GEFEN: OPTIMIZED STOCHASTIC OPTIMIZER

Anonymous authors
Paper under double-blind review

ABSTRACT

AdamW is a default optimizer for modern deep learning, but its first and second moment states add roughly two parameter-sized buffers to training memory, increasing the already substantial cost of large-scale pretraining. We propose Gefen, a memory-efficient optimizer that automatically shares second-moment estimates across parameter blocks and quantizes the first moment using a learned codebook, thereby reducing AdamW's memory footprint by up to $8\times$ while maintaining the same performance, corresponding to a reduction of 6.5 GiB per billion parameters. Prior work reduces optimizer memory by sharing second moments across parameters grouped along the Hessian's block-diagonal structure, but relies on hand-specified architectural rules and leaves unexplained why such grouping works. We prove that large mixed Hessian entries constrain the ratio of squared gradients toward one. This supplies the missing explanation, since a shared second moment is accurate when the squared gradients it pools are similar. It also shows the Hessian need not be computed: a block-structured Hessian forces the squared gradients to inherit that structure, so the blocks can be found in the squared gradients themselves. Gefen therefore infers block structure from the initial squared gradients, requiring no architecture-specific metadata or user-tuned hyperparameters beyond AdamW defaults. Gefen learns an exact histogram-based dynamic-programming quantization codebook and reuses the same blocks for first-moment scaling. Across diverse pretraining experiments, Gefen achieves the lowest peak optimizer memory among the compared methods that maintain AdamW-level performance. In single-machine and distributed training, the reduced footprint enables larger microbatches and substantially improves throughput over AdamW, making Gefen a drop-in replacement that can train larger models or use larger global batch sizes. We provide the complete Python implementation, including fused CUDA kernels.

1 INTRODUCTION

Adam (Kingma & Ba, 2014) and AdamW (Loshchilov & Hutter, 2017) are workhorse optimizers in deep learning due to their strong and consistent performance across a broad range of architectures and datasets. However, because they store both first and second moment moving averages, their optimizer state alone adds roughly twice the parameter memory. Reducing optimizer memory can enable training larger models or using larger batch sizes, which can in turn improve training throughput and final model quality. Therefore, we are motivated to design an optimizer that maintains AdamW-like performance while reducing memory usage.

Prior work on optimizers varies in the criteria it evaluates, the model architectures it targets, and the metrics it reports, making cross-paper comparisons challenging. Here, we set forth desiderata for an optimizer: their order does not imply importance, and some elements may be more or less relevant depending on the use case. In the following, we use *performance* to denote the relevant evaluation metric, such as validation or training loss.

- **Peak Memory Footprint**. The additional memory beyond model parameters, activations, and gradients should be as small as possible. For instance, a model with 1B parameters stores the same number of gradients, and Adam additionally stores the same number of first and second moment values, resulting in two parameter-sized buffers of extra optimizer state. Since hardware memory is typically fixed during training, peak memory footprint is

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.

a primary constraint on trainable model size, maximum batch size, and often throughput. Note that peak memory footprint is different from the persistent memory of the optimizer, as the optimizer can temporarily allocate intermediates, thereby requiring more memory than the final persistent state.

- **Performance versus Data**. A main goal of an optimizer is to get to the best possible performance as fast as possible (convergence speed). However, performance versus wall-clock time depends on the machine configuration, preventing reproducible comparisons across machines, and is therefore better decomposed into two measurements: (1) performance versus data, which is hardware-agnostic, and (2) throughput in units of $data/time$. Combining the two gives performance versus time. Plots of *performance* versus iteration count are less informative, since different methods may process different amounts of data per step (e.g., due to different batch sizes). Furthermore, in many cases, training data is limited, so the primary objective is to maximize performance for a fixed data budget (sample efficiency), and the x-axis should represent the amount of training data consumed rather than time. For language models, a typical comparison shows the *performance* (e.g., validation loss) as a function of the amount of training data (e.g., tokens or epochs) consumed.
- **Throughput**. Throughput in units of $data/time$ (e.g., tokens/second) can be estimated from shorter runs under the assumption that it remains approximately stable during training. Throughput should be measured over the full training iteration, including the forward and backward passes, data loading, and host-to-device memory transfer. An optimizer that enables larger batch sizes, reduces communication overhead in distributed training, or reaches comparable performance in fewer iterations can deliver substantial practical gains.
- **Generality**. A good optimizer should perform robustly across diverse use cases, model architectures, datasets, and hardware configurations. It should require minimal optimizer-specific hyperparameters or manual configuration choices.
- **Theoretical Grounding**. Although many theoretical analyses of optimizers rely on strong assumptions that may not hold in practice (e.g., convexity of the loss landscape), theoretical grounding can still provide useful intuition and increase confidence in the method, even in constrained settings.
- **Distributability**. A good optimizer should be compatible with common distributed-training frameworks and sharding strategies, including DDP, FSDP, and others.

In this paper, we seek to design an optimizer with a *lower peak memory* footprint than AdamW and similar or better *performance* and *throughput*, while remaining *general* and *distributable*.

Adam-mini (Zhang et al., 2025) is a recent variant of Adam that reduces optimizer memory by *sharing* second-moment values using manual rules that mimic the block diagonal structure of parameters' Hessians, but a theoretical justification for *why* Hessian-aligned grouping is beneficial remains missing. In practice, users may need to provide architecture-specific information (e.g., the number of attention heads), and grouping rules depend on tensor names, which may vary across implementations. Quantized optimizers (Dettmers et al., 2021; Li et al., 2023) reduce memory, but rely on manually crafted, suboptimal codebooks and introduce an implicit hyperparameter through the quantization block size.

Inspired by these methods, we introduce Gefen, an optimizer that shares second-moment estimates across automatically inferred parameter blocks. **(1)** We supply the missing justification: we prove that strong Hessian coupling constrains squared gradients to be similar, so Hessian-aligned grouping works precisely because it pools parameters whose squared gradients agree, which is the condition under which a single shared second moment is accurate for every member of a block. The same result also shows that the Hessian need not be computed: since the Hessian is block-structured, the squared gradients must inherit that structure, so the blocks can be recovered from the squared gradients themselves. **(2)** Our method infers periodic parameter blocks from initial squared gradients, without architecture-specific information or tensor-name-dependent rules. These blocks define both second-moment sharing and first-moment quantization scales, avoiding an additional block-size hyperparameter. **(3)** We further reduce memory by learning a first-moment quantization codebook using exact histogram-based dynamic programming. Together, these contributions reduce memory beyond Adam-mini while retaining general applicability and AdamW-level performance across both LLM and non-LLM models. After reviewing preliminaries and related work in Section 2, we

provide theoretical analysis in Section 3 showing that high Hessian affinity implies similar squared gradients, which motivates our grouping strategy. Section 4 presents our algorithm, which automatically groups parameters and shares second moments within these groups. The experiments in Section 5 demonstrate that Gefen, a drop-in replacement for AdamW, achieves the lowest peak optimizer memory among the compared methods that match AdamW's performance, reducing memory by up to $8\times$ relative to AdamW without learning-rate retuning or additional hyperparameters, while enabling higher throughput. Readers familiar with AdamW may skip directly to Section 3.

2 Preliminaries and Related Work

Adam and AdamW. Combining ideas from AdaGrad (Duchi et al., 2011) and RMSProp (Tieleman & Hinton, 2012), Adam is an adaptive first-order optimizer that maintains exponential moving averages of the gradient and squared gradient. At step t, with gradient $g_t = \nabla_\theta \mathcal{L}(\theta_t)$, Adam computes

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t \qquad \text{(EMA of first moment; momentum term)} \tag{1}$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \qquad \text{(EMA of second raw moment)} \tag{2}$$

then applies bias correction, which is needed because both EMAs are initialized at zero and are therefore biased toward zero at early steps,

$$\hat{m}_t = \frac{m_t}{1-{\beta_1}^t}, \qquad \hat{v}_t = \frac{v_t}{1-{\beta_2}^t} \qquad \text{(Bias correction)} \tag{3}$$

and updates the parameters using learning rate η as

$$\theta_{t+1} = \theta_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \tag{4}$$

AdamW uses the same adaptive moments, but applies weight decay directly in parameter space using coefficient λ, rather than injecting a penalty term into the gradient:

$$\theta_{t+1} = \theta_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} - \eta\lambda\theta_t \tag{5}$$

This decoupling avoids interactions between adaptive gradient scaling and regularization strength, and typically yields better generalization and more stable hyperparameter tuning. In modern deep learning practice, Adam and AdamW are the de facto standard optimizers. They keep first and second moment states, typically in FP32, so optimizer-state memory is roughly 8 bytes per parameter. For example, for a 13B-parameter model, Adam-style optimizer states alone require about $13 \times 10^9 \times 8 \approx 97$ GiB; this can make training more difficult on memory-constrained GPUs and force smaller batch sizes, which may also reduce training throughput.

