CLEAN: SECOND-ORDER LLM TRAINING AT LINEAR MEMORY COST VIA NYSTRÖM SKETCHING

Anonymous authors
Paper under double-blind review

ABSTRACT

Training large language models (LLMs) entails a fundamental trade-off: memory-efficient optimizers such as Adam discard cross-parameter curvature, whereas full-curvature methods such as SOAP can accelerate convergence at prohibitive memory costs. We introduce **Clean**, a memory-efficient and full-curvature optimizer designed to resolve this bottleneck. Clean leverages the randomized Nyström method to accurately approximate the left and right preconditioners in SOAP, and to reduce the optimizer's memory complexity from quadratic to linear in terms of model dimension. We subsequently reintegrate the off-subspace components to capture curvature information beyond the low-rank approximation, preserving rich curvature at minimal memory cost. We further propose **Q-Clean**, a low-precision variant that aggressively compresses optimizer states. Q-Clean reduces optimizer memory consumption by **over 50%** compared to Muon when pre-training a LLaMA-1.3B architecture, all while maintaining strong and competitive predictive performance. Notably, Clean operates with a smaller optimizer-state footprint than standard AdamW while reaching AdamW's final performance **26% faster** in wall-clock time. Furthermore, our methods uniquely enable the pre-training of a 13B-parameter model on a single 80GB GPU, providing a scalable, efficient, and accessible approach to large-scale model optimization.

1 INTRODUCTION

Training large language models (LLMs) forces a trade-off between memory and curvature. Adaptive optimizers store moment statistics that match or exceed the size of the model itself, and second-order methods such as Shampoo (Gupta et al., 2018) and SOAP (Vyas et al., 2024) store even more. The remedies that reduce this footprint typically discard curvature information, producing poorly conditioned updates and slowing convergence (Gupta et al., 2018). Balancing memory efficiency with curvature fidelity has therefore emerged as a central challenge for efficient LLM training.

A range of optimizers have been proposed, each making a different trade-off between memory and curvature. *First–order methods* such as Adam (Kingma & Ba, 2014) and Adafactor (Shazeer & Stern, 2018) are lightweight optimizers that store only diagonal statistics. Their low memory cost makes them practical for large models. *Full–matrix second–order methods* such as Shampoo (Gupta et al., 2018) and KFAC (Martens & Grosse, 2015) explicitly capture curvature and can accelerate training, but their quadratic state and computation make them prohibitive at LLMs scale. More recently, SOAP (Vyas et al., 2024) achieved highly stable and full-rank preconditioning by applying Adam to the gradient projected onto the eigenbasis of Shampoo's preconditioner; however, it still relies on explicitly accumulating full-rank covariance matrices. Several efficient optimizers have been proposed, but all of them have at least one

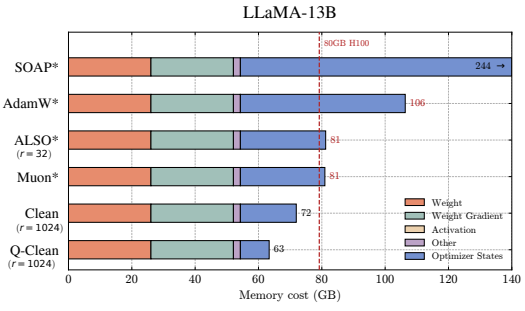


Figure 1: Measured memory consumption for pre-training a LLaMA-13B model on a single 80 GB H100 GPU (maximum sequence length of 1024, batch size of 1). All non-quantized matrices are stored in bfloat16. *Other* is an estimated optimizer workspace, shared across methods. Methods marked * ran out of memory, so their optimizer state is estimated.

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.

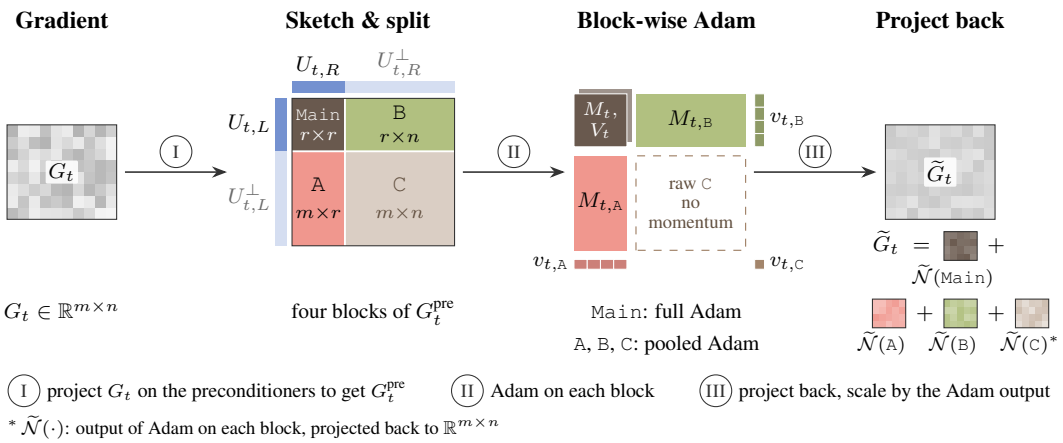


Figure 2: **Clean. (I)** G_t is projected via the estimated left and right preconditioners to yield G_t^{pre}, which is then partitioned into four blocks `Main`, `A`, `B`, and `C`. **(II)** Adam-style update is applied to each block, maintaining either full or pooled moments depending on the block's dimensions (only the shaded buffers are stored). **(III)** The Adam output from each block is projected back into the parameter space and summed to form a full-rank update.

optimizer memory state that grows quadratically with respect to model dimensions (i.e., $\mathcal{O}(mn)$) (Romashov et al., 2026; Jordan et al., 2024; Gong et al., 2026; Liu et al., 2025), making them difficult to scale. To resolve this, we introduce **Clean**, a curvature-aware optimizer that achieves SOAP-level performance while its memory states scale linearly with respect to model dimensions.

Clean replaces explicit gradient covariance accumulations with randomized Nyström approximations, extracting highly accurate, compact estimations for the left and right Shampoo preconditioners. It then splits the update into four orthogonal blocks, `Main`, `A`, `B`, and `C`, and applies full Adam to `Main`. Rather than discarding gradients that fall outside the low-rank core, it applies pooled Adam-like updates to the three remaining blocks. Finally, every block is projected back to the parameter space, and Clean recombines all four to produce a rich and full-rank update, as shown in Fig. 2. We also introduce Q-Clean, a quantized variant that aggressively compresses the stored optimizer states.

To the best of our knowledge, Clean is the first second-order SOAP-like optimizer whose memory grows only linearly with layer dimensions, requiring $\mathcal{O}((m+n)r)$ instead of $\mathcal{O}(mn)$ for a layer of size $m \times n$. Extensive pre-training evaluations on LLaMA-based architectures demonstrate that our approach successfully retains the benefits of full-rank second-order preconditioning at a fraction of the optimizer memory cost. Clean delivers validation perplexities highly competitive with SOAP, yet operates with a smaller memory footprint than AdamW, while showing significant speedup in comparison to Adam's convergence rate. Fig. 1 shows that across all the baselines, Clean and Q-Clean are the only optimizers that allow training a 13B model on an H100 GPU with 80GB of memory.

In summary, Clean is the first optimizer to perform SOAP-style preconditioning in the eigenbasis of randomized Nyström approximations of both Kronecker factors. The Nyström approximations allow the covariance factors to be updated without ever forming them explicitly, reducing their memory cost from quadratic to linear in the layer dimensions for a fixed rank r. Finally, we present Q-Clean, a low-precision extension that quantizes optimizer states to achieve competitive performance while further reducing optimizer memory requirements.

2 Background and Preliminaries

Most deep learning optimizers can be viewed as preconditioned gradient methods. Consider a weight matrix $W_t \in \mathbb{R}^{m\times n}$ with vectorization $w_t = \text{vec}(W_t) \in \mathbb{R}^{mn}$, loss function $\phi(\cdot)$ and stochastic gradient $g_t = \nabla\phi_t(w_t) \in \mathbb{R}^{mn}$. A generic preconditioned update takes the form

$$w_{t+1} = w_t - \eta\, P_t^{-1} g_t, \tag{1}$$

where $\eta > 0$ is the learning rate and $P_t \in \mathbb{R}^{mn\times mn}$ is a positive semi-definite (PSD) preconditioner that encodes curvature information. Different optimizers correspond to different choices of P_t, balancing the accuracy of the curvature with memory and computational costs. We first review several widely used optimizers, and then introduce the technical tools on which our method builds.

2.1 Optimizers as Preconditioners

Eqn. (1) provides a common lens for comparing optimizers: each corresponds to a particular choice of P_t, which dictates the balance between retained curvature information and memory requirements.

Adaptive optimization as preconditioning. Full-matrix AdaGrad (Duchi et al., 2011) sets $P_t = \sum_{s=1}^{t} g_s g_s^\top$, yielding well-conditioned updates but requiring $\mathcal{O}(m^2n^2)$ memory, which is infeasible for modern models with billions of parameters. Common approaches simplify P_t to make optimization practical. *Diagonal preconditioners*, used in optimizers such as Adam (Kingma & Ba, 2014), restrict P_t to a diagonal matrix, capturing individual parameter scales but ignoring cross-parameter correlations.

Balanced preconditioning for matrix layers. A practical surrogate to approximate $P_t \in \mathbb{R}^{mn\times mn}$ is maintaining left and right covariance accumulators:

$$L_t = \sum_{s=1}^{t} G_s G_s^\top, \quad R_t = \sum_{s=1}^{t} G_s^\top G_s, \tag{2}$$

where G_t is the gradient of W at step t. The preconditioned gradient is then

$$\tilde{G}_t = L_t^{-1/4} G_t R_t^{-1/4}, \tag{3}$$

which captures the row and column curvature without ever forming the full preconditioner. Concretely, P_t is approximated by the Kronecker product of two smaller factors, L_t and R_t, where the left preconditioner $L_t \in \mathbb{R}^{m\times m}$ acts on the rows of G_t, while $R_t \in \mathbb{R}^{n\times n}$ acts on its columns. Shampoo (Gupta et al., 2018), a popular second-order optimizer, takes exactly this route and has become the standard Kronecker-factored approximation to full-matrix AdaGrad (Duchi et al., 2011).

