 2018.

[27] J Gregory Pauloski, Lei Huang, Weijia Xu, Kyle Chard, Ian T Foster, and Zhao Zhang. Deep neural network training with distributed k-fac. *IEEE Transactions on Parallel and Distributed Systems*, 33(12):3616–3627, 2022.

[28] Barak A Pearlmutter. Fast exact multiplication by the hessian. *Neural computation*, 6(1): 147–160, 1994.

[29] Mert Pilanci and Martin J Wainwright. Newton sketch: A near linear-time optimization algorithm with linear-quadratic convergence. *SIAM Journal on Optimization*, 27(1):205–245, 2017.

[30] Jason Rader, Terry Lyons, and Patrick Kidger. Lineax: unified linear solves and linear least-squares in jax and equinox. *arXiv preprint arXiv:2311.17283*, 2023.

[31] Jason Rader, Terry Lyons, and Patrick Kidger. Optimistix: modular optimisation in jax and equinox. *arXiv:2402.09983*, 2024.

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

[33] Robin M Schmidt, Frank Schneider, and Philipp Hennig. Descending through a crowded valley-benchmarking deep learning optimizers. In *International Conference on Machine Learning*, pages 9367–9376. PMLR, 2021.

[34] Frank Schneider, Lukas Balles, and Philipp Hennig. Deepobs: A deep learning optimizer benchmark suite. *arXiv preprint arXiv:1903.05499*, 2019.

[35] Frank Schneider, Felix Dangel, and Philipp Hennig. Cockpit: A practical debugging tool for the training of deep neural networks. *Advances in Neural Information Processing Systems*, 34: 20825–20837, 2021.

[36] Nikhil Vyas, Depen Morwani, Rosie Zhao, Mujin Kwun, Itai Shapira, David Brandfonbrener, Lucas Janson, and Sham Kakade. Soap: Improving and stabilizing shampoo using adam. *arXiv preprint arXiv:2409.11321*, 2024.

[37] Simon Wiesler, Jinyu Li, and Jian Xue. Investigations on Hessian-free optimization for cross-entropy training of deep neural networks. In *Interspeech*, pages 3317–3321, 2013.

[38] Adrian G Wills and Thomas B Schön. Stochastic quasi-Newton with line-search regularisation. *Automatica*, 127:109503, 2021.

[39] Zhewei Yao, Amir Gholami, Kurt Keutzer, and Michael W Mahoney. Pyhessian: Neural networks through the lens of the hessian. In *2020 IEEE international conference on big data (Big data)*, pages 581–590. IEEE, 2020.

[40] Zhewei Yao, Amir Gholami, Sheng Shen, Mustafa Mustafa, Kurt Keutzer, and Michael Mahoney. Adahessian: An adaptive second order optimizer for machine learning. In *proceedings of the AAAI conference on artificial intelligence*, volume 35, pages 10665–10673, 2021.

[41] Lin Zhang, Shaohuai Shi, Wei Wang, and Bo Li. Scalable k-fac training for deep neural networks with distributed preconditioning. *IEEE Transactions on Cloud Computing*, 11(3):2365–2378, 2022.

A Related Work

Second-order methods in modern machine learning are often better understood as step pipelines than as monolithic algorithms. A typical pipeline combines a curvature model (e.g., Hessian, generalized Gauss-Newton, or Fisher), an approximation or estimator (e.g., diagonal probing, Kronecker factors, or axis-wise structure), a linear solver (direct or iterative; parameter-space or row-space), optional preconditioning and warm starts, and a damping policy. Across method families, prior work shows that changing these components can materially affect stability and wall-clock time-to-accuracy. This motivates systems abstractions that make those choices explicit and measurable.