Quantized optimizer states. Recent work reduces optimizer memory by quantizing Adam-style moment states. Adam8bit (Dettmers et al., 2021) flattens tensors into fixed-size blocks ($B = 2048$), quantizes both first and second moments, and stores per-block absolute-max scaling (recomputed each step) together with 8-bit values per element. Its Dynamic Tree Quantization encodes normalized values in $[-1, 1]$, providing higher precision near zero while still representing larger magnitudes. However, this design has practical limitations: (1) the dynamic-tree quantizer can be wasteful due to duplicated codebook entries, (2) in practice the method is not applicable to all tensor types (e.g., embeddings should not be quantized), so it is not fully plug-and-play as a universal drop-in, and (3) the fixed user-chosen block size introduces an additional hyperparameter burden. Adam4bit (Li et al., 2023) pushes state quantization to 4 bits with improved handling of outliers (including finer-grained and asymmetric treatments for moment statistics), further reducing memory while preserving competitive accuracy in many settings. The method includes a block-size hyperparameter, which defaults to 128 in the authors' implementation.

See Appendix G for an extended related work discussion, including additional optimizers and gradient communication compression methods. In this work, we focus on general-purpose optimizers that require no manual configuration, additional hyperparameters, or learning-rate retuning, and that can be used as drop-in replacements for AdamW across training regimes while achieving a significantly lower memory footprint.

3 THEORETICAL ANALYSIS

AdamW maintains, for each parameter, an exponential moving average of its squared gradient. A single such estimate can be shared across a group of parameters with minimal loss of accuracy when their squared gradients are similar. We therefore analyze conditions under which two entries of a parameter tensor with a large-magnitude mixed Hessian entry also have similar squared-gradient magnitudes. This connects the criterion we can measure, squared-gradient similarity, to the Hessian-based coupling that prior work targets with hand-specified, architecture-dependent rules.

Theorem 3.1 (Large Hessian entries contract squared-gradient ratios)**.** *Consider the two-layer MLP $z = W_1 x + b_1$, $y = W_2\sigma(z) + b_2$, where $y \in \mathbb{R}^{d_{\text{out}}}$, with MSE loss $L = d_{\text{out}}^{-1}\sum_{j=1}^{d_{\text{out}}}(y_j - t_j)^2$. Let $e_j := y_j - t_j$. Define:*

$$A_k := \sum_{i=1}^{d_{\text{out}}} e_i W_2[i,k], \quad U_{k,l} := |x_l \sigma'(z_k)|, \quad C_{k,k'} := \left|\frac{A_k}{A_{k'}}\right|, \quad D_{k,k'} := \left|\sum_i W_2[i,k]W_2[i,k']\right|$$

Assume:

$$D_{k,k'} \le \alpha, \quad U_{k,l} \le \beta, \quad |\log C_{k,k'}| \le \gamma, \quad \forall k,l,k',l' \text{ s.t. } k \ne k',\ A_k, A_{k'} \ne 0,\ H_{(k,l),(k',l')} \ne 0$$

For any such pair of entries $W_1[k,l]$ and $W_1[k',l']$, let $g_{k,l} := \partial L/\partial W_1[k,l]$ and define the ratio between their squared gradients and their mixed Hessian entry as

$$R := \frac{g_{k,l}^2}{g_{k',l'}^2}, \qquad H_{(k,l),(k',l')} := \frac{\partial^2 L}{\partial W_1[k,l]\,\partial W_1[k',l']}$$

Then, for $A := 2\alpha\beta^2/d_{\text{out}}$,

$$e^{-2\gamma}\left(\frac{|H_{(k,l),(k',l')}|}{A}\right)^2 \le R \le e^{2\gamma}\left(\frac{A}{|H_{(k,l),(k',l')}|}\right)^2$$

The theorem implies that for feasible $|H_{(k,l),(k',l')}| \le A$, larger mixed Hessian magnitude tightens the admissible range of the squared-gradient ratio around 1. The same form of contraction also holds for second-layer weights that share the same output coordinate and for hidden-bias terms, whereas output-bias Hessian entries are either zero or constant. Empirically, the same phenomenon is observed in real network architectures, including attention tensors and convolutional layers, as shown in Appendix B.

Theorem 3.2 (Within-layer Hessian affinity controls gradient magnitudes)**.** *Let $f_\theta(x) \in \mathbb{R}^m$ be a feedforward ReLU network, and let $L(\theta) = \ell(f_\theta(x))$, where $\ell : \mathbb{R}^m \to \mathbb{R}$ is twice differentiable and convex. Write $g = \nabla_\theta L(\theta)$, $H = \nabla_\theta^2 L(\theta)$, and $z = f_\theta(x)$, and assume that $\nabla\ell(z)$ is in the column span of $\nabla^2\ell(z)$. Further assume that θ is away from ReLU activation boundaries for this input, i.e., all preactivations are nonzero. Then for any two parameters i, j belonging to the same layer, we have $H_{ii} \ge 0, H_{jj} \ge 0$, and for the normalized Hessian affinity $\rho_{ij} = |H_{ij}|/\sqrt{H_{ii}H_{jj}}$ we have*

$$\left|\frac{g_i^2}{H_{ii}} - \frac{g_j^2}{H_{jj}}\right| \le 2\lambda(z)^2\sqrt{2(1-\rho_{ij})},$$

where $\lambda(z)$ is the Newton decrement at z, namely $\lambda(z)^2 = \nabla\ell(z)^\top\left(\nabla^2\ell(z)\right)^\dagger\nabla\ell(z)$. Here, we use the convention that $g_i^2/H_{ii} = 0$ whenever $H_{ii} = 0$, and similarly $\rho_{ij} = 0$ whenever either $H_{ii} = 0$ or $H_{jj} = 0$.

Thus, if the affinity ρ_{ij} is close to one, then the Hessian-normalized gradient magnitudes of coordinates i and j are close. If, additionally, H_{ii} and H_{jj} are comparable, then the raw gradient magnitudes $|g_i|$ and $|g_j|$ are comparable up to the same curvature scale. In particular, for the squared loss $\ell(z) = o^{-1}\|z - y\|^2$, we have $\nabla\ell(z) = \frac{2}{o}(z - y)$, $\nabla^2\ell(z) = \frac{2}{o}I$, and therefore the Newton decrement satisfies

$$\lambda(z)^2 = \nabla\ell(z)^\top\left(\nabla^2\ell(z)\right)^{-1}\nabla\ell(z) = \frac{2}{o}\|z - y\|^2 = 2\ell(z).$$

Hence, the theorem gives, at any point such that $\ell(z) > 0$,

$$\left|\frac{g_i^2}{H_{ii}} - \frac{g_j^2}{H_{jj}}\right| \le 4\ell(z)\sqrt{2(1-\rho_{ij})}.$$

Proofs are in Appendix A.

Algorithm 1 Gefen: our proposed algorithm for optimized stochastic optimization. Default settings are identical to AdamW. All operations on vectors are element-wise. The algorithm has a fused CUDA implementation to avoid unnecessary temporary memory allocations.

Require: $\alpha, \lambda \in \mathbb{R}$: Stepsize and Weight decay, $\beta_1, \beta_2 \in [0, 1)$: Decay rates for moment estimates. $f(\theta)$: Stochastic objective function with parameters θ. Initialize $t \leftarrow 0$, $\boldsymbol{G}_0 = \nabla_\theta f_t(\boldsymbol{\theta}_0)$
for each parameter-tensor θ run AUTOMATIC BLOCK PARTITIONING using $\boldsymbol{G}_0$
Run EXACT DP QUANTIZATION CODEBOOK LEARNING.
for each parameter-tensor θ **do**
    Initialize quantized 1st momentum codebook $\bar{\boldsymbol{m}}_0 \leftarrow \mathbf{0}$ per weight and $||\boldsymbol{m}_0||_\infty$ per block
    Initialize a single 2nd moment scalar $\bar{\boldsymbol{v}}_0 \leftarrow 0$ per block
repeat
  $t \leftarrow t + 1$
  for each parameter-tensor θ **do**
    $\boldsymbol{g}_t \leftarrow \nabla_\theta f_t(\boldsymbol{\theta}_{t-1})$ ▷ Get gradients for the current parameter
    $\boldsymbol{m}_t \leftarrow \beta_1 \cdot \text{DEQUANTIZE}(\bar{\boldsymbol{m}}_{t-1}, ||\boldsymbol{m}_{t-1}||_\infty) + (1 - \beta_1) \cdot \boldsymbol{g}_t$ ▷ Update biased momentum estimate
    $\bar{\boldsymbol{m}}_t, ||\boldsymbol{m}_t||_\infty \leftarrow \text{QUANTIZE}(\boldsymbol{m}_t)$ ▷ Quantize and recalculate absmax per block
    $\hat{\boldsymbol{m}}_t \leftarrow \boldsymbol{m}_t / (1 - {\beta_1}^t)$ ▷ Bias correction for momentum
    $\boldsymbol{v}_t \leftarrow \beta_2 \cdot \boldsymbol{v}_{t-1} + (1 - \beta_2) \cdot mean(\boldsymbol{g}_t^2)$ ▷ Update biased second moment using a mean per block
    $\hat{\boldsymbol{v}}_t \leftarrow \boldsymbol{v}_t / (1 - {\beta_2}^t)$ ▷ Bias correction of compact second moment estimate
    $\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} - \alpha \cdot \left(\hat{\boldsymbol{m}}_t / (\sqrt{\hat{\boldsymbol{v}}_t} + \epsilon) + \lambda \boldsymbol{\theta}_{t-1}\right)$ ▷ Compact $\hat{\boldsymbol{v}}_t$ are broadcasted to full shape
until *stopping criterion is met*
return optimized parameters $\boldsymbol{\theta}_t$

4 ALGORITHM

Computing the Hessian directly in large models is not tractable, and it is also unnecessary: the accuracy of a shared second moment is governed by the similarity of the squared gradients it pools, which we can measure at the first step. Section 3 explains why such groups exist and align with architecture, since Hessian-coupled entries are forced to have similar squared gradients, so the structure our algorithm detects is the same structure prior work specifies by hand. We therefore present Algorithm 1, a method that partitions each parameter tensor into blocks whose squared gradients are similar, and shares the moving average of the second moment within each block in a fully automatic way, without relying on user-specified configuration. We further reduce the memory footprint of the first moment using a *learnable* and *exact* dynamic programming quantization, which *reuses* the same block partitioning and therefore does not introduce an additional scaling constant.