Shampoo and practical variants. Shampoo computes and stores per-layer L_t and R_t, utilizing fractional powers $L_t^{-\alpha}, R_t^{-\alpha}$ as left and right preconditioners. The original $\alpha = 1/4$ matches AdaGrad in theory, but Anil et al. (2020) found that $\alpha = 1/2$ often improves stability. However, fully storing L_t and R_t still incurs quadratic memory in the layer width, which is prohibitive for wide layers.

SOAP and eigenspace optimization. Rather than explicitly computing the costly inverse fractional powers of L_t and R_t at every step, SOAP (Vyas et al., 2024) periodically computes their eigen decompositions. It then projects the gradients into this slowly changing principal curvature eigenspace, applies a coordinate-wise Adam update, and projects the step back to the original parameter space. Ultimately, this hybrid approach unifies the structural curvature benefits of second-order optimization with the low per-step cost of diagonal methods, empirically reducing both the number of iterations and the wall-clock time compared to standard Shampoo and AdamW. However, SOAP's memory footprint remains $\mathcal{O}(m^2 + n^2)$, the same as Shampoo's, since it must still store the full $m \times m$ and $n \times n$ Kronecker factors, L_t and R_t at each step. The SOAP algorithm is provided in Appendix C.

2.2 Randomized Nyström preconditioner

The randomized Nyström approximation (Gittens & Mahoney, 2016) provides a framework for efficiently representing symmetric PSD matrices using low-rank approximations. Given a symmetric PSD matrix $A \in \mathbb{R}^{n\times n}$ and a random (Gaussian) matrix $\Omega \in \mathbb{R}^{n\times r}$ where r is a rank parameter, the randomized Nyström approximation is given by:

$$Y = A\Omega, \quad B = \Omega^\top A\Omega, \quad \tilde{A} = YB^\dagger Y^\top \simeq A, \tag{4}$$

where $\tilde{A}$ is a rank-r surrogate, requiring only $\mathcal{O}(nr)$ storage instead of $\mathcal{O}(n^2)$ when $r \ll n$. Strong theoretical guarantees (Gittens & Mahoney, 2016) and practical efficiency of Nyström approximations, make them attractive for optimizer states. However, a naive implementation based on Eqn. (4) is numerically unstable (Frangella et al., 2023). Alg. 2 presents a numerically stable approach for constructing a Nyström approximation (Tropp et al., 2017). Given the eigenvalue decomposition $\tilde{A} = U\Lambda U^\top$ returned by Alg. 2, where $U \in \mathbb{R}^{n\times r}$ has orthonormal columns and $\Lambda = \mathrm{diag}(\lambda_1, \ldots, \lambda_r)$ with $\lambda_1 \geq \cdots \geq \lambda_r \geq 0$, the Nyström preconditioner is obtained by (Frangella et al., 2023):

$$P = (\lambda_r + \mu)^{-1} U(\Lambda + \mu I)U^\top + (I - UU^\top). \tag{5}$$

In effect, P^{-1} scales down each of the top-r eigendirections of A by the factor $(\lambda_r+\mu)/(\lambda_i+\mu) \leq 1$, so that directions with larger eigenvalues are scaled down more, while acting as the identity on $U^\perp$.

3 METHODOLOGY

In this section, we present our second-order memory-efficient optimizer, Clean, starting by approximating the left and right preconditioners in SOAP via the Nyström method. During training, the singular values of the gradients decay rapidly, concentrating most of their energy in a small subspace (Zhao et al., 2024). Since the eigenvalues of $G_t G_t^\top$ and $G_t^\top G_t$ are the squared singular values of G_t, these covariances exhibit an even sharper spectral decay, making the SOAP factors L_t and R_t natural targets for low-rank approximation. Clean exploits this structure to reduce SOAP's $\mathcal{O}(m^2+n^2)$ memory to $\mathcal{O}((m+n)r)$, which is linear in m and n.

Smoothed covariances. Similar to SOAP, Clean preconditions with exponential moving averages of the gradient covariances (Eqn. (6)). However, it never stores these covariances in full, instead, it maintains the low-rank approximations as in Eqn. (7),

$$L_t = \beta_2\, L_{t-1} + (1-\beta_2)\, G_t G_t^\top, \quad R_t = \beta_2\, R_{t-1} + (1-\beta_2)\, G_t^\top G_t. \tag{6}$$

The averaging prevents abrupt changes of basis across refreshes. For $\beta_2 \in [0,1]$, each update is a convex combination of PSD matrices, so L_t and R_t remain PSD and therefore admit randomized Nyström approximations (Frangella et al., 2023).

Low-rank covariance factors. Clean never forms L_t or R_t explicitly. Instead, it maintains rank-r_1 and rank-r_2 Nyström approximations

$$L_t \approx U_{t,L}\Lambda_{t,L}U_{t,L}^\top, \qquad R_t \approx U_{t,R}\Lambda_{t,R}U_{t,R}^\top, \tag{7}$$

where $U_{t,L} \in \mathbb{R}^{m\times r}$ and $U_{t,R} \in \mathbb{R}^{n\times r}$ have orthonormal columns, $\Lambda_{t,L}, \Lambda_{t,R} \in \mathbb{R}^{r\times r}$ are diagonal and non-negative, and $r_1, r_2 \ll \min(m,n)$. Clean only stores thin factors $U_{t,L}, \Lambda_{t,L}$ and $U_{t,R}, \Lambda_{t,R}$. To update them at step t, we replace L_{t-1} in the Eqn. (6) with its approximation and access the result only through a random sketch. Let $\Omega_L \in \mathbb{R}^{m\times(r_1+p)}$ and $\Omega_R \in \mathbb{R}^{n\times(r_2+p)}$ be Gaussian random matrices, where p is an oversampling parameter (Halko et al., 2011). The left sketch is

$$L_t\Omega_L \approx \beta_2\, U_{t-1,L}\Lambda_{t-1,L}\big(U_{t-1,L}^\top\Omega_L\big) + (1-\beta_2)\, G_t\big(G_t^\top\Omega_L\big), \tag{8}$$

and $R_t\Omega_R$ is computed analogously (see Appendix A for details of our Nyström approximations). Both require only thin matrix products with the stored factors and gradient, so no $m\times m$ or $n\times n$ matrix is ever formed. Applying Alg. 3 to these sketches yields the updated factors in Eqn. (7).

Without loss of generality, we assume $r_1 = r_2 = r$. SOAP accumulates L_t and R_t at every step, whereas Clean accumulates them only every T steps, without forming any $m\times m$ or $n\times n$ matrix. Moreover, it only stores the low-rank factors, at a cost of $(m+n+2)r$ parameters, entirely eliminating the quadratic dependence of memory on the layer dimensions. As in SOAP, the factors are refreshed every T iterations and reused in between, so L_{t-1} and R_{t-1} always refer to the most recent refresh.

Block-wise low-rank preconditioning. Following (Gupta et al., 2018; Morwani et al., 2024), we precondition the gradient as $G_t^{\text{pre}} = L_t^{-1/2} G_t R_t^{-1/2}$. To expand G_t^{pre} with the Nyström preconditioner in Eqn. (5), for $S \in \{L, R\}$, we write

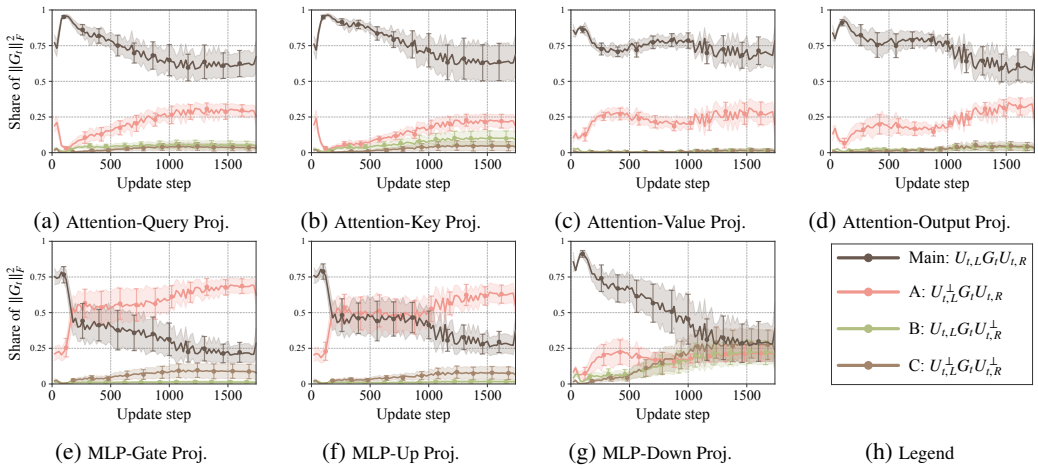
$$P_S = U_{t,S}U_{t,S}^\top, \qquad P_S^\perp = I - P_S, \qquad D_S = (\Lambda_{t,S} + \mu I)^{-1/2}, \qquad c_S = (\lambda_{S,r} + \mu)^{1/2}, \tag{9}$$

where $\lambda_{S,r}$ is the smallest retained eigenvalue. This gives

$$\begin{aligned} G_t^{\text{pre}} = {} & \underbrace{c_L c_R\, U_{t,L} D_L\, U_{t,L}^\top G_t U_{t,R}\, D_R U_{t,R}^\top}_{\texttt{Main}} + \underbrace{c_R\, P_L^\perp G_t U_{t,R}\, D_R U_{t,R}^\top}_{\texttt{A}} \\ & + \underbrace{c_L\, U_{t,L} D_L\, U_{t,L}^\top G_t P_R^\perp}_{\texttt{B}} + \underbrace{P_L^\perp G_t P_R^\perp}_{\texttt{C}}. \end{aligned} \tag{10}$$

Running Adam directly on G_t^{pre} would require $\mathcal{O}(mn)$ memory for its moments. Clean instead keeps separate moment statistics for each block of Eqn. (10) according to its size. For the `Main` block, Adam runs on the two-sided projection $U_{t,L}^\top G_t U_{t,R} \in \mathbb{R}^{r\times r}$. In addition, the cross components in blocks `A` and `B` lie outside one subspace but inside the other, so they are also low-dimensional:

$$G_{t,\texttt{A}} = P_L^\perp G_t U_{t,R} \in \mathbb{R}^{m\times r}, \qquad G_{t,\texttt{B}} = U_{t,L}^\top G_t P_R^\perp \in \mathbb{R}^{r\times n}. \tag{11}$$



(a) Attention-Query Proj. (b) Attention-Key Proj. (c) Attention-Value Proj. (d) Attention-Output Proj.