Second-order methods in machine learning. Early work on second-order optimization for deep networks leveraged Hessian-free and truncated-Newton schemes implemented with conjugate gradients [19, 20]. A central lesson from subsequent studies is that practical behavior depends strongly on how the inner linear solve is composed with damping and step control, e.g., ablations of Hessian-free style training document sensitivity to solver-damping interactions and other low-level design choices that are typically omitted from the method description [19, 20, 37]. In particular, studies of Hessian-free style training show that changing only the damping update signal or the inner-loop initialization (e.g., warm-starting CG from the previous solution) can change the amount of Krylov work substantially, with downstream effects on time-to-accuracy [37]. This supports treating the solver-damping loop as a first-class design choice rather than a minor implementation detail. Natural-gradient variants such as K-FAC similarly combine a specific curvature surrogate with structured approximations and non-trivial damping and scheduling choices [18]. Empirically, robust K-FAC behavior depends not only on the factorization itself, but also on how damping and rescaling are applied. This reinforces that the curvature approximation and the control policy must be specified together to define the practical method [18]. Structure-aware preconditioners such as Shampoo exploit tensor axes to reduce cost [9], while diagonal-curvature methods (e.g., AdaHessian, Sophia) estimate Hessian or Gauss-Newton diagonals and combine smoothing and clipping to keep overhead close to first-order training [15, 40]. Recent Gauss-Newton variants revisit stochastic generalized Gauss-Newton operators solved by CG with autodiff-based Hessian-vector products (HVPs) [8] as well as row-space realizations that change the cost model without changing the underlying subproblem [14, 32]. Taken together, these lines of work reinforce the compositional view: solver choice, damping, curvature modeling, and estimation are coupled, and the system should expose them as explicit axes for controlled ablation.

Scaling and systems. Systems work has pushed K-FAC and natural-gradient training to larger scales, emphasizing refresh policies, factor staleness, and communication patterns [24, 25]. Distributed implementations study how factor construction and inversion are distributed across layers, as well as decoupled updates and lower communication volume in all-reduce operations [27, 41]. More broadly, hybridization that mixes higher-order preconditioners with first-order updates in suitable bases (e.g., SOAP: Adam in Shampoo's eigenbasis) highlights practical trade-offs between preconditioning frequency, robustness, and cost [36]. Together, these results support treating second-order training as an execution problem in which scheduling, reuse, and cost control are first-class concerns.

Curvature telemetry and Hessian spectra. Curvature measurements can be made online at relatively low cost. PyHessian, for example, provides scalable estimates of top eigenvalues, traces, and spectral densities, which can inform damping and step-size control [39]. Large-scale studies of deep-network Hessian spectra analyze training dynamics and sample-size effects [26]. BackPACK shows how to expose per-example gradients and curvature quantities with minimal overhead in common frameworks [5], while Cockpit demonstrates the utility of such dense telemetry for real-time debugging and tuning [35]. For trace estimation, Hutch++ reduces probe complexity and makes frequent estimation more practical [22].

Libraries and implementations. Our abstraction builds on the Optax gradient-transformation chain design [6] while targeting step-level semantics required by curvature-aware methods. It complements solver-centric packages such as JAXopt [2] and Lineax [30], which provide optimization and linear-solve primitives but do not manage the optimizer state required for curvature-aware

training. KFAC-JAX offers a modular JAX implementation of K-FAC and related estimators [3]. Optimistix separates search and descent in JAX, providing a useful design precedent for decomposing optimizer logic [31]. Production-oriented implementations of Shampoo and axis-preconditioners in the JAX ecosystem further illustrate memory and communication trade-offs that motivate execution-aware system layers [7].

Benchmarking optimizers. Standardized evaluations caution that optimizer rankings are highly sensitive to tuning and protocol. Suites like DeepOBS, BenchOpt, and broad empirical studies advocate time-to-accuracy metrics and fair search spaces [23, 33, 34]. Our evaluation follows these recommendations by reporting steady-state step time and one-toggle ablations that isolate module and execution-lane choices under a fixed step contract.