Algorithm 2: AUTOMATIC BLOCK PARTITIONING

Require: first-step gradient $\boldsymbol{g}_0$ for a parameter, $n = |\boldsymbol{g}_0|$, ϵ
Flatten $\boldsymbol{g}_0$ and let $\mathcal{P}$ be all divisors p of n, sorted increasingly
$\boldsymbol{z} \leftarrow \log(\boldsymbol{g}_0^2 + \epsilon)$, $\mathcal{C} \leftarrow \emptyset$
for each candidate period $p \in \mathcal{P}$ **do**
    Reshape $\boldsymbol{z}$ into $\boldsymbol{B} \in \mathbb{R}^{(n/p) \times p}$
    $\mu_i(p) \leftarrow \text{mean}(\boldsymbol{B}_{i,:}) \quad \forall i \in \{1, \ldots, n/p\}$
    $S(p) \leftarrow \text{mean}_i \, |\mu_{i+1}(p) - \mu_i(p)|$ ▷ Equation (6)
for each $p \in \mathcal{P}$ s.t. $1 < p < n$ **do**
    if $S(p)$ is larger than the scores of its neighboring candidate periods **then**
        Add p to $\mathcal{C}$
return $\arg\max_{p \in \mathcal{C}} S(p)$ if $\mathcal{C} \neq \emptyset$, otherwise 1

4.1 AUTOMATIC PARTITIONING

As shown in Figure 1 and by Collobert (2004); Zhang et al. (2025), the Hessians of some parameters exhibit block-diagonal structure during pretraining. Splitting this parameter into 4 blocks, corresponding to the number of attention heads in this example, groups together parameters with stronger Hessian coupling, which, as suggested by the theorem and empirical evidence, leads to more similar squared gradients. Splitting into 16 groups provides finer granularity, as shown in the figure. However, computing the two-dimensional Hessian of mixed gradients is not tractable in large models. We therefore use the observed structure of the squared gradients: the same 16-block partition produces regions with more similar squared-gradient magnitudes. This observation motivates our one-dimensional squared-gradient-based algorithm for partitioning parameters into blocks, without computing the Hessian directly. Appendix E dis-

Algorithm 3 EXACT DP QUANTIZATION CODEBOOK LEARNING

Require: gradients $\{\boldsymbol{g}_0^{(\theta)}\}$, automatic periods $\{p_\theta\}$
Set number of codebook entries $k = 256$. Initialize histogram counts $\boldsymbol{c} \in \mathbb{R}^{16k}$ over $[-1, 1]$
for each parameter tensor θ **do**
  Reshape $\boldsymbol{g}_0^{(\theta)}$ into blocks $\boldsymbol{B} \in \mathbb{R}^{(n_\theta/p_\theta)\times p_\theta}$
  Normalize each nonzero block by its maximum absolute value: $\boldsymbol{Z}_{i,:} \leftarrow \boldsymbol{B}_{i,:}/\|\boldsymbol{B}_{i,:}\|_\infty$
  Accumulate all entries of $\boldsymbol{Z}$ into histogram counts $\boldsymbol{c}$
Let $\{(m_i, c_i)\}_{i=1}^{b}$ be the nonempty histogram bin centers and counts, sorted by m_i
Compute prefix sums of c_i, $c_i m_i$, and $c_i m_i^2$
Define the interval cost function $C(b_\ell, b_r, k')$ for assigning bins $[b_\ell, b_r]$ to codebook index k': use center -1.0 if $k' = 1$, center 1.0 if $k' = k$, and otherwise use the weighted mean of bins $[b_\ell, b_r]$ ▷ Appendix D
$D_{b_r,k'}$:= the cost for quantizing the first b_r bins using the first k' codepoints; initialize $D_{0,0} \leftarrow 0$ and all other entries to ∞
for $k' = 1, \ldots, k$ **do**
  for $b_r = k', \ldots, b$ **do**
    $D_{b_r,k'} \leftarrow \min_{b_\ell \le b_r} \; D_{b_\ell - 1, k'-1} + C(b_\ell, b_r, k')$
    Store the minimizing split point
Backtrack the stored splits to recover the k codebook entries
return the sorted codebook

cusses how this squared-gradient periodicity differs between pretraining and finetuning. For each parameter tensor, Gefen infers the sharing pattern from the first-iteration gradient only. Let $\boldsymbol{g}_0 \in \mathbb{R}^n$ be the *flattened* gradient of a parameter tensor at the first optimization step. For every divisor p of n we reshape $\log(\boldsymbol{g}_0^2 + \epsilon)$ into consecutive blocks of size p and measure the average contrast between adjacent block means:

$$\mu_i(p) = \frac{1}{p} \sum_{j=(i-1)p+1}^{ip} \log(g_{0,j}^2 + \epsilon)$$

$$S(p) = \frac{1}{n/p - 1} \sum_{i=1}^{n/p-1} |\mu_{i+1}(p) - \mu_i(p)| \tag{6}$$

For large n, the number-of-divisors function $\tau(n)$ satisfies the classical bound $\tau(n) \le n^{(\ln 2 + o(1))/\ln\ln n}$, showing that the number of divisors grows subpolynomially in n (Landau, 1909), so the number of candidate periods is low. For example, the maximum value of $\tau(n)$ over the intervals $[0.9 \cdot 10^6, 1.1 \cdot 10^6]$ and $[0.9 \cdot 10^9, 1.1 \cdot 10^9]$ is only 256 and 1344, respectively. Large values of $S(p)$ indicate that the block boundaries induced by period p separate regions with different squared-gradient magnitudes. Ideally, we want large p for reduced memory while still aligning block boundaries with structure in the squared gradients.

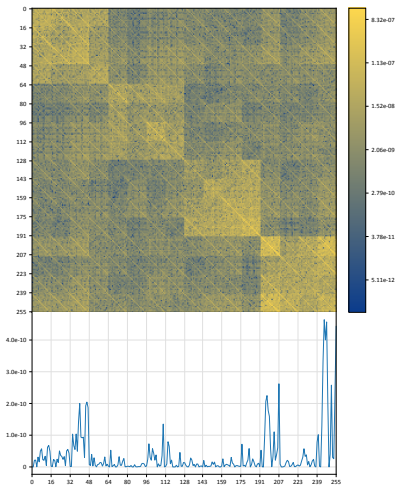


Figure 1: Hessian of a toy transformer attention-query parameter, showing block-diagonal structure, and the corresponding squared gradients of the same parameter below. Qualitatively, both exhibit a visible 16-block structure.

Some divisors align better with the structure of the Hessian and therefore produce pronounced local peaks in $S(p)$. We sort the candidate periods in increasing order, keep only strict local maxima, and choose the remaining period with the largest score. If no such candidate exists, we set $p = 1$. The selected period is fixed for the remainder of training. Algorithm 2 provides the pseudocode of the algorithm. The period we found has dual use: it is used to share the second moment, and also to quantize the first moment, as described in the next section.

4.2 Exact Quantization of Momentum Using Histogram-Based Dynamic Programming

To reduce first-moment memory, we learn a codebook using exact histogram-based dynamic programming. Unlike the fixed quantization schemes of Adam8bit and Adam4bit, our method reuses the existing block partitioning without an additional quantization block-size hyperparameter. Lloyd-Max (Lloyd, 1982; Max, 1960) is a classical algorithm for scalar quantization. It iteratively alternates between assigning each value to the nearest codebook entry and updating each codebook entry to be the mean of the values assigned to it. However, we identify two major problems with this approach. The first problem is *contraction*. Because Lloyd-Max learns its codebook from the observed values, the two extreme codebook entries are not necessarily fixed at the endpoints of the normalized range $[-1, 1]$. If we recompute the quantization scale at every optimizer step, this can create an undesirable feedback loop: after dequantization, the largest stored value is bounded by the largest learned codebook entry, which may be strictly smaller than 1. The next absmax estimate is then smaller as well, causing the effective quantization range to shrink over time. To solve this problem, we force the codebook to always include the two extreme entries -1 and 1, which guarantees that the quantization range is preserved and prevents contraction. Another issue is that we found Lloyd-Max to be sensitive to the codebook initialization strategy: linear spacing or initialization by quantiles can lead to different results, with no consistent winner. See Appendix C.1 for more details.