(e) MLP-Gate Proj. (f) MLP-Up Proj. (g) MLP-Down Proj. (h) Legend

Figure 3: Fraction of the gradient norm captured in each of the four blocks. Generally, `Main` and `A` are dominant; the contributions of `B` and `C` slowly increase only towards the end of training, where the gradients become noisier. Values are averaged across all 24 instances of a layer type in a LLaMA-350M architecture.

Only the component in block `C` is orthogonal to both subspaces and is a full $m \times n$ matrix. Rather than discarding the components outside the `Main` block, Clean applies Adam-style updates to each block with its own moment estimates. However, applying Adam to these blocks as in Romashov et al. (2026), requires buffers of size $\mathcal{O}(r^2)$, $\mathcal{O}(mr)$, $\mathcal{O}(rn)$, and $\mathcal{O}(mn)$ for `Main`, `A`, `B`, and `C`, respectively. The first three are low-rank, but the last is as large as running Adam directly on G_t^{pre}, reintroducing exactly the cost that Clean aims to remove.

Fig. 3 illustrates the fraction of the gradient norm, averaged across all 24 instances of a layer type in a LLaMA-350M, for each of the blocks in Eqn. (10). `Main` dominates across nearly all layers, while `C` consistently contributes less. The figure also underscores the necessity of reintegrating information from blocks other than the `Main`, as they carry a substantial portion of the training signal, particularly in later iterations. We therefore tailor the moment state to each block, as summarized in Table 1.

Table 1: **Moments per block.** $\mathbb{E}_\beta[\cdot]$ denotes an exponential moving average with rate $\beta \in \{\beta_1, \beta_2\}$, and memory counts stored floats. `Main` keeps both full moments; `A` and `B` keep full first moments, but sum their second moments over the long dimension, leaving one value per column or row; `C` keeps no first moment and a single scalar second moment, since per-entry states would cost $\mathcal{O}(mn)$.

Block	Adam's Input	1^{st} Moment M	2^{nd} Moment V	Memory
`Main`	$S_t = U_{t,L}^{\mathsf{T}} G_t U_{t,R} \in \mathbb{R}^{r\times r}$	$\mathbb{E}_{\beta_1}[S_t]$	$\mathbb{E}_{\beta_2}[S_t^{\odot 2}]$	$2r^2$
`A`	$G_{t,A} \in \mathbb{R}^{m\times r}$	$\mathbb{E}_{\beta_1}[G_{t,A}]$	$\mathbb{E}_{\beta_2}\left[\sum_i (G_{t,A})_{ij}^2\right]$	$mr + r$
`B`	$G_{t,B} \in \mathbb{R}^{r\times n}$	$\mathbb{E}_{\beta_1}[G_{t,B}]$	$\mathbb{E}_{\beta_2}\left[\sum_j (G_{t,B})_{ij}^2\right]$	$rn + r$
`C`	$C_t \in \mathbb{R}^{m\times n}$	none, raw C_t	$\mathbb{E}_{\beta_2}\left[\|C_t\|_F^2\right]$	1

For the `Main` block, we apply regular Adam, maintaining entry-wise first and second moments (M and V), which each require $\mathcal{O}(r^2)$ memory.

$$M_t \leftarrow \beta_1 M_{t-1} + (1-\beta_1) S_t, \quad V_t \leftarrow \beta_2 V_{t-1} + (1-\beta_2) S_t^{\odot 2}, \tag{12}$$

where $\odot$ denotes the Hadamard product. For blocks `A` and `B`, we retain the full-resolution first moments but spatially pool the second moments. Specifically, by assuming a shared scale for each column in `A` and each row in `B`, we reduce the second moment to a single scalar per column or row.

$$\begin{aligned} M_{t,\mathtt{A}} &\leftarrow \beta_1 M_{t-1,\mathtt{A}} + (1-\beta_1) G_{t,\mathtt{A}}, \quad v_{t,\mathtt{A}} \leftarrow \beta_2 v_{t-1,\mathtt{A}} + (1-\beta_2)\, \mathbf{1}^\top (G_{t,\mathtt{A}}^{\odot 2}) \\ M_{t,\mathtt{B}} &\leftarrow \beta_1 M_{t-1,\mathtt{B}} + (1-\beta_1) G_{t,\mathtt{B}}, \quad v_{t,\mathtt{B}} \leftarrow \beta_2 v_{t-1,\mathtt{B}} + (1-\beta_2)\, (G_{t,\mathtt{B}}^{\odot 2}) \mathbf{1} \end{aligned} \tag{13}$$

Finally, for block `C`, we omit the first moment entirely, utilizing the raw gradient instead, and employ its squared Frobenius norm as a scalar second moment.

Projection-aware adaptive moments. On subspace-refresh steps, when the bases $U_{t,S}$ have changed ($S \in \{L, R\}$), the moments will get misaligned. The first moments M, $M_{\mathtt{A}}$, and $M_{\mathtt{B}}$ are valid operators in the original coordinates, and they can be moved exactly into the new coordinates by:

$$\begin{aligned} M &\leftarrow (U_{t,L}^\top U_{t-1,L}) M (U_{t-1,R}^\top U_{t,R}), \\ M_{\mathtt{A}} \leftarrow U_{t,L}^\perp M_{\mathtt{A}} (U_{t-1,R}^\top U_{t,R}), &\qquad M_{\mathtt{B}} \leftarrow (U_{t,L}^\top U_{t-1,L}) M_{\mathtt{B}} U_{t,R}^\perp, \end{aligned} \tag{14}$$

However, the second moments cannot be exactly transformed. V is an exponential moving average of element-wise squared gradients, tracking only the diagonal (per-coordinate) variances. Since element-wise squaring does not commute with basis rotations, calculating the true rotated variances would require unrecorded off-diagonal cross-terms. Similar to SOAP (Vyas et al., 2024), we explicitly rotate the first moment M, and leave V in place. By relying on warm-started refreshes, we constrain the overall frame drift, thereby ensuring the stale-basis error in V remains sufficiently small.

For A and B, we transport the pooled second moments on change of basis. Since each value stores the squared gradient along one direction, we use the squared overlap between the old and new bases:

$$v_{\mathtt{A}}^\top \leftarrow v_{\mathtt{A}}^\top (U_{t-1,R}^\top U_{t,R})^{\odot 2}, \qquad v_{\mathtt{B}} \leftarrow (U_{t,L}^\top U_{t-1,L})^{\odot 2} v_{\mathtt{B}}. \tag{15}$$

Finally, the scalar $v_{\mathtt{C}}$ does not need to be rotated because it has no direction associated with it.

Projection back and parameter update. After applying the block-wise Adam-style updates, each block is projected back to the original parameter space to construct the final weight update, as shown in Eqn. (10). Note that the momentum terms used in this step are bias-corrected, and similar to SOAP Vyas et al. (2024), we are using Adam scaling factors while projecting back, which eliminates the need for D_S and c_S factors in reconstructing G_t^{pre} (10).

$$\widetilde{G}_t = \underbrace{U_{t,L}\mathcal{N}(\hat{M}_t, \hat{V}_t)U_{t,R}^\top}_{\text{Block Main (Full Adam)}} + \underbrace{\mathcal{N}(\hat{M}_{t,\mathtt{A}}, \hat{v}_{t,\mathtt{A}})U_{t,R}^\top}_{\text{Block A (Pooled Adam)}} + \underbrace{U_{t,L}\mathcal{N}(\hat{M}_{t,\mathtt{B}}, \hat{v}_{t,\mathtt{B}})}_{\text{Block B (Pooled Adam)}} + \underbrace{\mathcal{N}(\hat{C}_t, \hat{v}_{t,\mathtt{C}})}_{\text{Block C (Pooled Adam)}}, \tag{16}$$

where $\mathcal{N}(\hat{M}, \hat{V}) = \hat{M}/(\sqrt{\hat{V}} + \epsilon)$ for the corresponding bias-corrected moments $\hat{M}$ and $\hat{V}$. The weight update is then $W_t \leftarrow W_{t-1} - \alpha\eta \cdot \widetilde{G}_t$ where η is the learning rate and α is a scale factor.

Clean avoids building any $\mathcal{O}(mn)$ memory while leveraging full-rank information and maintaining performance by (i) representing preconditioners with Nyström factors $U_{t,*}, \Lambda_{t,*}$, (ii) operating on four complementary blocks, and (iii) updating the stored low-rank moments when bases change. Memory therefore grows only *linear* with respect to model dimensions. Furthermore, our warm-start Nyström variant (see Alg. 3 in Appendix A) enables gradual basis evolution, improving stability during periodic subspace refreshes. As shown in Table 2, Clean is the only evaluated optimizer that scales linearly in state memory.

Table 2: Optimizer-state memory requirement for an $m \times n$ matrix.

Optimizer	State size
Adam	$2mn$
SOAP	$2mn + 2m^2 + 2n^2$
Muon	mn
ALSO	$mn + 2mr + 2nr + \ r^2 + 2r + O(1)$
Clean (Ours)	$2mr + 2nr + 2r^2 + 4r + O(1)$

4 Q-CLEAN: LOW-PRECISION SECOND-ORDER OPTIMIZATION

To further reduce memory overhead, we explore the stability of Clean under quantization, a technique that stores matrices as low-bit integers. In this section, we briefly introduce Q-Clean, which employs quantized optimizer states. More details are included in Appendix B.