B Usage and Reproducibility

This section documents the Somax API surface used in Section 4 of the main paper. Somax exposes a functional `init/step` interface backed by declarative specifications, allowing researchers to swap modules (e.g., replacing `SolverSpec` from CG with Row-Cholesky) without rewriting the training interface. Code corresponding to this paper is available at https://github.com/cor3bit/somax.

B.1 Core Interface

The entry point `somax.assemble(spec, ...)` transforms a configuration into a `SecondOrderMethod` with a JAX-compatible signature. For reproducibility, we also provide named presets (e.g., `somax.make("newton_cg")`) that recover the default hyperparameter settings used in this paper.

```
# 1. Construction
method = somax.make("newton_cg", curvature_kwargs={...})
# OR
method = somax.assemble(curvature=..., solver=..., damping=...)

# 2. Execution (JIT-compatible)
state = method.init(params)

@jax.jit
def train_step(params, batch, state, rng):
    # The step fuses linearization, solve, and updates
    params_next, state_next, info = method.step(params, batch, state, rng)
    return params_next, state_next, info
```

B.2 Declarative Specification System

Somax separates the definition of a method from its execution plan. Table 5 summarizes the primary specification objects used to compose the optimizers in the main paper (Experiments section). These specs are data containers (Pytrees) that the planner analyzes to determine the execution lane and telemetry schema.

Table 5: Primary configuration objects (Specs) in the Somax API.

Spec Object	Role & Planner Effect
`CurvatureSpec`	Defines the operator family (e.g., `hessian`, `ggn_ce`). Declares if row-primitives are available.
`SolverSpec`	Selects the algorithm (e.g., `cg`, `row_cholesky`). **Determines the execution lane** (Param vs. Row).
`DampingSpec`	Defines the trust-region or regularization policy. Requests telemetry (e.g., rho-packs) that the planner must schedule.
`PrecondSpec`	(Optional) Defines diagonal scaling (e.g., `diag_ema`) for the Diagonal lane or as a preconditioner for CG.
`EstimatorSpec`	(Optional) Layers randomized probing (e.g., Hutchinson) onto the step without extra linearizations.

B.3 Example: Assembling a Custom Optimizer

The following example shows how to assemble a more complex method: a Generalized Gauss-Newton (GGN) optimizer with trust-region damping, EMA preconditioning, and a custom Optax chain. The planner automatically resolves the dependency between the trust-region policy (which requires `rho`) and the telemetry machinery (which must schedule `loss_after`).

```
import somax
import optax
from somax.specs import *

# 1. Define the Spec
method = somax.assemble(
    # Use Cross-Entropy GGN
    curvature=CurvatureSpec("ggn_ce", kwargs={"predict_fn": model.apply}),

    # Solve in Parameter Space using PCG
    solver=SolverSpec("cg", kwargs={"maxiter": 20, "tol": 1e-4}),

    # Precondition with EMA diagonal (Adam-style heuristic)
    precond=PrecondSpec("diag_ema", kwargs={"beta": 0.99}),

    # Use Trust-Region Damping
    damping=DampingSpec("trust_region", lam0=1.0),

    # Embed standard Optax transforms for the update application
    tx=optax.chain(
        optax.clip_by_global_norm(1.0),
        optax.scale(-1.0) # Descent direction
    )
)

# 2. The method is now a compiled JAX Pytree ready for .init/.step
```

The assembler resolves defaults and validates compatibility at build time. This is where Somax

enforces lane-specific constraints, so the traced step contains no runtime compatibility checks beyond the lane fixed by the plan. Table 6 summarizes the contracts used throughout the paper.

C Experiment Details

C.1 Case Study I: Switching Lanes

This experiment corresponds to the lane-scaling study in Section 4.2. Its purpose is to isolate how execution-lane choice changes wall-clock cost under a fixed step contract.