One-dimensional Lloyd-Max quantization is not guaranteed to converge to the optimal codebook solution and is sensitive to the codebook initialization strategy (Appendix C.1). We therefore seek an optimal solution. The quantization problem is NP-hard in dimension two or more (Aloise et al., 2009; Dasgupta, 2008; Mahajan et al., 2012); however, for the one-dimensional case, inspired by Wang & Song (2011), we design an optimal dynamic programming solution that operates on the histogram of the first-step gradients and runs in $O(b^2k)$, where b is the number of bins in the histogram and k is the number of quantization levels. Histogram calculation is performed in CUDA kernels, and dynamic programming is performed on the CPU using Numba (Lam et al., 2015), allowing the codebook learning process, which takes place only once at the beginning of training, to finish quickly. See Algorithm 3 and Appendix D for more details on the algorithm. See Appendix C.1 for a comparison of the exact method with the Lloyd-Max heuristic, and Section 6 for ablations.

5 Experiments

We evaluate Gefen across the desiderata: peak memory footprint, performance, throughput, and distributability, using a variety of model architectures, model sizes, and datasets. Both in theory and in implementation, Gefen reduces the peak and persistent memory of the optimizer state by $8\times$ in full precision training, while achieving performance similar to AdamW. In the evaluated distributed-training configurations, Gefen improves FSDP throughput by 56% compared to AdamW and enables the DDP setting where AdamW cannot fit even a microbatch size of 1. Alternatively, one can use the reduced memory footprint to increase model size. Implementation details and hyperparameters are found in Appendix M.

Table 1: Memory footprint normalized by parameter memory. All methods use momentum. The two best values are bolded. As the model size grows, Gefen has a **lower** memory footprint than all Adam variants.

Optimizer	CNN (1.2M)		GPT-2 (125M)	
	Persistent mem	Peak mem	Persistent mem	**Peak mem**
AdamW	x2.00	x2.00	x2.01	x2.01
Adam8bit	x0.51	x0.51	x0.51	x0.51
Adam4bit	**x0.27**	**x0.27**	**x0.30**	**x0.30**
Adam-mini	x1.00	x2.97	x1.01	x1.02
Adafactor	x1.02	x3.05	x1.01	x3.01
Muon	x1.00	x3.12	x1.01	x1.63
SM3	x1.01	x2.02	x1.01	x1.84
Gefen	**x0.27**	**x0.29**	**x0.25**	**x0.25**

5.1 Peak Memory Footprint

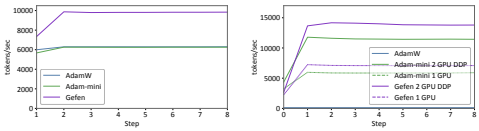
We begin with the first criterion in the desiderata above: the optimizer's peak memory footprint. We compare Gefen against the methods described in Section 2, along with a few representative methods from Appendix G, and report the results in Table 1. Consistent with the algorithm description and our empirical measurements, Gefen reduces AdamW's peak and persistent 32-bit memory footprints by $\sim 8\times$. Although Adam4bit has a compelling memory footprint, as shown next, it achieves suboptimal quality compared with the other methods.

Table 2: CNN on MNIST.

Optimizer	Validation Loss ↓
AdamW	$\mathbf{0.0282_{\pm 0.0023}}$
Adam8bit	$\mathbf{0.0290_{\pm 0.0020}}$
Adam4bit	$0.0313_{\pm 0.0013}$
Adam-mini	$0.0321_{\pm 0.0004}$
Adafactor	$0.0398_{\pm 0.0051}$
Muon	$0.0351_{\pm 0.0003}$
SM3	$0.0455_{\pm 0.0012}$
Gefen	$\mathbf{0.0283_{\pm 0.0024}}$

In the next section, for the Llama 3-1.5B model, the total optimizer state is 11.2 GiB with AdamW and 1.5 GiB with Gefen, a reduction of 9.7 GiB that can be crucial in memory-constrained settings. For example, training Llama 8B can be done on four A6000 GPUs with Gefen, whereas with AdamW we had to use four H100 GPUs due to insufficient memory.

Qwen3-4B pre-training used 16-bit mixed precision, where AdamW's optimizer states occupied 14.99 GiB of VRAM per GPU, compared with 3.75 GiB for Gefen, a $4\times$ reduction, saving 11.24 GiB per GPU. Next, we focus mainly on optimizers whose *peak* memory footprint is $\times 1$ or lower, i.e., at least a $2\times$ reduction relative to AdamW.

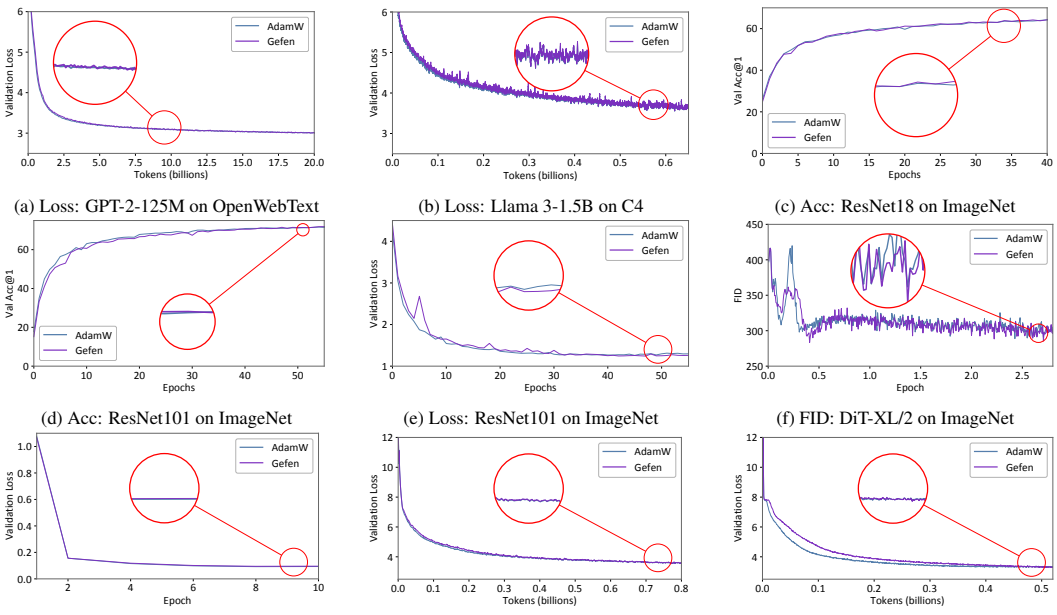


(a) Llama 3-1.5B with FSDP (b) GPT2-1B with DDP

Figure 2: Training throughput in two distributed settings. Gefen improves FSDP throughput by 56%, and in DDP and single-GPU settings enables training where AdamW is infeasible and improves throughput by 21% over Adam-mini.

5.2 Performance

Figure 3 shows performance across several settings: Llama 3-1.5B and Llama 8B on C4, Qwen3-4B on FineWeb-Edu, GPT-2-125M on OpenWebText, ResNet18, ResNet101, and DiT-XL/2 on ImageNet, CNN on MNIST and DDPM on Bollywood-Celebs. Gefen performs on par with AdamW across the settings. Notably, in all experiments, we use the same learning rate for Gefen as AdamW to demonstrate that our method can serve as a drop-in replacement. In Table 2 it can be seen that not all methods are AdamW learning rate compatible.



(a) Loss: GPT-2-125M on OpenWebText (b) Loss: Llama 3-1.5B on C4 (c) Acc: ResNet18 on ImageNet

(d) Acc: ResNet101 on ImageNet (e) Loss: ResNet101 on ImageNet (f) FID: DiT-XL/2 on ImageNet

(g) Loss: DDPM (Diffusion) (h) Loss: Llama 8B on C4 (i) Loss: Qwen3-4B on FineWeb-Edu

Figure 3: Training curves of GPT-2-125M on OpenWebText, Llama 3-1.5B and Llama 8B on C4, Qwen3-4B on FineWeb-Edu, ResNet18 and ResNet101 on ImageNet, DiT-XL/2 on ImageNet, and DDPM on Bollywood-Celebs. Gefen performs on par with AdamW across the settings. Additional graphs can be seen in Appendix H.

5.3 Throughput and Distributability

We next evaluate whether the reduced memory footprint translates into practical throughput gains and whether the method is compatible with distributed training schemes. Since optimizer-step time is only one component of the full training iteration, we measure end-to-end training throughput in tokens per second under fixed global batch size and sequence length. In

both distributed settings, the lower optimizer-state memory of Gefen allows a larger microbatch to fit on each GPU, reducing gradient accumulation overhead and increasing throughput. As shown in Figure 2, this yields a 56% throughput improvement over other methods for Llama 3-1.5B with FSDP sharding on C4, by allowing the microbatch size per GPU to increase from 1 to 2, which is not possible with the other methods due to their larger memory footprint. Adam4bit and Adam8bit implementations do not support FSDP sharding (Liang et al., 2025). For the GPT2-1B model, under both DDP on two 24GiB GPUs and single-machine training, AdamW is infeasible even with a microbatch size of 1. Adam-mini supports a microbatch size of 1, whereas Gefen supports a microbatch size of 2, yielding a 21% throughput improvement over Adam-mini while making this setting feasible compared with AdamW. We compare optimizer distributability in more detail in Appendix F.