Following (Dettmers et al., 2021), block-wise quantization partitions a flattened matrix X into contiguous blocks of size B and quantizes each block independently. With an n-bit asymmetric (affine) quantizer, block i uses a scale s_i and a zero-point z_i to represent a value in lower precision.

Per-block scaling limits the effect of outliers to a single block rather than coarsening the quantization grid of the entire matrix (Dettmers et al., 2022). However, because optimizer states are continually updated, round-to-nearest scheme discards any update smaller than half a quantization step, so small but consistent updates never accumulate. To remove this bias, the linear quantizer can instead use

Algorithm 1 Clean at iteration t. Colors mark the blocks of Eqn. (10): Main, A, B, C. Between refreshes, the previous factors are reused.

Require: $G_t \in \mathbb{R}^{m\times n}$; step size η, scale α; $\beta_1, \beta_2, \epsilon$; rank r; refresh period T, $t = 1$

if $t - 1 \bmod T = 0$ **then**
  $U_{t,L}, \Lambda_{t,L} \leftarrow \mathrm{Nys}\left(\beta_2 U_{t-1,L}\Lambda_{t-1,L}U_{t-1,L}^\top + (1-\beta_2)G_tG_t^\top\right)$ ▷ Alg. 3
  $U_{t,R}, \Lambda_{t,R} \leftarrow \mathrm{Nys}\left(\beta_2 U_{t-1,R}\Lambda_{t-1,R}U_{t-1,R}^\top + (1-\beta_2)G_t^\top G_t\right)$ ▷ Alg. 3
  Rotate M, M_A, M_B, v_A, v_B into the new bases ▷ Eqn. (14) & Eqn. (15)
$U_{t,L}, \Lambda_{t,L} \leftarrow U_{t-1,L}, \Lambda_{t-1,L}$
$U_{t,R}, \Lambda_{t,R} \leftarrow U_{t-1,R}, \Lambda_{t-1,R}$

$S_t \leftarrow U_{t,L}^\top G_t U_{t,R}$, $G_{t,A} \leftarrow P_L^\perp G_t U_{t,R}$, $G_{t,B} \leftarrow U_{t,L}^\top G_t P_R^\perp$, $C_t \leftarrow P_L^\perp G_t P_R^\perp$ ▷ Eqn. (10)
M_t, V_t, $M_{t,A}$, $v_{t,A}$, $M_{t,B}$, $v_{t,B}$, $v_{t,C}$ ▷ Moments, Table 1
$\widetilde{G}_t \leftarrow U_{t,L}\,\mathcal{N}(M_t, V_t)\,U_{t,R}^\top + \mathcal{N}(M_{t,A}, v_{t,A})\,U_{t,R}^\top + U_{t,L}\,\mathcal{N}(M_{t,B}, v_{t,B}) + \mathcal{N}(C_t, v_{t,C})$ ▷ Eqn. (16)

$W_t \leftarrow W_{t-1} - \alpha\eta\,\widetilde{G}_t$

stochastic rounding (Von Neumann & Goldstine, 1947; Gupta et al., 2015), which is unbiased in expectation, and therefore lets small updates accumulate over time.

For the first-moment and projection matrices, we apply the block-wise linear asymmetric scheme mentioned above. In contrast, the second-moments are strictly non-negative and exhibit a highly skewed distribution with a large dynamic range (see Appendix B). Because linear quantization is suboptimal for such statistics, we instead adopt the non-linear quantization implementation from `bitsandbytes` (Dettmers et al., 2021), which is also used in the Adam-8bit optimizer. We quantize the first moments of all blocks listed in Table 1, the second moment of the `Main`, and the projection matrices in INT8. The second moments of blocks `A` and `B` are kept in full precision: each is a vector of size r, so its memory footprint is negligible and quantizing it would yield no meaningful savings.

5 Related Work

Low-rank Projection Methods. A popular approach to reducing optimizer memory is to project gradients onto a low-rank subspace. GaLore (Zhao et al., 2024) introduced this strategy by performing gradient updates within a low-rank subspace defined by the singular value decomposition (SVD) of gradients. A family of works follows this line of optimizers (Chen et al.; Zhu et al., 2025; Liang et al., 2024; He et al., 2024). While effective at saving memory, this one-sided projection can be unstable as the subspace drifts over time. LDAdam (Robert et al., 2024) and Subtrack++ (Rajabi et al., 2025) attempted to mitigate this by incorporating projection-aware updates and more robust subspace tracking mechanisms. However, these methods operate on one-sided approximations and do not capture the row-column geometry of the gradient covariances.

Second-order Preconditioning. Another line of work focuses on capturing curvature information. K-FAC (Martens & Grosse, 2015) approximates the Fisher information matrix with Kronecker products, while Shampoo (Gupta et al., 2018) directly approximates the full-matrix AdaGrad (Duchi et al., 2011) preconditioner by maintaining separate row and column preconditioners. These balanced, two-sided preconditioners are powerful but require storing full-sized covariance matrices, making them memory- and compute-intensive. SOAP (Vyas et al., 2024) combines Shampoo's preconditioning with first-order updates but still retains the costly full-matrix accumulators.

Low-rank Second-order Methods. To achieve the benefits of second-order preconditioning without the prohibitive memory costs, recent methods have proposed efficient second-order optimizers. Alice (Gong et al., 2026) provides a low-rank extension that converts general full-rank optimizers into memory-efficient approximations by relying on structured Fisher information matrix approximations. Building on this, ALSO (Romashov et al., 2026), extends toward a two-sided approximation of SOAP (Vyas et al., 2024), while compensating for the discarded residual blocks outside the core low-rank projection. Another direction leverages the varying importance of eigensubspaces; for instance, COSMOS (Liu et al., 2025) introduces a hybrid approach that applies memory-intensive SOAP updates exclusively to the leading eigensubspace of the gradient matrix, while utilizing a computationally cheaper, Muon-like update (Jordan et al., 2024) for the residual subspace.

Table 3: Final validation perplexity (lower is better) and optimizer memory when pre-training LLaMA models of different sizes. Clean achieves competitive perplexity compared to full SOAP, while using less memory than AdamW. Clean, Q-Clean and ALSO use rank $r = 256$ at 350M and $r = 512$ at 1.3B. Excluding SOAP, the best results are shown in **bold** and the second best are underlined.

Methods	LLaMA-350M			LLaMA-1.3B		
	Perplexity	Opt. Mem.	Peak Mem.	Perplexity	Opt. Mem.	Peak Mem.
AdamW (Kingma & Ba, 2014)	19.63	1.40G	47.93G	12.12	5.11G	54.13G
SOAP (Vyas et al., 2024)	15.12	7.64G	54.25G	11.52	13.62G	79.02G
Muon (Jordan et al., 2024)	16.97	0.83G	47.36G	11.78	2.80G	51.83G
ALSO (Romashov et al., 2026)	**15.36**	1.30G	47.88G	12.19	4.71G	53.82G
Clean (Ours)	15.87	1.25G	47.68G	**11.65**	4.48G	53.44G
Q-Clean (Ours)	16.44	0.38G	46.82G	12.10	1.28G	50.24G

Low Precision Training with Quantization. Quantization improves memory efficiency by representing data in reduced-precision formats. These techniques broadly fall into Post-Training Quantization (PTQ), where pre-trained models are quantized without additional training (Ashkboos et al., 2025; Dettmers et al., 2023; Ashkboos et al., 2024), and Quantization-Aware Training (QAT), which integrates quantization directly into the training loop. A major challenge in QAT is training instability. Inspired by Izmailov et al. (2018), SWALP (Yang et al., 2019) mitigates this using stochastic weight averaging, though it incurs significant memory overhead when applied to large foundation models. Q-Galore (Zhang et al., 2024) substantially reduces memory consumption by integrating full-model quantization with the low-rank gradient projection of GaLore (Zhao et al., 2024). Similarly, 4-bit Shampoo (Wang et al., 2024) quantizes the left and right eigenvector matrices and applies Björck orthonormalization to restore orthogonality after quantization.

6 Experiments

Setup. We evaluate Clean and Q-Clean on pre-training LLaMA-based architectures (Touvron et al., 2023) with 350M and 1.3B parameters on the C4 dataset (Raffel et al., 2020). All models are trained from scratch with a sequence length of 1024 and an effective batch size of 4096 (≈ 4.2M tokens per step), identical across optimizers. Following the Chinchilla rule (Hoffmann et al., 2022) of ≈ 20 training tokens per parameter, we train for 1750 steps (7.3B tokens) at 350M and 5000 steps (21.0B tokens) at 1.3B. We use a cosine schedule with 10% linear warmup decaying to $0.1\times$ the peak learning rate, weight decay 0.1, gradient clipping at 1.0, $\beta_1 = \beta_2 = 0.95$, and bfloat16 parameters and gradients. Training is carried out on nodes of up to four NVIDIA H100 80GB GPUs.