Dataset. We use a synthetic scalar regression task generated with sklearn `make_regression`. The dataset contains 200,000 examples with input dimension 128 and additive noise 0.1. Data generation and the train/test split use a fixed seed so that all compared runs see the same underlying problem instance. We use a 90/10 train/test split.

Model. The model is a depth-3 ReLU MLP with scalar output. For the width sweep, we vary the hidden width in $\{32, 64, 128, 256, 512, 1024, 2048, 4096\}$ while keeping batch size fixed at 128. For the batch sweep, we fix hidden width at 128 and vary batch size in $\{32, 64, 128, 256, 512, 1024, 2048, 4096\}$.

Compared methods. We compare the Somax presets SGN [8] `sgn_mse` and EGN [14] with CG solver `egn_mse_cg`, which instantiate the same outer training setup but execute in different lanes: parameter space versus row space. Both use the same learning rate 10^{-3}, constant damping with $\lambda_0 = 1.0$, CG tolerance 10^{-5}, maximum 5 CG iterations per step, and warm-started inner solves. Thus, execution lane is the only intended systems-level difference between the compared configurations.

Measurement protocol. All runs execute on a single NVIDIA RTX A4000 under JAX `jit`. Before timing, we perform two warmup steps to exclude compilation and one-time setup from the reported step-time measurements. Each run then executes for at most 2000 optimization steps. The script records wall-clock step time over windows of 100 training steps. For each run, it stores the mean, median, and standard deviation of these per-window timings. The main figure uses `step_time_median_ms` as the plotted metric. With the current runner configuration, the study uses 5 seeds. For each seed, we first compute a per-run median step time over timing windows. Each plotted point is then the median of these five per-seed summaries.

C.2 Case Study II: Module Interactions

This experiment corresponds to the module-interaction study in Section 4.3. Its purpose is to isolate how solver modules interact within a fixed second-order execution regime.

Dataset. We use Fashion-MNIST, with the standard training and test splits. Training data are loaded into host memory, while the full test set is placed on device for evaluation. No data augmentation is used.

Model. The model is a medium-size convolutional neural network (CNN). It consists of two convolution-ReLU-max-pooling stages followed by three dense layers with ReLU activations and a final 10-class output layer. All compared runs use the same model architecture and batch size 128.

Table 6: **Planner-aware contracts (math → components).** Each component declares requirements that the planner resolves into a static plan (lane, metric schema, cadences), so the executor can fuse the step without redundant recomputation.

Component	**Consumes**	**Produces**	**Planner effect (why it matters)**	**Persistent state**
Curvature operator	parameters, batch	linearization state; matrix-free primitives (and optionally row primitives); loss/grad consistent with snapshot	determines feasibility of the row lane; declares whether post-update probes can reuse the step-local snapshot	none (the linearization snapshot is step-local)
Estimator wrapper (optional)	curvature primitives and RNG (cadence-gated)	additional curvature summaries (e.g., diagonal, trace, spectrum)	adds optional probes without creating a second curvature snapshot; updates the fixed metric schema	optional estimator state (e.g., probe buffers), else none
Preconditioner (optional)	diagonal estimate and damping	diagonal scaling (used by diagonal lane or by PCG)	controls whether the param lane specializes to CG vs PCG; avoids runtime switching inside traced code	EMA buffers (optional)
Linear solver	lane-specific operator primitives, damping, right-hand side	scaled direction; solver diagnostics (if enabled)	fixes the lane (diag/param/row); enables cadence-gated solver telemetry keys	warm start (solution to the previous iteration, optional)
Damping policy	plan-selected telemetry (e.g., gain-ratio bundle) at declared cadence	updated damping state	turns on post-update probes only when required; ensures probes are computed against the applied update	damping state
Optax post-transform	scaled direction and Optax state	applied update and updated Optax state	forces step-level semantics: post-update probes must match the exact applied update, not the raw direction	Optax state