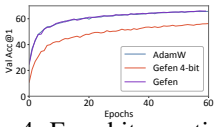
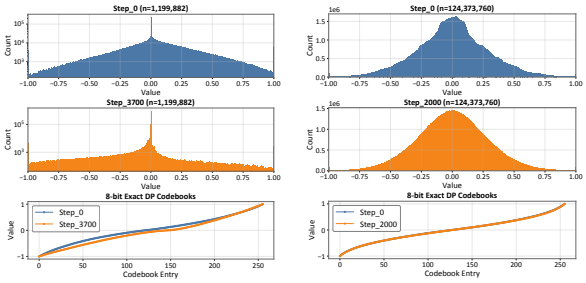


Figure 4: Four-bit quantization.

6 Ablation Study

Getting Down to Four Bits. While taking Gefen down to four bits works well in many situations, our goal is a general-purpose optimizer, so we cannot accept performance degradation on some tasks. Figure 4 shows a ResNet18 pretraining example in which reducing quantization to four bits harms performance. The same phenomenon can be seen for other four-bit methods in Figure 13.

Stable Quantization. A natural question for the ablation study is whether it is sufficient to learn the quantization codebook only at the beginning of training. Figure 5 shows the gradient distributions at different stages of training and their corresponding codebooks. The codebooks are relatively stable, suggesting that they can be reused throughout training. See Appendix C for more ablations.



(a) CNN on MNIST. (b) GPT-2 (125M) on OpenWebText.

Figure 5: Top: Histograms of gradient values after block-wise absmax normalization at different stages of training. Bottom: Corresponding learned codebooks at the same stages. The codebooks are relatively stable throughout training, allowing us to learn the codebook once.

7 Conclusion

In this work, we presented Gefen, an optimizer that significantly reduces AdamW's second-moment memory by sharing second-moment estimates in parameter regions where the Hessian, or the squared-gradient proxy used by our method, exhibits periodic structure; empirically, this structure appears across the pretraining tasks we considered. We first formulated desiderata for practical memory-efficient optimizers, and then provided theoretical results showing that strong Hessian coupling makes squared gradients, and hence AdamW second moments, close to each other, motivating shared second-moment estimates. Since computing Hessians is impractical at scale, Gefen uses first-step gradients to expose useful information about network topology without requiring user-provided architecture metadata or additional hyperparameters. It detects periodicity, groups parameters accordingly, shares second moments within the resulting blocks, and reuses the same structure to quantize first moments using an exact dynamic-programming algorithm for codebook construction. Among the baselines that reduce AdamW's peak memory footprint by $2\times$ or more, Gefen matches AdamW's performance, requires no learning-rate adjustment or additional hyperparameters, and achieves low memory footprint (up to $8\times$ lower than AdamW) and high throughput. In practice, switching from AdamW to Gefen requires changing only a single line of code, since the optimizer's arguments are fully compatible. Gefen can also be combined with other optimization methods, as discussed in Appendix I.

Reproducibility statement

We provide the complete implementation including fused CUDA kernels. Algorithmic details are given in Section 4 and Appendix D, proofs in Appendix A, additional experimental analysis in Appendices B to H, and complete experimental settings and hyperparameters in Appendix M.

AI use statement

In this work, artificial intelligence tools were used for grammar correction, identifying formatting and style issues, rephrasing text, validating and cleaning code, and implementing specific software functions. All artifacts were reviewed and verified by the authors. We take responsibility for the final content of this work.

References

Daniel Aloise, Amit Deshpande, Pierre Hansen, and Preyas Popat. Np-hardness of euclidean sum-of-squares clustering. *Machine learning*, 75(2):245–248, 2009.

Rohan Anil, Vineet Gupta, Tomer Koren, and Yoram Singer. Memory efficient adaptive optimization. *Advances in Neural Information Processing Systems*, 32, 2019.

Axolotl. Axolotl: Open source llm post-training, 2023. URL `https://github.com/axolotl-ai-cloud/axolotl`.

Åke Björck and Clazett Bowie. An iterative algorithm for computing the best estimate of an orthogonal matrix. *SIAM Journal on Numerical Analysis*, 8(2):358–364, 1971.

Xiangning Chen, Chen Liang, Da Huang, Esteban Real, Kaiyuan Wang, Hieu Pham, Xuanyi Dong, Thang Luong, Cho-Jui Hsieh, Yifeng Lu, et al. Symbolic discovery of optimization algorithms. *Advances in neural information processing systems*, 36:49205–49233, 2023.

Ronan Collobert. Large scale machine learning. 2004.

Sanjoy Dasgupta. The hardness of k-means clustering. 2008.

Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pp. 248–255. IEEE, 2009.

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

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

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

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

Nicholas J Higham. *Functions of matrices: theory and computation*. SIAM, 2008.

Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. *Advances in neural information processing systems*, 33:6840–6851, 2020.

Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.

Shuoran Jiang, Qingcai Chen, Youcheng Pan, Yang Xiang, Yukang Lin, Xiangping Wu, Chuanyi Liu, and Xiaobao Song. Zo-adamu optimizer: Adapting perturbation by the momentum and uncertainty in zeroth-order optimization. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pp. 18363–18371, 2024.

Keller Jordan. Muon: An optimizer for hidden layers in neural networks. https://kellerjordan.github.io/posts/muon/, 2024. Blog post.

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

Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, pp. 1–6, 2015.

Edmund Landau. *Handbuch der Lehre von der Verteilung der Primazahlen*, volume 1. BG Teubner, 1909.

Bingrui Li, Jianfei Chen, and Jun Zhu. Memory efficient optimizers with 4-bit states. *Advances in Neural Information Processing Systems*, 36:15136–15171, 2023.

Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. Pytorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704*, 2020.

Wanchao Liang, Tianyu Liu, Less Wright, Will Constable, Andrew Gu, Chien-Chin Huang, Iris Zhang, Wei Feng, Howard Huang, Junjie Wang, Sanket Purandare, Gokul Nadathur, and Stratos Idreos. Torchtitan: One-stop pytorch native solution for production ready LLM pretraining. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=SFN6Wm7YBI.

Hong Liu, Zhiyuan Li, David Hall, Percy Liang, and Tengyu Ma. Sophia: A scalable stochastic second-order optimizer for language model pre-training. In *International Conference on Learning Representations*, volume 2024, pp. 1621–1650, 2024.

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

Yong Liu, Zirui Zhu, Chaoyu Gong, Minhao Cheng, Cho-Jui Hsieh, and Yang You. Sparse mezo: Less parameters for better performance in zeroth-order llm fine-tuning. *Advances in Neural Information Processing Systems*, 38:21139–21169, 2026.

Stuart Lloyd. Least squares quantization in pcm. *IEEE transactions on information theory*, 28(2): 129–137, 1982.

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

Meena Mahajan, Prajakta Nimbhorkar, and Kasturi Varadarajan. The planar k-means problem is np-hard. *Theoretical computer science*, 442:13–21, 2012.

Sadhika Malladi, Tianyu Gao, Eshaan Nichani, Alex Damian, Jason D Lee, Danqi Chen, and Sanjeev Arora. Fine-tuning language models with just forward passes. *Advances in Neural Information Processing Systems*, 36:53038–53075, 2023.

Joel Max. Quantizing for minimum distortion. *IRE Transactions on Information Theory*, 6(1):7–12, 1960.

Guilherme Penedo, Hynek Kydlíček, Anton Lozhkov, Margaret Mitchell, Colin Raffel, Leandro Von Werra, Thomas Wolf, et al. The fineweb datasets: Decanting the web for the finest text data at scale. *Advances in Neural Information Processing Systems*, 37:30811–30849, 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.

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *SC20: international conference for high performance computing, networking, storage and analysis*, pp. 1–16. IEEE, 2020.

Sashank J Reddi, Satyen Kale, and Sanjiv Kumar. On the convergence of adam and beyond. *arXiv preprint arXiv:1904.09237*, 2019.

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

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

James C Spall. Multivariate stochastic approximation using a simultaneous perturbation gradient approximation. *IEEE transactions on automatic control*, 37(3):332–341, 1992.

Hanlin Tang, Shaoduo Gan, Ammar Ahmad Awan, Samyam Rajbhandari, Conglong Li, Xiangru Lian, Ji Liu, Ce Zhang, and Yuxiong He. 1-bit adam: Communication efficient large-scale training with adam's convergence speed. In *International Conference on Machine Learning*, pp. 10118–10129. PMLR, 2021.

Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5—RMSProp: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural Networks for Machine Learning, 2012.

Haizhou Wang and Mingzhou Song. Ckmeans. 1d. dp: optimal k-means clustering in one dimension by dynamic programming. *The R journal*, 3(2):29, 2011.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Yushun Zhang, Congliang Chen, Ziniu Li, Tian Ding, Chenwei Wu, Diederik Durk Kingma, Yinyu Ye, Zhi-Quan Luo, and Ruoyu Sun. Adam-mini: Use fewer learning rates to gain more. In *International Conference on Learning Representations*, volume 2025, pp. 28033–28063, 2025.

Jiawei Zhao, Zhenyu Zhang, Beidi Chen, Zhangyang Wang, Anima Anandkumar, and Yuandong Tian. GaLore: Memory-efficient LLM training by gradient low-rank projection. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 61121–61143. PMLR, 2024. URL https://proceedings.mlr.press/v235/zhao24s.html.

Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, et al. Pytorch fsdp: experiences on scaling fully sharded data parallel. *arXiv preprint arXiv:2304.11277*, 2023.

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, 2024. URL https://arxiv.org/abs/2412.05270.