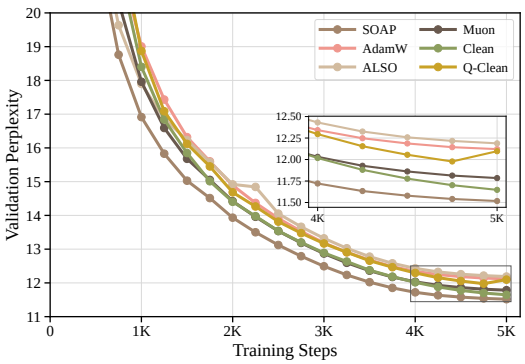
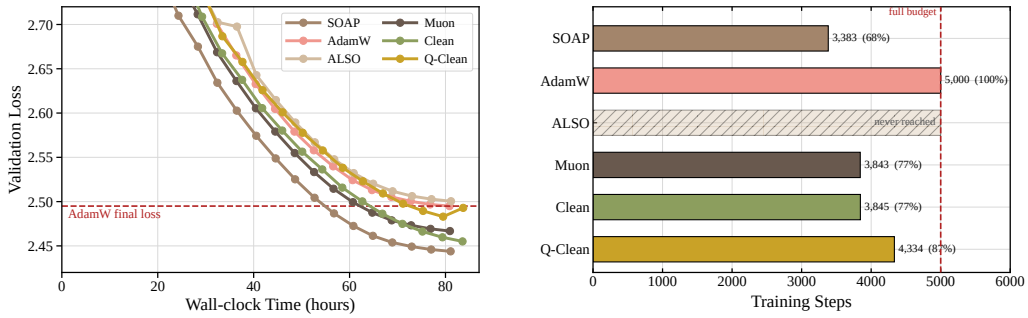


Figure 4: Validation perplexity recorded during the training of a LLaMA-1.3B architecture across various optimization methods.

Baselines and tuning. We compare against AdamW (Kingma & Ba, 2014), SOAP (Vyas et al., 2024), Muon (Jordan et al., 2024), and ALSO (Romashov et al., 2026). At 350M we sweep three learning rates per method and report the best run in Table 3. Due to the prohibitive cost of a full sweep at 1.3B, we performed a shortened sweep and resumed only the best configurations to 5,000 steps. Clean, Q-Clean and ALSO all use rank $r = 256$ at 350M and $r = 512$ at 1.3B, and all preconditioned methods refresh their preconditioner every $T = 10$ steps.

Results. Table 3 summarizes the final results, demonstrating that Clean achieves competitive performance compared to SOAP, at a fraction of its cost. Q-Clean cuts the optimizer state memory consumption by more than 50% compared to Muon, and still delivers results on par with the full-rank



(a) Validation loss against wall-clock time.

(b) Steps to reach AdamW's final loss.

Figure 5: Convergence efficiency on pre-training LLaMA-1.3B, all runs on four H100 GPUs with the same batch size, schedule and token budget. **(a) Wall-clock:** Clean reaches AdamW's final validation loss in 64 of the 81 hours AdamW needs, demonstrating 26% speedup. **(b) Token efficiency:** Clean reaches that loss in 77% of the training budget and Q-Clean in 87%.

methods. Fig. 4 plots validation perplexity throughout training for LLaMA-1.3B, where Clean follows SOAP closely and has the best performance after the SOAP.

Efficiency. Clean's memory advantage grows with scale (Figs. 7 and 8) because its memory complexity scales linearly with matrix dimensions, whereas every baseline carries at least one quadratic term (Table 2). Furthermore, Fig. 7 demonstrates a reduction in per-parameter memory cost as model size increases. See Appendix D for more details. To evaluate sample efficiency and convergence, Fig. 5 compares the methods across time and token budgets rather than focusing solely on the final loss. Clean reaches AdamW's final performance in 64 hours instead of 81, while simultaneously reducing the optimizer memory footprint. It also only consumes 77% of the overall token budget, with Q-Clean requiring 87%.

Ablations. To better investigate the sensitivity of Clean, we performed ablation studies on the LLaMA-350M pre-training, focusing on two key settings: the projection rank r and the preconditioner update interval T. In each experiment, we vary one parameter while keeping the other fixed at our default configuration ($T = 10$ and $r = 256$). First, Fig. 9a shows that the validation perplexity improves as the rank increases up to 256. While heavily compressed ranks ($r = 32, 64$) fail to catch up to the baselines, setting $r = 128$ achieves competitive results, and increasing the rank further to $r = 512$ yields no additional gains. Second, Fig. 9b demonstrates that $T = 10$ provides an optimal balance; while refreshing the preconditioners either too frequently ($T = 1$) or at large intervals ($T = 100$) negatively impacts the final performance.

7 Conclusion

We introduce Clean, a memory-efficient second-order optimizer that reduces the memory complexity of two-sided preconditioning from quadratic to linear. By leveraging randomized Nyström sketching, Clean accurately approximates left and right gradient covariances while uniquely reintegrating off-subspace residual blocks to preserve essential training signals. We also develop Q-Clean, a low-precision variant that aggressively compresses optimizer states.

Empirically, both Clean and Q-Clean achieve validation perplexities that are highly competitive with full-rank methods like SOAP, yet they operate with a smaller memory footprint than standard AdamW. Crucially, this aggressive memory reduction uniquely enables the pre-training of a 13B parameter architecture on a single 80GB GPU, and increases the convergence speed by 26% compared to AdamW. We are optimistic that these linear-scaling curvature approximations will encourage broader exploration into accessible, memory-efficient optimization frameworks.

AI USE STATEMENT

In this work, we used generative AI tools to provide feedback on our research methodology and to assist in implementing methods. We did not use generative AI for other tasks requiring disclosure, such as generating synthetic datasets, help develop theoretical models or conceptual frameworks, propose or refine hypothesis, design research methodology, assist in the writing of proofs, provide critical ingredients for proving mathematical claims, assist with translation, clean and reformat dataset, support qualitative and thematic data analysis, developing theoretical models, formulating or proving mathematical claims, or interpreting the final results.

Additionally, we utilized generative AI for several tasks with recommended disclosure: analyzing existing literature, creating scientific figures and images, and revising the paper draft to improve wording, flow, and overall readability.

We have thoroughly reviewed and verified all AI-assisted work. We take full responsibility for the final content of this work, including all text, claims, code, and visual artifacts produced with the aid of generative AI.

REFERENCES

Rohan Anil, Vineet Gupta, Tomer Koren, Kevin Regan, and Yoram Singer. Scalable second order optimization for deep learning. *arXiv preprint arXiv:2002.09018*, 2020.

Saleh Ashkboos, Amirkeivan Mohtashami, Maximilian L Croci, Bo Li, Pashmina Cameron, Martin Jaggi, Dan Alistarh, Torsten Hoefler, and James Hensman. Quarot: Outlier-free 4-bit inference in rotated llms. *Advances in Neural Information Processing Systems*, 37:100213–100240, 2024.

Saleh Ashkboos, Mahdi Nikdan, Soroush Tabesh, Roberto L Castro, Torsten Hoefler, and Dan Alistarh. Halo: Hadamard-assisted lower-precision optimization for llms. *arXiv preprint arXiv:2501.02625*, 2025.

Xi Chen, Kaituo Feng, Changsheng Li, Xunhao Lai, Xiangyu Yue, Ye Yuan, and Guoren Wang. Fira: Can we achieve full-rank training of llms under low-rank constraint?, 2024. *URL https://arxiv.org/abs/2410.01623*.

Tim Dettmers, Mike Lewis, Sam Shleifer, and Luke Zettlemoyer. 8-bit optimizers via block-wise quantization. *arXiv preprint arXiv:2110.02861*, 2021.

Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Gpt3. int8 (): 8-bit matrix multiplication for transformers at scale. *Advances in neural information processing systems*, 35: 30318–30332, 2022.

Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms. *Advances in neural information processing systems*, 36:10088–10115, 2023.

John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7), 2011.

Zachary Frangella, Joel A Tropp, and Madeleine Udell. Randomized nyström preconditioning. *SIAM Journal on Matrix Analysis and Applications*, 44(2):718–752, 2023.

Alex Gittens and Michael W Mahoney. Revisiting the nyström method for improved large-scale machine learning. *The Journal of Machine Learning Research*, 17(1):3977–4041, 2016.

Wenbo Gong, Meyer Scetbon, Chao Ma, and Edward Meeds. Towards efficient optimizer design for LLM via structured fisher approximation with a low-rank extension. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=KUFZXdem5R.

Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *International conference on machine learning*, pp. 1737–1746. PMLR, 2015.

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

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.

Yutong He, Pengrui Li, Yipeng Hu, Chuyan Chen, and Kun Yuan. Subspace optimization for large language models with convergence guarantees. *arXiv preprint arXiv:2410.11289*, 2024.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*, 2022.

Pavel Izmailov, Dmitrii Podoprikhin, Timur Garipov, Dmitry Vetrov, and Andrew Gordon Wilson. Averaging weights leads to wider optima and better generalization. *arXiv preprint arXiv:1803.05407*, 2018.

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/`.

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

Kaizhao Liang, Bo Liu, Lizhang Chen, and Qiang Liu. Memory-efficient llm training with online subspace descent. *Advances in Neural Information Processing Systems*, 37:64412–64432, 2024.

Liming Liu, Zhenghao Xu, Zixuan Zhang, Hao Kang, Zichong Li, Chen Liang, Weizhu Chen, and Tuo Zhao. Cosmos: A hybrid adaptive optimizer for memory-efficient training of llms. *arXiv preprint arXiv:2502.17410*, 2025.

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

Depen Morwani, Itai Shapira, Nikhil Vyas, Eran Malach, Sham Kakade, and Lucas Janson. A new perspective on shampoo's preconditioner. *arXiv preprint arXiv:2406.17748*, 2024.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of machine learning research*, 21(140):1–67, 2020.

Sahar Rajabi, Nayeema Nonta, and Sirisha Rambhatla. Subtrack++: Gradient subspace tracking for scalable llm training. *arXiv preprint arXiv:2502.01586*, 2025.

Thomas Robert, Mher Safaryan, Ionut-Vlad Modoranu, and Dan Alistarh. Ldadam: Adaptive optimization from low-dimensional gradient statistics. *arXiv preprint arXiv:2410.16103*, 2024.

Fedor Romashov, Nikolay Yudin, and Maxim Rakhuba. Two-sided low-rank SOAP for efficient LLM training. In *ICML'26 workshop on CoLoRAI - The 2nd Workshop on Connecting Low-rank Representations in AI*, 2026. URL `https://openreview.net/forum?id=INtGT8yune`.

Noam Shazeer and Mitchell Stern. Adafactor: Adaptive learning rates with sublinear memory cost. In *International Conference on Machine Learning*, pp. 4596–4604. PMLR, 2018.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

Joel A Tropp, Alp Yurtsever, Madeleine Udell, and Volkan Cevher. Fixed-rank approximation of a positive-semidefinite matrix from streaming data. *Advances in Neural Information Processing Systems*, 30, 2017.