Compared methods. We include SGD and Adam as first-order baselines. For second-order runs, we fix the solver family to parameter-space SGN. Within this fixed lane, we vary three modules: (i) damping policy: constant damping or trust-region damping; (ii) CG budget: a light or heavy budget; and (iii) diagonal preconditioning: on or off. When enabled, the diagonal preconditioner is built from elementwise squared batch gradients. The light budget uses `cg_maxiter` $= 3$, `cg_tol` $= 10^{-3}$, `cg_stabilise_every` $= 10$, and warm start enabled. The heavy budget uses `cg_maxiter` $= 10$, `cg_tol` $= 10^{-5}$, `cg_stabilise_every` $= 5$, and warm start enabled. For trust-region damping, we use `tr_every_k` $= 5$, lower/upper thresholds $(0.25, 0.75)$, multiplicative update factors $(0.5, 1.5)$, and damping bounds $[10^{-12}, 10^{6}]$ with ρ clipping at 5.0. All SGN runs use $\lambda_0 = 1.0$.

Training and measurement protocol. All runs use 10 epochs, batch size 128, and execute on a single GPU under JAX `jit`. Training data are iterated from precomputed epoch permutations so that epoch-boundary reshuffling does not contaminate timing windows. Reported time-to-target denotes wall-clock time to reach 85% test accuracy.

Learning-rate selection and seed aggregation. For each solver configuration, we perform a learning-rate search. After selecting the learning rate for each configuration, the final reported table values are recomputed on the selected runs and aggregated over 5 seeds.

C.3 Case Study III: Composability and New Optimizer Regimes

This experiment corresponds to the composability study in Section 4.4. Its purpose is to test whether Somax modularity can support not only named optimizer families, but also new optimizer regimes instantiated and evaluated under the same training loop.

Dataset. We use CIFAR-10 with the standard training and test splits. Training examples are loaded from TensorFlow Datasets and normalized channel-wise using the standard CIFAR-10 mean and standard deviation. For training, we apply standard data augmentation consisting of zero-padding to 40×40, random cropping back to 32×32, and random horizontal flipping. Test data are normalized but not augmented.

Model. The model is a CIFAR-style ResNet-20. It uses a 3×3 convolutional stem with 16 channels, followed by three residual stages with channel widths 16, 32, and 64. Each stage contains 3 basic residual blocks, yielding the standard ResNet-20 depth for CIFAR. Batch normalization is used throughout, with global average pooling and a final dense classification layer. All compared runs use batch size 128.

Compared methods. We compare SGDM, Sophia-H [15], Sophia-G [15], and a modularly recombined variant denoted Sophia-N. SGDM is included as a first-order baseline. The three Sophia-style methods share the same outer update but differ in how curvature information is constructed. Sophia-H uses Hutchinson-style diagonal estimation based on the Exact Hessian curvature operator. Sophia-G uses a Gauss-Newton-Bartlett-style diagonal estimate based on sampled targets. Sophia-N combines GGN curvature with Hutchinson diagonal estimation. This yields a new Sophia-style configuration obtained through modular recombination in Somax. Sophia-G uses `gamma`=0.05, `n_samples`=1, and `eval_every_k`=10. Sophia-H uses `gamma`=0.01, `n_probes`=1, and `eval_every_k`=10. Sophia-N uses `gamma`=0.05, `n_probes`=1, and `eval_every_k`=10.

Training and measurement protocol. All runs use 200 epochs, batch size 128, and execute on a single GPU under JAX `jit`. SGDM uses momentum 0.9, weight decay 5×10^{-4}, and a step learning-rate schedule. All Sophia-style runs use cosine learning-rate decay with 2000 warm-up steps. Test evaluation is performed every 500 training steps.

Learning-rate selection and seed aggregation. For each solver configuration, we perform a learning-rate search. After selecting the learning rate and learning-rate schedule for each configuration, the final reported table values are recomputed on the selected runs and aggregated over 5 seeds.