CONTENTS

1 Introduction **1**

2 Preliminaries and Related Work **3**

3 Theoretical Analysis **4**

4 Algorithm **5**

4.1 Automatic Partitioning 5

4.2 Exact Quantization of Momentum Using Histogram-Based Dynamic Programming 7

5 Experiments **7**

5.1 Peak Memory Footprint 7

5.2 Performance 8

5.3 Throughput and Distributability 8

6 Ablation Study **9**

7 Conclusion **9**

A Proof of the Main Theoretical Result **15**

B Empirical Evidence That High Hessian Implies Close Squared Gradients **19**

C Extended Ablation Study **20**

C.1 Exact Dynamic Programming Quantization 20

C.2 Stable Partitioning 21

C.3 Correct Partitioning 21

C.4 Dynamic Tree Quantization 21

D Exact DP Quantization Codebook Learning: Details **23**

E Pretraining vs. Finetuning **23**

F Distributability of Optimizers **24**

G Extended Related Work **25**

H Additional Experiments **26**

I Extensions to Gefen **27**

J Gefen-Muon **27**

K The Need for Moments **28**

L Limitations **28**

M Hyperparameters **29**

A Proof of the Main Theoretical Result

Theorem A.1 (Large Hessian entries contract squared-gradient ratios)**.** *Consider the two-layer MLP $z = W_1 x + b_1$, $y = W_2\sigma(z) + b_2$, where $y \in \mathbb{R}^{d_{\text{out}}}$, with MSE loss $L = d_{\text{out}}^{-1}\sum_{j=1}^{d_{\text{out}}}(y_j - t_j)^2$. Let $e_j := y_j - t_j$. Define:*

$$A_k := \sum_{i=1}^{d_{\text{out}}} e_i W_2[i,k], \quad U_{k,l} := |x_l \sigma'(z_k)|, \quad C_{k,k'} := \left|\frac{A_k}{A_{k'}}\right|, \quad D_{k,k'} := \left|\sum_i W_2[i,k]W_2[i,k']\right|$$

Assume:

$$D_{k,k'} \le \alpha, \quad U_{k,l} \le \beta, \quad |\log C_{k,k'}| \le \gamma, \quad \forall k,l,k',l' \text{ s.t. } k \ne k',\ A_k, A_{k'} \ne 0,\ H_{(k,l),(k',l')} \ne 0$$

For any such pair of entries $W_1[k,l]$ and $W_1[k',l']$, let $g_{k,l} := \partial L/\partial W_1[k,l]$ and define the ratio between their squared gradients and their mixed Hessian entry as

$$R := \frac{g_{k,l}^2}{g_{k',l'}^2}, \qquad H_{(k,l),(k',l')} := \frac{\partial^2 L}{\partial W_1[k,l]\,\partial W_1[k',l']}$$

Then, for $A := 2\alpha\beta^2/d_{\text{out}}$,

$$e^{-2\gamma}\left(\frac{|H_{(k,l),(k',l')}|}{A}\right)^2 \le R \le e^{2\gamma}\left(\frac{A}{|H_{(k,l),(k',l')}|}\right)^2$$

Proof. Consider a simple two-layer MLP with input vector x, and output vector y:

$$z = W_1 x + b_1 \qquad z \in \mathbb{R}^h \tag{7}$$

$$y = W_2\sigma(z) + b_2 \qquad y \in \mathbb{R}^{d_{\text{out}}} \tag{8}$$

with output dimension d_{out}, and MSE loss

$$L = \frac{1}{d_{\text{out}}}\sum_{j=1}^{d_{\text{out}}}(y_j - t_j)^2 = \frac{1}{d_{\text{out}}}\sum_{j=1}^{d_{\text{out}}} e_j^2 \tag{9}$$

Here $t \in \mathbb{R}^{d_{\text{out}}}$ is the target vector, and the error vector is $e_j := y_j - t_j$.

$$y_i = \sum_k W_2[i,k]\sigma\left(\underbrace{\sum_l W_1[k,l]x_l + b_1[k]}_{=z_k}\right) + b_2[i] \tag{10}$$

$$\frac{\partial L}{\partial y_i} = \frac{2}{d_{\text{out}}}e_i \qquad \frac{\partial y_i}{\partial W_1[k,l]} = W_2[i,k]\sigma'(z_k)x_l \tag{11}$$

Define:

$$A_k := \sum_{i=1}^{d_{\text{out}}} e_i W_2[i,k] \tag{12}$$

Hence the gradient entry w.r.t. $W_1[k,l]$ is

$$g_{k,l} := \frac{\partial L}{\partial W_1[k,l]} = \frac{2}{d_{\text{out}}}x_l\sigma'(z_k)\sum_{i=1}^{d_{\text{out}}} e_i W_2[i,k] = \frac{2}{d_{\text{out}}}x_l\sigma'(z_k)A_k \tag{13}$$

Both $\sigma'(z_k)$ and e_i depend on $W_1[k,l]$.

The Hessian w.r.t. $W_1[k,l]$ and $W_1[k',l']$ is:

$$H_{(k,l),(k',l')} := \frac{\partial^2 L}{\partial W_1[k,l]\,\partial W_1[k',l']} = \frac{\partial g_{k,l}}{\partial W_1[k',l']} = \frac{2}{d_{\text{out}}} x_l \left(\frac{\partial \sigma'(z_k)}{\partial W_1[k',l']} A_k \;+\; \sigma'(z_k) \frac{\partial A_k}{\partial W_1[k',l']} \right) \tag{14}$$

Now compute each derivative:

$$\frac{\partial \sigma'(z_k)}{\partial W_1[k',l']} = \sigma''(z_k) \frac{\partial z_k}{\partial W_1[k',l']} = \sigma''(z_k) \delta_{k,k'} x_{l'} \tag{15}$$

$$\frac{\partial A_k}{\partial W_1[k',l']} = \sum_{i=1}^{d_{\text{out}}} W_2[i,k] \frac{\partial e_i}{\partial W_1[k',l']} \tag{16}$$

$$\frac{\partial e_i}{\partial W_1[k',l']} = \frac{\partial y_i}{\partial W_1[k',l']} = W_2[i,k'] \sigma'(z_{k'}) x_{l'} \tag{17}$$

So

$$\frac{\partial A_k}{\partial W_1[k',l']} = x_{l'} \sigma'(z_{k'}) \sum_{i=1}^{d_{\text{out}}} W_2[i,k] W_2[i,k'] \tag{18}$$

Substitute into Equation (14) to get the Hessian:

$$H_{(k,l),(k',l')} = \frac{2}{d_{\text{out}}} x_l x_{l'} \Bigg[\delta_{k,k'}\, \sigma''(z_k) \sum_{i=1}^{d_{\text{out}}} e_i W_2[i,k] + \sigma'(z_k) \sigma'(z_{k'}) \sum_{i=1}^{d_{\text{out}}} W_2[i,k] W_2[i,k'] \Bigg] \tag{19}$$

Here $\delta_{k,k'}$ is the Kronecker delta. This is the second derivative between two *different* entries of W_1. For $k \neq k'$ (so the $\delta_{k,k'}$ term is zero), define the indexed quantities

$$U_{k,l} := |x_l \sigma'(z_k)|, \quad C_{k,k'} := \left| \frac{A_k}{A_{k'}} \right|, \quad D_{k,k'} := \left| \sum_i W_2[i,k] W_2[i,k'] \right|. \tag{20}$$

Then from Equations (13) and (19) we have:

$$R := \frac{g_{k,l}^2}{g_{k',l'}^2} = \left(\frac{U_{k,l}}{U_{k',l'}} \right)^2 C_{k,k'}^2 \tag{21}$$

$$|H_{(k,l),(k',l')}| = \frac{2}{d_{\text{out}}} U_{k,l} U_{k',l'} D_{k,k'} \tag{22}$$

Assume:

$$\begin{aligned} &D_{k,k'} \leq \alpha, \qquad U_{k,l} \leq \beta, \qquad |\log C_{k,k'}| \leq \gamma, \\ &\forall k,l,k',l' \text{ s.t. } k \neq k',\; A_k, A_{k'} \neq 0,\; H_{(k,l),(k',l')} \neq 0 \end{aligned} \tag{23}$$

Now fix any such pair. From Equation (22) and $D_{k,k'} \leq \alpha$, we get

$$U_{k,l} U_{k',l'} \geq \frac{|H_{(k,l),(k',l')}|\, d_{\text{out}}}{2\alpha} \tag{24}$$

Since the uniform bound applies to both index pairs, we also have $U_{k,l}, U_{k',l'} \leq \beta$, so:

$$\begin{aligned} U_{k,l}, U_{k',l'} &\in \left[\frac{|H_{(k,l),(k',l')}|\, d_{\text{out}}}{2\alpha\beta},\, \beta \right] \\ &\overset{\text{log arithmetic}}{\Longrightarrow} \quad \left| \log \frac{U_{k,l}}{U_{k',l'}} \right| \leq \log\left(\frac{2\alpha\beta^2}{|H_{(k,l),(k',l')}|\, d_{\text{out}}} \right) \;=:\; L_{k,l,k',l'} \end{aligned} \tag{25}$$

From Equations (21), (23) and (25) we get the following bounds on R:

$$e^{-2(L_{k,l,k',l'}+\gamma)} \leq R \leq e^{2(L_{k,l,k',l'}+\gamma)} \tag{26}$$