John Von Neumann and Herman Heine Goldstine. Numerical inverting of matrices of high order. 1947.

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.

Sike Wang, Pan Zhou, Jia Li, and Hua Huang. 4-bit shampoo for memory-efficient network training. *Advances in Neural Information Processing Systems*, 37:126997–127029, 2024.

Guandao Yang, Tianyi Zhang, Polina Kirichenko, Junwen Bai, Andrew Gordon Wilson, and Chris De Sa. Swalp: Stochastic weight averaging in low precision training. In *International Conference on Machine Learning*, pp. 7015–7024. PMLR, 2019.

Zhenyu Zhang, Ajay Jaiswal, Lu Yin, Shiwei Liu, Jiawei Zhao, Yuandong Tian, and Zhangyang Wang. Q-galore: Quantized galore with int4 projection and layer-adaptive low-rank gradients. *arXiv preprint arXiv:2407.08296*, 2024.

Jiawei Zhao, Zhenyu Zhang, Beidi Chen, Zhangyang Wang, Anima Anandkumar, and Yuandong Tian. Galore: Memory-efficient llm training by gradient low-rank projection. *arXiv preprint arXiv:2403.03507*, 2024.

Hanqing Zhu, Zhenyu Zhang, Wenyan Cong, Xi Liu, Sem Park, Vikas Chandra, Bo Long, David Z. Pan, Zhangyang Wang, and Jinwon Lee. APOLLO: SGD-like memory, adamw-level performance. In *Eighth Conference on Machine Learning and Systems*, 2025. URL https://openreview.net/forum?id=mJrPkdcZDj.

A NYSTRÖM ALGORITHM

Algorithm 2 Randomized Nyström Approximation (RandNystromApprox)

Input: Matrix $G \in \mathbb{R}^{m \times n}$, rank size $r \leq m$, $\Omega \in \mathbb{R}^{m \times r}$.

$\Omega \sim \mathcal{N}(0, I) \in \mathbb{R}^{m \times r}$ {random Gaussian matrix}
$\Omega \leftarrow \text{QR}(\Omega, 0)$ {thin QR decomposition}
$Y \leftarrow G\Omega$
$\nu \leftarrow \text{eps}(\text{norm}(Y, 2))$ {compute shift}
$Y_\nu \leftarrow Y + \nu\Omega$ {add shift for stability}
$B \leftarrow \Omega^\mathsf{T} Y_\nu$
$C \leftarrow \text{CHOL}((B + B^\mathsf{T})/2)$ {Force symmetry and Cholesky decomposition}
$B \leftarrow Y_\nu / C$ {triangular solve}
$[U, \Sigma, \sim] = \text{SVD}(B, 0)$ {thin SVD}
$\hat{\Lambda} \leftarrow \max\{0, \Sigma^2 - \nu I\}$ {remove shift, compute eigs}
Output: $[U, \hat{\Lambda}]$

Nyström approximations are well-suited for optimizer states due to their computational efficiency and solid theoretical foundations (Gittens & Mahoney, 2016). While a direct implementation using Eqn. (4) suffers from numerical instability, Alg. 2 introduces a robust, numerically stable alternative for generating randomized Nyström approximations (Tropp et al., 2017).

Crucially, rather than sampling a purely random matrix at each step, we warm-start the sketch using the eigenvectors from the previous step (U_{prev}), augmented with p columns of fresh Gaussian noise to continuously explore new gradient directions. To further improve approximation accuracy, we perform q steps of power iteration with periodic orthogonalization. Finally, we apply sign alignment to the resulting eigenvectors to prevent abrupt subspace flips, ensuring temporal consistency across consecutive optimization steps. The pseudo-code of this updated method is presented in Alg. 3.

Algorithm 3 Warm-started Randomized Nyström with Subspace Iteration

Require: $A \in \mathbb{R}^{m \times n}$ PSD, target rank r, oversampling p, power-iterations passes q, warm start $U_{\text{prev}} \in \mathbb{R}^{m \times r}$ (or $\emptyset$)

if $U_{\text{prev}} \neq \emptyset$ **then**
  $\Psi \sim \mathcal{N}(0, I) \in \mathbb{R}^{m \times p}$; $\Omega \leftarrow [\, U_{\text{prev}},\ \Psi \,]$ {warm start + fresh exploration block}
else
  $\Omega \sim \mathcal{N}(0, I) \in \mathbb{R}^{m \times (r+p)}$
$\Omega \leftarrow \text{QR}(\Omega, 0)$
for $j = 1$ **to** q **do**
  $Y \leftarrow A\Omega$
  $\Omega \leftarrow \text{QR}(Y, 0)$ {re-orthonormalize *every* pass}
$Y \leftarrow A\Omega$ {final sketch must match the final Ω}
$\nu \leftarrow \text{eps}(\text{norm}(Y, 2))$
$Y_\nu \leftarrow Y + \nu\Omega$
$B \leftarrow \Omega^\mathsf{T} Y_\nu$
$C \leftarrow \text{CHOL}((B + B^\mathsf{T})/2)$
$[U, \Sigma, \sim] \leftarrow \text{SVD}(Y_\nu / C, 0)$
$\hat{\Lambda} \leftarrow \max\{0, \Sigma^2 - \nu I\}$
$U \leftarrow U_{:,1:r}$, $\hat{\Lambda} \leftarrow \hat{\Lambda}_{1:r,1:r}$ {discard oversampling}
$s_i \leftarrow \text{sign}(u_i^\mathsf{T} u_{\text{prev},i})$; $U_{:,i} \leftarrow s_i U_{:,i}$ {sign alignment}
Output: $[U, \hat{\Lambda}]$

B QUANTIZATION DETAILS

In this section, we explain the Q-Clean optimizer in more detail.

B.1 BLOCK-WISE n-BIT QUANTIZATION

Block-wise quantization was originally introduced by (Dettmers et al., 2021) for compressing optimizer states during training, and was later adapted by (Dettmers et al., 2023) to frozen base-model weights for memory-efficient LLM fine-tuning. We adopt the same block-wise principle but use a uniform affine quantizer, with an *asymmetric* formulation (scale and zero-point) for most matrices and a *symmetric* formulation for the second moment, as detailed below.

The block-wise scheme partitions a tensor into contiguous blocks and computes an independent scale s (and, in the asymmetric case, a zero-point z) for each block. The motivation is robustness to outliers: with a single tensor-wide scale, even one anomalously large value forces every quantization step to be coarse, mapping most typical values to only a handful of integer codes and losing most of the resolution (Dettmers et al., 2022). A per-block scale isolates this effect: only the block containing the outlier loses precision, while every other block keeps a fine-grained scale fitted to its own local data.

Asymmetric Quantization (Default). Given a tensor $X \in \mathbb{R}^{d_1 \times d_2}$, we flatten and partition it into $N = \lceil d_1 d_2 / B \rceil$ contiguous blocks $\{X^{(i)}\}_{i=1}^{N}$ of size at most B (the last block may be smaller). Each block is quantized independently with a per-block scale s_i and zero-point z_i computed from the block's full range:

$$s_i = \frac{\max(X^{(i)}) - \min(X^{(i)})}{2^n - 1}, \quad z_i = -\left\lfloor \min(X^{(i)})/s_i \right\rceil \tag{17}$$

Then, the quantization and dequantization maps are

$$\begin{aligned} X_q^{(i)} &= \text{Quant}_n^{\text{asym}}(X^{(i)}, s_i, z_i) = \text{clamp}\Big(\Big\lfloor \tfrac{X^{(i)}}{s_i} \Big\rceil + z_i,\ 0,\ 2^n - 1\Big), \\ \widehat{X}^{(i)} &= \text{Dequant}_n^{\text{asym}}(X_q^{(i)}, s_i, z_i) = s_i\big(X_q^{(i)} - z_i\big) \ \approx\ X^{(i)} \end{aligned} \tag{18}$$

where $\lfloor \cdot \rceil$ denotes round-to-nearest and n is the number of bits. We apply this affine form with the stochastic-rounding explained in section B.2 to the projection matrices ($U_{t,L}$ and $U_{t,R}$) and to all first-moment matrices (M_t, $M_{t,A}$, and $M_{t,B}$). However, we do not use this quantization for the second-moment matrices, because their strictly non-negative, highly skewed distributions and large dynamic ranges make linear methods suboptimal. Instead, we apply a non-linear symmetric quantization approach for the main second moment. The second moments of blocks `A` and `B` remain in full precision; as simple size-r vectors, their memory footprint is already negligible and quantizing them offers no meaningful savings.

Non-linear Quantization for the Second-moment. Because the second moment statistics are strictly non-negative ($X^{(i)} \geq 0$) and exhibit a highly skewed distribution with a large dynamic range, standard linear quantization allocates insufficient resolution to the majority of values concentrated near zero. To address this, we utilize the non-linear quantization scheme from `bitsandbytes` (Dettmers et al., 2021).

Rather than spacing bins equally, this method employs a fixed non-linear codebook $\mathcal{C}$ consisting of 256 quantization levels (for 8-bit representation). These levels are dynamically spaced to provide high precision for small-magnitude values while remaining capable of capturing large outliers. For a given block, we first compute the maximum value to serve as the block-wise scaling factor:

$$s_i = \max(X^{(i)}) \tag{19}$$

The elements of the block are normalized by s_i to lie in the range $[0, 1]$. Each normalized value is then mapped to the nearest entry in the codebook $\mathcal{C}$. The quantization and dequantization mappings are formalized as:

$$\begin{aligned} X_q^{(i)} &= \text{Quant}_{\text{8-bit}}^{\text{NL}}(X^{(i)}, s_i) = \arg\min_{c \in \mathcal{C}} \left| \frac{X^{(i)}}{s_i} - c \right|, \\ \widehat{X}^{(i)} &= \text{Dequant}_{\text{8-bit}}^{\text{NL}}(X_q^{(i)}, s_i) = s_i \cdot X_q^{(i)}, \end{aligned} \tag{20}$$

where $X_q^{(i)}$ is stored in memory as an 8-bit integer corresponding to the index of the matched codebook entry c. This formulation eliminates the need for a zero-point, requiring only the per-block scale s_i as metadata, while perfectly matching the non-linear, non-negative distribution of the optimizer's second moments.