Equivalently, define

$$A := \frac{2\alpha\beta^2}{d_{\text{out}}} \tag{27}$$

which upper bounds the Hessian magnitude under the stated assumptions:

$$|H_{(k,l),(k',l')}| = \frac{2}{d_{\text{out}}} U_{k,l} U_{k',l'} D_{k,k'} \leq \frac{2}{d_{\text{out}}} \alpha\beta^2 = A. \tag{28}$$

Since $L_{k,l,k',l'} = \log\Big(\frac{2\alpha\beta^2}{|H_{(k,l),(k',l')}|\, d_{\text{out}}}\Big) = \log\Big(\frac{A}{|H_{(k,l),(k',l')}|}\Big)$:

$$\exp\left(-2\left[\log\left(\frac{A}{|H_{(k,l),(k',l')}|}\right)+\gamma\right]\right) \leq R \leq \exp\left(2\left[\log\left(\frac{A}{|H_{(k,l),(k',l')}|}\right)+\gamma\right]\right) \tag{29}$$

That is,

$$e^{-2\gamma}\left(\frac{|H_{(k,l),(k',l')}|}{A}\right)^2 \leq R \leq e^{2\gamma}\left(\frac{A}{|H_{(k,l),(k',l')}|}\right)^2 \tag{30}$$

Thus, as $|H_{(k,l),(k',l')}|$ increases within its feasible range $0 < |H_{(k,l),(k',l')}| \leq A$, it raises the lower bound and lowers the upper bound. By Hessian symmetry, swapping (k,l) and (k',l') leaves $|H_{(k,l),(k',l')}|$ unchanged and replaces R by $1/R$; therefore the admissible interval is reciprocal-symmetric around 1. Hence the admissible interval for R shrinks to the band $[e^{-2\gamma}, e^{2\gamma}]$ (and to 1 if γ is small). Empirically we see that the same phenomenon happens even when $k = k'$.

The Same Argument for W_2 The second-layer weights give an even simpler case. For $W_2[i,k]$, using Equation (10), since z does not depend on W_2,

$$g^{(2)}_{i,k} := \frac{\partial L}{\partial W_2[i,k]} = \frac{2}{d_{\text{out}}} e_i \sigma(z_k) \tag{31}$$

For two entries $W_2[i,k]$ and $W_2[i',k']$,

$$H^{(2)}_{(i,k),(i',k')} := \frac{\partial^2 L}{\partial W_2[i,k]\,\partial W_2[i',k']} = \frac{2}{d_{\text{out}}} \delta_{i,i'} \sigma(z_k)\sigma(z_{k'}) \tag{32}$$

Thus if $i \neq i'$, the mixed Hessian is zero. If $i = i'$ and we define

$$U^{(2)}_k := |\sigma(z_k)| \tag{33}$$

then

$$R^{(2)} := \frac{(g^{(2)}_{i,k})^2}{(g^{(2)}_{i,k'})^2} = \left(\frac{U^{(2)}_k}{U^{(2)}_{k'}}\right)^2 \qquad |H^{(2)}_{(i,k),(i,k')}| = \frac{2}{d_{\text{out}}} U^{(2)}_k U^{(2)}_{k'} \tag{34}$$

Assuming $U^{(2)}_k \leq \beta^{(2)}$ for all k, the same log-arithmetic gives

$$\left|\log \frac{U^{(2)}_k}{U^{(2)}_{k'}}\right| \leq \log\left(\frac{2(\beta^{(2)})^2}{|H^{(2)}_{(i,k),(i,k')}|\, d_{\text{out}}}\right) \tag{35}$$

Equivalently, with $A^{(2)} := 2(\beta^{(2)})^2/d_{\text{out}}$,

$$\left(\frac{|H^{(2)}_{(i,k),(i,k')}|}{A^{(2)}}\right)^2 \leq R^{(2)} \leq \left(\frac{A^{(2)}}{|H^{(2)}_{(i,k),(i,k')}|}\right)^2 \tag{36}$$

Therefore, for second-layer weights that share the same output coordinate, larger Hessian magnitude again forces the squared-gradient ratio toward 1.

Bias Terms The hidden bias $b_1[k]$ behaves like the first-layer weight $W_1[k,l]$ with the input factor x_l removed. From Equation (10), its gradient is

$$g_k^{(b_1)} := \frac{\partial L}{\partial b_1[k]} = \frac{2}{d_{\text{out}}}\sigma'(z_k)A_k \tag{37}$$

and the mixed Hessian between $b_1[k]$ and $b_1[k']$ is

$$H_{k,k'}^{(b_1)} := \frac{\partial^2 L}{\partial b_1[k]\,\partial b_1[k']} = \frac{2}{d_{\text{out}}}\left[\delta_{k,k'}\sigma''(z_k)A_k + \sigma'(z_k)\sigma'(z_{k'})\sum_i W_2[i,k]W_2[i,k']\right] \tag{38}$$

Thus, for $k \neq k'$, the $\delta_{k,k'}$ term disappears and the same argument applies with

$$U_k^{(b_1)} := |\sigma'(z_k)|, \qquad C_{k,k'} := \left|\frac{A_k}{A_{k'}}\right|, \qquad D_{k,k'} := \left|\sum_i W_2[i,k]W_2[i,k']\right| \tag{39}$$

In this case,

$$R^{(b_1)} := \frac{(g_k^{(b_1)})^2}{(g_{k'}^{(b_1)})^2} = \left(\frac{U_k^{(b_1)}}{U_{k'}^{(b_1)}}\right)^2 C_{k,k'}^2, \qquad |H_{k,k'}^{(b_1)}| = \frac{2}{d_{\text{out}}}U_k^{(b_1)}U_{k'}^{(b_1)}D_{k,k'} \tag{40}$$

so the same conditional bound as for W_1 follows.

For the output bias b_2, the Hessian is either zero or constant, so it is irrelevant to our claim:

$$g_i^{(b_2)} := \frac{\partial L}{\partial b_2[i]} = \frac{2}{d_{\text{out}}}e_i \tag{41}$$

and

$$H_{i,i'}^{(b_2)} := \frac{\partial^2 L}{\partial b_2[i]\,\partial b_2[i']} = \frac{2}{d_{\text{out}}}\delta_{i,i'} \tag{42}$$

□

Theorem A.2 (Within-layer Hessian affinity controls gradient magnitudes)**.** *Let $f_\theta(x) \in \mathbb{R}^m$ be a feedforward ReLU network, and let $L(\theta) = \ell(f_\theta(x))$, where $\ell : \mathbb{R}^m \to \mathbb{R}$ is twice differentiable and convex. Write $g = \nabla_\theta L(\theta)$, $H = \nabla_\theta^2 L(\theta)$, and $z = f_\theta(x)$, and assume that $\nabla\ell(z)$ is in the column span of $\nabla^2\ell(z)$. Further assume that θ is away from ReLU activation boundaries for this input, i.e., all preactivations are nonzero. Then for any two parameters i, j belonging to the same layer, we have $H_{ii} \geq 0, H_{jj} \geq 0$, and for the normalized Hessian affinity $\rho_{ij} = |H_{ij}|/\sqrt{H_{ii}H_{jj}}$ we have*

$$\left|\frac{g_i^2}{H_{ii}} - \frac{g_j^2}{H_{jj}}\right| \leq 2\lambda(z)^2\sqrt{2(1-\rho_{ij})},$$

where $\lambda(z)$ is the Newton decrement at z, namely $\lambda(z)^2 = \nabla\ell(z)^\top\big(\nabla^2\ell(z)\big)^\dagger\nabla\ell(z)$. Here, we use the convention that $g_i^2/H_{ii} = 0$ whenever $H_{ii} = 0$, and similarly $\rho_{ij} = 0$ whenever either $H_{ii} = 0$ or $H_{jj} = 0$.

Proof of Theorem 3.2. Let $J_a := \partial_a f_\theta(x) \in \mathbb{R}^m$, $q := \nabla\ell(z)$, and $S := \nabla^2\ell(z)$. For a composite loss $L(\theta) = \ell(f_\theta(x))$, the Hessian entries satisfy

$$H_{ab} = J_a^\top S J_b + \langle q, \partial_{ab}^2 f_\theta(x)\rangle.$$

Since the network uses ReLU activations and θ is away from activation boundaries, all ReLU masks are locally constant in a neighborhood of θ. Hence, locally, the network is obtained by composing affine maps and fixed diagonal activation masks. Holding all other layers fixed, $f_\theta(x)$ is affine in the parameters of any single affine layer. Therefore, for two parameters a, b belonging to the same affine layer, $\partial_{ab}^2 f_\theta(x) = 0$. In particular, for $a, b \in \{i, j\}$, we have $H_{ab} = J_a^\top S J_b$. Since $S \succeq 0$, this implies $H_{aa} \geq 0$ and $0 \leq \rho_{ij} \leq 1$. Note that if $H_{aa} = 0$, then $S^{1/2}J_a = 0$, and since $q \in \text{Range}(S)$, it follows that $g_a = \langle q, J_a\rangle = 0$. Thus, this is consistent with our convention that $g_a^2/H_{aa} = 0$ for zero-diagonal coordinates.