B.2 Maintaining High-precision Training with Stochastic Rounding Accumulated Quantization

To reduce memory footprint, we store optimizer states as tensors in low precision (INT8) throughout the forward and backward passes. Doing so, however, introduces a subtle but fatal failure mode: under round-to-nearest (RTN), if the update magnitude is not large enough, the effect will be mitigated by RTN. Following Q-GaLore (Zhang et al., 2024), we apply stochastic rounding (SR) (Von Neumann & Goldstine, 1947; Gupta et al., 2015) in our asymmetric quantization. A floating-point value X lying between two adjacent quantization levels $\lfloor X \rfloor$ and $\lceil X \rceil$ is rounded *up* to $\lceil X \rceil$ with probability $X - \lfloor X \rfloor$ and rounded *down* to $\lfloor X \rfloor$ otherwise. This rule is unbiased, i.e. $\mathbb{E}(X_q) = X$. More concretely, the two rounding schemes, namely rounding-to-nearest (RTN) and stochastic rounding, are defined as follows, respectively:

$$\lfloor X \rceil = \begin{cases} \lfloor X \rfloor & \text{if } \lfloor X \rfloor \leq X < \lfloor X \rfloor + \frac{\epsilon}{2}, \\ \lfloor X \rfloor + \epsilon & \text{if } \lfloor X \rfloor + \frac{\epsilon}{2} \leq X \leq \lfloor X \rfloor + \epsilon, \end{cases} \tag{21}$$

$$\text{round}(X) = \begin{cases} \lfloor X \rfloor & \text{with probability } \lceil X \rceil - X, \\ \lceil X \rceil & \text{with probability } X - \lfloor X \rfloor, \end{cases} \tag{22}$$

where $\epsilon = 2^{-\text{FL}}$ and FL denotes the length of the fractional part.

B.3 Quantizer Choice

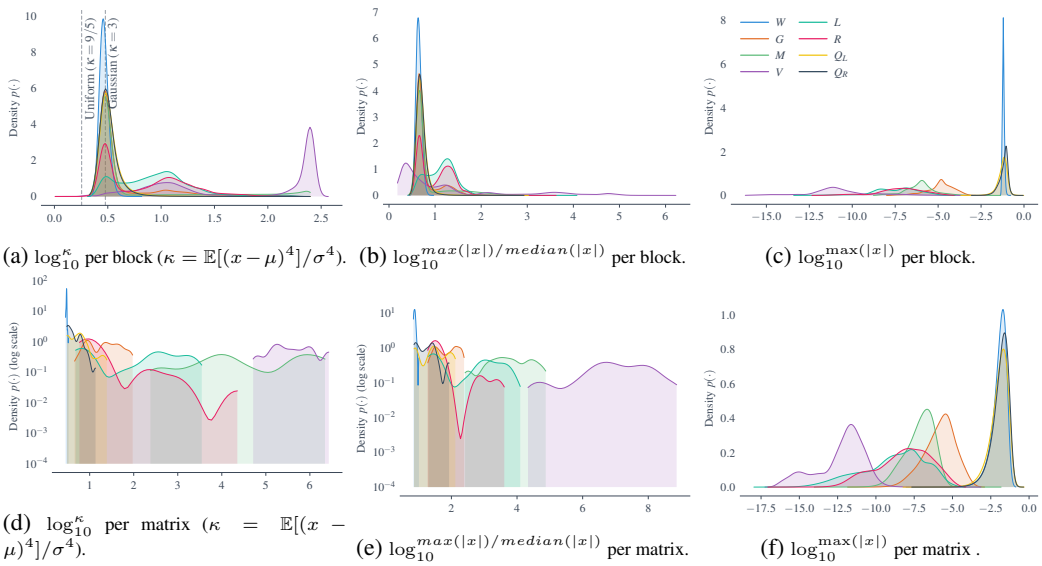
We tracked every matrix family W (weights), G (gradients), M and V (Adam moments), L and R (SOAP's accumulated covariance matrices), and Q_L and Q_R (eigenbasis of L and R) at every layer of the LLaMA-350M model while optimizing via SOAP (see Alg. 4). For each family, we then pool all captured matrices and report empirical densities of three quantities: scale ($\log_{10} \max |x|$), dynamic range $\left(\log_{10}^{\left(\max |x| / \text{median}|x|\right)} \right)$, and the non-excess kurtosis ($\log_{10} \kappa$), with $\kappa = \mathbb{E}[(x-\mu)^4]/\sigma^4$.

As shown in Figure 6, we evaluate the statistics using two perspectives based on the quantization strategy. The per-matrix view computes a single scalar for an entire matrix, highlighting differences between matrices; this applies when sharing one quantization scale across a matrix. Conversely, the per-block view divides matrices into blocks, computing and pooling statistics for each block individually.

Together, these views motivate our quantization strategy. In the per-matrix view, overall matrices (V, L, R, etc.) exhibit extreme distributional probabilities; a global linear scale stretches to cover these heavy tails, wasting resolution and causing high error. Moving to a block-wise approach enhances the tiles by isolating these extreme values, which lightens the tails across all families. For most matrices (W, G, M, L, R, U_L, U_R), this shift makes them suitable for asymmetric and linear quantization. However, even after block-wise quantization, the second-moment matrix (V) still shows heavy tails, and linear quantization leads to divergence on V, whereas it succeeds for the other matrices. Therefore, we use a symmetric and non-linear quantizer for V and retain linear block-wise codes for the rest in our low-rank proposed method.

C SOAP

Algorithm 4 presents the pseudocode for the SOAP optimizer (Vyas et al., 2024). As shown, the full L and R covariance matrices are constructed and stored, while the Adam optimizer is applied to the gradient projected onto the eigenbasis. Although SOAP demonstrates accelerated convergence compared to other standard optimizers, it introduces prohibitive memory requirements by simultaneously maintaining both the covariance matrices and the full Adam moments.



(a) $\log_{10}^{\kappa}$ per block ($\kappa = \mathbb{E}[(x-\mu)^4]/\sigma^4$). (b) $\log_{10}^{max(|x|)/median(|x|)}$ per block. (c) $\log_{10}^{\max(|x|)}$ per block.

(d) $\log_{10}^{\kappa}$ per matrix ($\kappa = \mathbb{E}[(x-\mu)^4]/\sigma^4$). (e) $\log_{10}^{max(|x|)/median(|x|)}$ per matrix. (f) $\log_{10}^{\max(|x|)}$ per matrix .

Figure 6: The plots compare non-excess kurtosis, dynamic range, and scale under two quantization granularities: per-block (a–c) and per-matrix (d–f). The shift from a per-matrix to a per-block approach significantly reduces the heavy-tailed distributions for most matrices, rendering them suitable for linear quantization. However, the second moment matrix (V, in purple) remains a prominent outlier, exhibiting persistent heavy tails even in the per-block view, which motivates the use of a non-linear quantizer specifically for V.

Algorithm 4 SOAP

Sample batch B_t.
$G \in \mathbb{R}^{m \times n} \leftarrow -\nabla_W \phi_{B_t}(W_t)$
$G' \leftarrow Q_L^T G Q_R$
$M \leftarrow \beta_1 M + (1-\beta_1) G$
$M' \leftarrow Q_L^T M Q_R$
{Now we "run" Adam on G'}
$V \leftarrow \beta_2 V + (1-\beta_2)(G' \odot G')$ {Elementwise multiplication}
$N' \leftarrow \frac{M'}{\sqrt{\hat{V}_t}+\epsilon}$ {Elementwise division and square root}
{Now that we have preconditioned by Adam in the rotated space, we go back to the original space.}
$N \leftarrow Q_L N' Q_R^T$
$W \leftarrow W - \eta N$
{End of gradient step, we now update L and R and possibly also Q_L and Q_R. }
$L \leftarrow \beta_2 L + (1-\beta_2) G G^T$
$R \leftarrow \beta_2 R + (1-\beta_2) G^T G$
if t % f == 0 **then**
    $Q_L \leftarrow \texttt{Eigenvectors}(L, Q_L)$
    $Q_R \leftarrow \texttt{Eigenvectors}(R, Q_R)$

D Linear Memory Discussion

The linear memory complexity of Clean and Q-Clean with respect to model dimensions makes them exceptionally scalable.

As demonstrated in Figure 7 (left), Clean and Q-Clean are the only optimizers that allow pre-training a LLaMA-13B architecture on a single 80 GB H100 GPU. Notably, the other low-rank second-order baseline, ALSO (Romashov et al., 2026), requires more than 80 GB of memory even at rank $r = 32$.

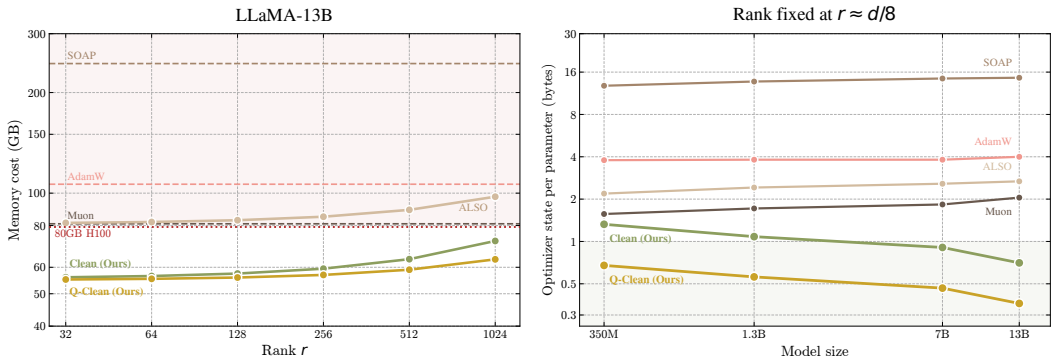


Figure 7: **Left:** total memory for pre-training LLaMA-13B on a single 80 GB H100 as a function of rank; the shaded region is infeasible for all of our baselines at any rank. **Right:** optimizer state per parameter across model sizes at $r \approx d/8$. Every baseline carries an mn term (Table 2) and so stays flat or rises; Clean and Q-Clean scale as $r(m+n)$ and are the only methods whose per-parameter cost *decreases* with model size.

This is due to the fact that Clean keeps no matrix with dimensions of order $\mathcal{O}(mn)$ or $\mathcal{O}(m^2+n^2)$, while all other optimizers have at least one $\mathcal{O}(mn)$ component.

Furthermore, Figure 7 (right) illustrates the cost per parameter for different optimizers. Remarkably, our methods are the only ones that reduce this per-parameter cost compared to the baselines. To ensure a realistic setup, we scale the rank proportionally with the model dimension, setting $r = d/8$, where $d = \min(m, n)$ is the smaller dimension of each weight matrix. This trend is highly promising in terms of democratizing training of much larger foundation models on more accessible hardware. Additionally, Figure 8 illustrates the effect of varying ranks across different model sizes for Clean, Q-Clean, and the low-rank second-order baseline, ALSO (Romashov et al., 2026). Figure 8 (left) demonstrates that for a fixed rank of $r = 32$, the memory savings gap between our methods and the baseline increases as the model size scales. This improvement stems from the omission of quadratic memory terms, allowing our methods to operate at a significantly lower cost. Furthermore, Figure 8 (right) compares how the memory consumption of ALSO scales relative to our methods across different ranks and model dimensions. As expected, the memory efficiency gap widens as the model size increases.

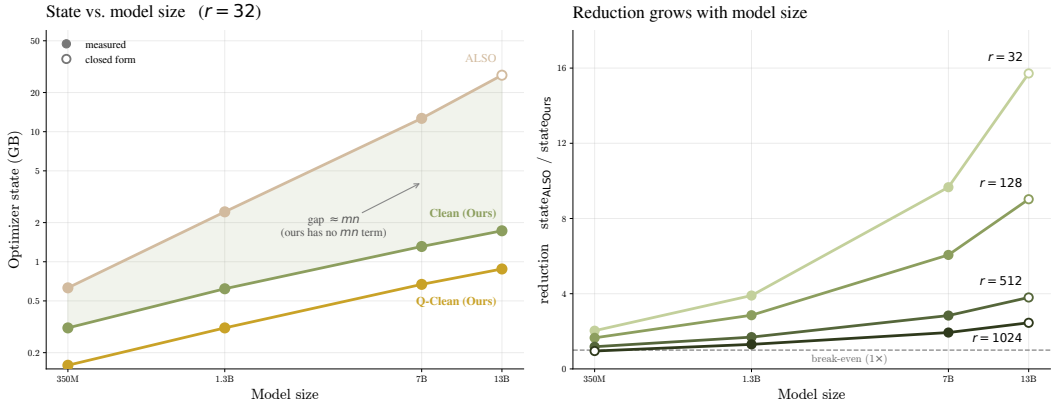
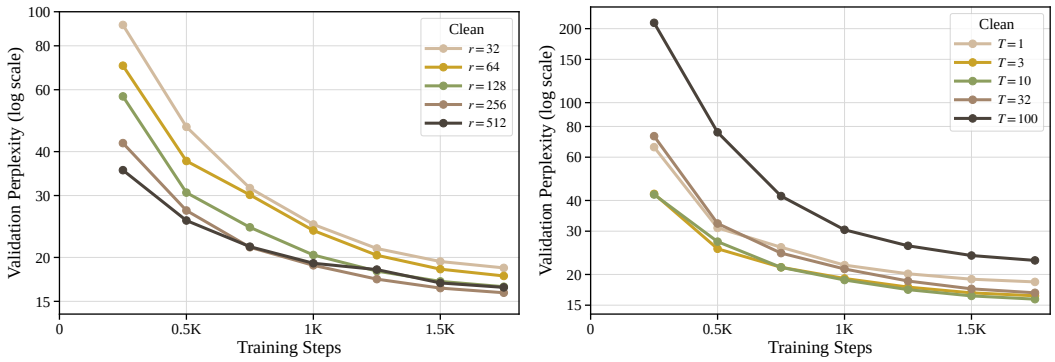


Figure 8: **Measured optimizer state on LLaMA.** Baselines carry an mn term; ours replaces it with $r(r+2)$, so our state is smaller whenever $mn > r(r+2)$. The margin widens with scale because mn grows with the model (2.8M to 70.8M from 350M to 13B) while $r(r+2)$ is fixed by the rank. **Left:** the shaded gap is the omitted mn term at r=32. **Right:** reduction over ALSO, 15.7× at r=32 and 3.8× at r=512 on 13B. Batch 1, seq 1024, bf16 state; filled markers measured, hollow closed-form where the run exceeded 80 GB in ALSO.

E ABLATING HYPERPARAMETERS

Here we provide the supporting figures for the hyperparameter ablation studies discussed in section 6. Fig. 9a presents the learning curves and validation perplexities for varying projection ranks (r) on the LLaMA-350M pre-training task. Figure 9b details the performance across different preconditioner update intervals (T). These figures illustrate the full training trajectories, confirming the optimal configurations of $r = 256$ and $T = 10$ established in our primary analysis.



(a) Rank r in projection matrices.

(b) Preconditioner update frequency T.

Figure 9: Ablations of Clean on pre-training LLaMA-350M. **(a) Rank r:** Performance improves as rank increases up to 128 with marginal improvements afterward. **(b) Update interval T:** $T = 10$ is optimal; updating too often ($T = 1$) or too rarely ($T = 100$) negatively impacts performance.

F HYPERPARAMETERS

Tables 4 and 6 list the hyperparameters of all runs. SOAP, Clean and Q-Clean optimize all parameters themselves: 1-D parameters receive Adam updates with the method's own (β_1, β_2). Q-Clean quantizes the eigenbases and all first moments, including the compensation moments, with asymmetric per-group linear quantization and stochastic rounding, and the second moments with block-wise dynamic quantization; the eigenvalues and the pooled compensation second moments stay in original data type.

Table 4: Model architectures and training setup shared by all optimizers. Gradient accumulation makes up the global batch. Peak memory in Table 3 is measured on a single GPU at this micro-batch size, without activation checkpointing.

	LLaMA-350M	LLaMA-1.3B
Architecture		
Parameters	368M	1.34B
Hidden dimension	1024	2048
Intermediate dimension	2736	5461
Attention heads	16	32
Layers	24	24
Vocabulary size	32000 (LLaMA tokenizer)	
Training		
Dataset	C4 (English)	
Sequence length	1024	
Batch size (sequences)	4096	
Micro-batch size	32	16
Training steps	1750	5000
Warmup steps	175	500
Learning-rate schedule	linear warmup, cosine decay to $0.1\times$ peak	
Weight decay (decoupled, all parameters)	0.1	
Gradient clipping (global norm)	1.0	
Adam ϵ	10^{-8}	
Precision	bf16 (weights, gradients, activations)	
Evaluation	C4 validation, $\approx$10M tokens	

Table 5: Learning rates swept at 350M. Three values per optimizer; the selected value, used in Table 6 and reported in Table 3, is shown in **bold**.

Method	Learning rate		
AdamW	7×10^{-4}	$\mathbf{1.4 \times 10^{-3}}$	2.8×10^{-3}
SOAP	7×10^{-4}	1.4×10^{-3}	$\mathbf{2.8 \times 10^{-3}}$
Muon	2×10^{-3}	6×10^{-3}	$\mathbf{1.0 \times 10^{-2}}$
ALSO	1.4×10^{-3}	$\mathbf{4.4 \times 10^{-3}}$	1.4×10^{-2}
Clean (Ours)	1.4×10^{-3}	$\mathbf{4.4 \times 10^{-3}}$	1.4×10^{-2}
Q-Clean (Ours)	1.4×10^{-3}	$\mathbf{4.4 \times 10^{-3}}$	1.4×10^{-2}

Table 6: Optimizer-specific hyperparameters; learning rates are peak values of the schedule in Table 4.

Method	Hyperparameter	LLaMA-350M	LLaMA-1.3B
AdamW	Learning rate	1.4×10^{-3}	1.4×10^{-3}
	(β_1, β_2)	$(0.95, 0.95)$	
SOAP	Learning rate	2.8×10^{-3}	2.8×10^{-3}
	(β_1, β_2)	$(0.95, 0.95)$	
	Shampoo β	0.95	
	Preconditioning frequency	10	
Muon	Learning rate	1.0×10^{-2}	1.4×10^{-2}
	Momentum (Nesterov)	0.9	
	Newton–Schulz iterations	5	
ALSO	Learning rate	1.4×10^{-3}	4.0×10^{-3}
	Rank r	256	512
	Basis update interval K	10	
	$(\beta_1, \beta_2, \beta_3)$	$(0.95, 0.95, 0.95)$	
	Compensation weight α_c	1.0	
	Norm-growth limit γ	1.01	
Clean (Ours)	Learning rate	4.4×10^{-3}	1.0×10^{-3}
	(β_1, β_2)	$(0.95, 0.95)$	
	Rank r	256	512
	Oversampling p	8	16
	Power iterations	1	
	Preconditioning frequency T	10	
	Shampoo β (per refresh)	0.95	
	Max. precond. dimension	4096	8192
	Norm-growth limit γ	1.01	
Q-Clean (Ours)	Learning rate	4.4×10^{-3}	1.0×10^{-3}
	Other hyperparameters	as Clean	
	Eigenbases U_L, U_R	INT8 linear, group 256, SR	
	First moments	INT8 linear, group 256, SR	
	Second moments	INT8 dynamic, block 256	
	Eigenvalues	as Clean	