Define $u_i := S^{1/2}J_i/\sqrt{H_{ii}}$ and $u_j := S^{1/2}J_j/\sqrt{H_{jj}}$. Since $H_{ii} = J_i^\top S J_i$ and $H_{jj} = J_j^\top S J_j$, we have $\|u_i\| = \|u_j\| = 1$. Moreover,

$$|\langle u_i, u_j\rangle| = \frac{|J_i^\top S J_j|}{\sqrt{H_{ii}H_{jj}}} = \frac{|H_{ij}|}{\sqrt{H_{ii}H_{jj}}} = \rho_{ij}.$$

Let $r := (S^\dagger)^{1/2}q$. Since $q \in \mathrm{Range}(S)$, we have $q = S^{1/2}r$, and $\|r\| = \lambda(z)$. Therefore,

$$g_i = \partial_i L(\theta) = \langle q, J_i\rangle = \langle r, S^{1/2}J_i\rangle = \sqrt{H_{ii}}\langle r, u_i\rangle,$$

and similarly $g_j = \sqrt{H_{jj}}\langle r, u_j\rangle$.

Choose $s \in \{-1, 1\}$ such that $s\langle u_i, u_j\rangle = |\langle u_i, u_j\rangle|$. Then

$$\|u_i - s u_j\|^2 = 2 - 2|\langle u_i, u_j\rangle| = 2(1 - \rho_{ij}).$$

Using $\big||\xi| - |\eta|\big| \le |\xi - s\eta|$, valid for any $\xi, \eta \in \mathbb{R}$ and $s \in \{-1, 1\}$, we get

$$\begin{aligned}\left|\frac{|g_i|}{\sqrt{H_{ii}}} - \frac{|g_j|}{\sqrt{H_{jj}}}\right| &= \big||\langle r, u_i\rangle| - |\langle r, u_j\rangle|\big| \\ &\le |\langle r, u_i - s u_j\rangle| \\ &\le \|r\|\,\|u_i - s u_j\| \\ &= \lambda(z)\sqrt{2(1-\rho_{ij})}.\end{aligned}$$

Note that $|g_i|/\sqrt{H_{ii}} \le \lambda(z)$ and $|g_j|/\sqrt{H_{jj}} \le \lambda(z)$. Hence

$$\begin{aligned}\left|\frac{g_i^2}{H_{ii}} - \frac{g_j^2}{H_{jj}}\right| &\le \left(\frac{|g_i|}{\sqrt{H_{ii}}} + \frac{|g_j|}{\sqrt{H_{jj}}}\right)\left|\frac{|g_i|}{\sqrt{H_{ii}}} - \frac{|g_j|}{\sqrt{H_{jj}}}\right| \\ &\le 2\lambda(z)^2\sqrt{2(1-\rho_{ij})}.\end{aligned}$$

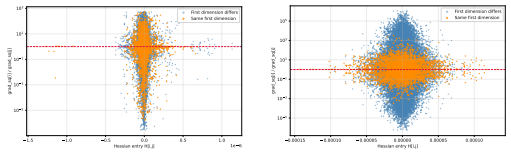
The proof is complete. □

A possible convergence analysis for our method may be developed by adapting existing analyses of Adam-style adaptive optimizers. The original convergence argument for Adam (Kingma & Ba, 2014) was later shown to be incomplete by Reddi et al. (2019), which introduced AMSGrad and established convergence guarantees under standard assumptions. More recently, convergence analyses have also been developed for quantized adaptive optimizers such as Adam4bit (Li et al., 2023). The theorem above shows that, under the assumption that the ratios between squared gradients within each shared block are bounded, the shared second-moment estimate remains within a bounded factor of the corresponding per-parameter estimates. This suggests that convergence analyses developed for adaptive optimizers may potentially be extended to the shared-second-moment setting. Establishing such guarantees rigorously is left for future work. Since convergence analyses of adaptive optimizers typically rely on assumptions that may not hold in modern deep-learning settings, we focus in this paper on empirical validation. The theoretical result is intended primarily as motivation for sharing second moments among Hessian-aligned parameters.

B Empirical Evidence That High Hessian Implies Close Squared Gradients

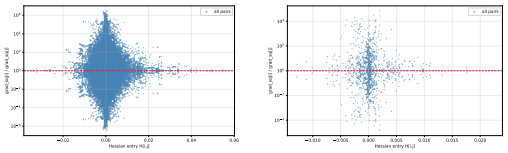
We trained small models and examined the relationship between the squared gradients of selected weight entries within several parameter tensors and the corresponding Hessian entries. The models are intentionally small, since computing the Hessian for large models is difficult, but they all tell the same empirical story. This matches the theory in Theorem 3.1: when the Hessian magnitude between two weights is high, the ratio between their squared gradients tends to be close to 1.

In Figure 6, as the Hessian entry moves away from zero in either direction, the ratio of the squared gradients, shown on a log scale, approaches one, marked by the horizontal dashed line. The same phenomenon appears both when the two weights share the same first dimension and when they do not. Another example, this time for a convolutional architecture, is shown in Figure 7. The same phenomenon appears there, even though the networks are very different.



(a) **(b)**

Figure 6: Squared-gradient ratio on a log scale vs. Hessian for GPT-toy at the beginning of training: (a) first-layer query matrix; (b) fourth-layer value matrix. As the Hessian entry moves away from zero, the squared-gradient ratio approaches one, as predicted by the theorem.



(a) **(b)**

Figure 7: Squared-gradient ratio on a log scale vs. Hessian for CNN on MNIST at the beginning of training: (a) `conv1.weight`; (b) `conv1.bias`. The same phenomenon appears in both the weight and bias parameters, as in the transformer model.

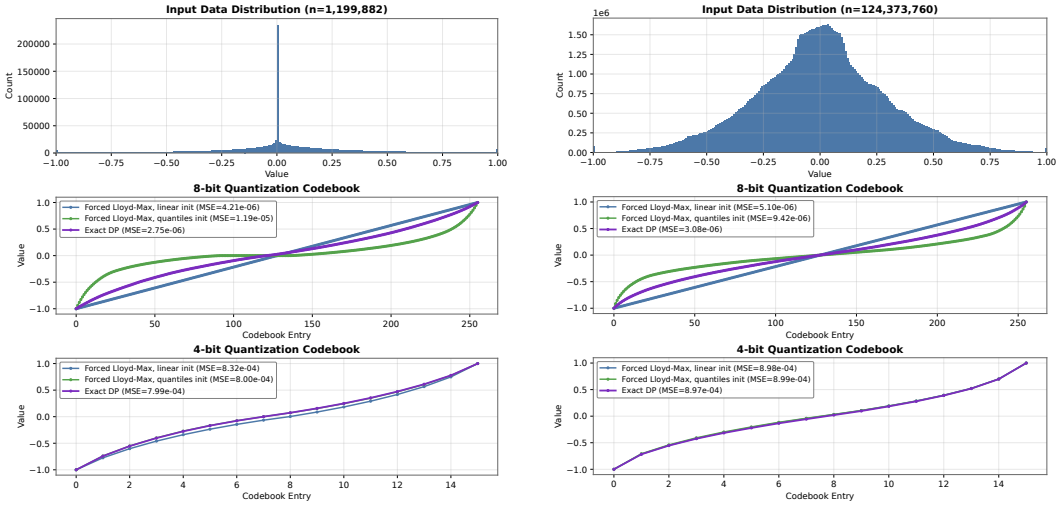
C Extended Ablation Study

In addition to the previous ablations, we present an extended ablation study below.

C.1 Exact Dynamic Programming Quantization

We compare three methods for quantizing gradients, which reduces the memory footprint of the first moment.

Lloyd-Max is an iterative method for learning a codebook for a given distribution. However, it has no theoretical guarantee of convergence to the optimal codebook, and in practice it can get stuck in local minima. We also find that it is sensitive to the initialization strategy: linear initialization (i.e., a uniformly spaced initial codebook) and quantile initialization (i.e., initializing the codebook at distribution quantiles) can lead to substantially different results. As shown in Figure 8, linear initialization sometimes outperforms quantile initialization, while in other cases the reverse is true.



(a) CNN on MNIST (b) GPT-2 on OpenWebText

Figure 8: Comparison of gradient quantization methods. The top panels show the gradient distributions used as input to the quantization algorithms, and the bottom panels show the learned 8-bit and 4-bit codebooks for each method. Lloyd-Max is sensitive to the initialization scheme. In contrast, the exact dynamic programming method requires no initialization and achieves the lowest quantization error while remaining fast.

We also compare against our exact histogram-based dynamic programming quantization algorithm. As expected, this method achieves the lowest quantization error because it is guaranteed to find

the optimal codebook for the given distribution and quantization level, up to the histogram binning resolution. For example, for GPT-2 with 8-bit quantization, Gefen achieves an MSE of 3.08×10^{-6}, compared with 5.10×10^{-6} for Lloyd-Max with linear (uniform) initialization and 9.42×10^{-6} with quantile initialization.

Finally, the optimal codebook found by our exact algorithm is nearly identical across models, datasets, and input distributions for the same quantization level. This is surprising: although the underlying distributions differ, they produce very similar optimal codebooks.

As an ablation, we also compared the final performance of our exact quantization method against Lloyd-Max with different initialization strategies. For a CNN on MNIST over three consecutive seeds, the exact method achieved a final validation loss of 0.0269 ± 0.0009, compared with 0.0283 ± 0.0020 for Lloyd-Max with quantile initialization and 0.0276 ± 0.0007 for Lloyd-Max with linear initialization.

C.2 Stable Partitioning

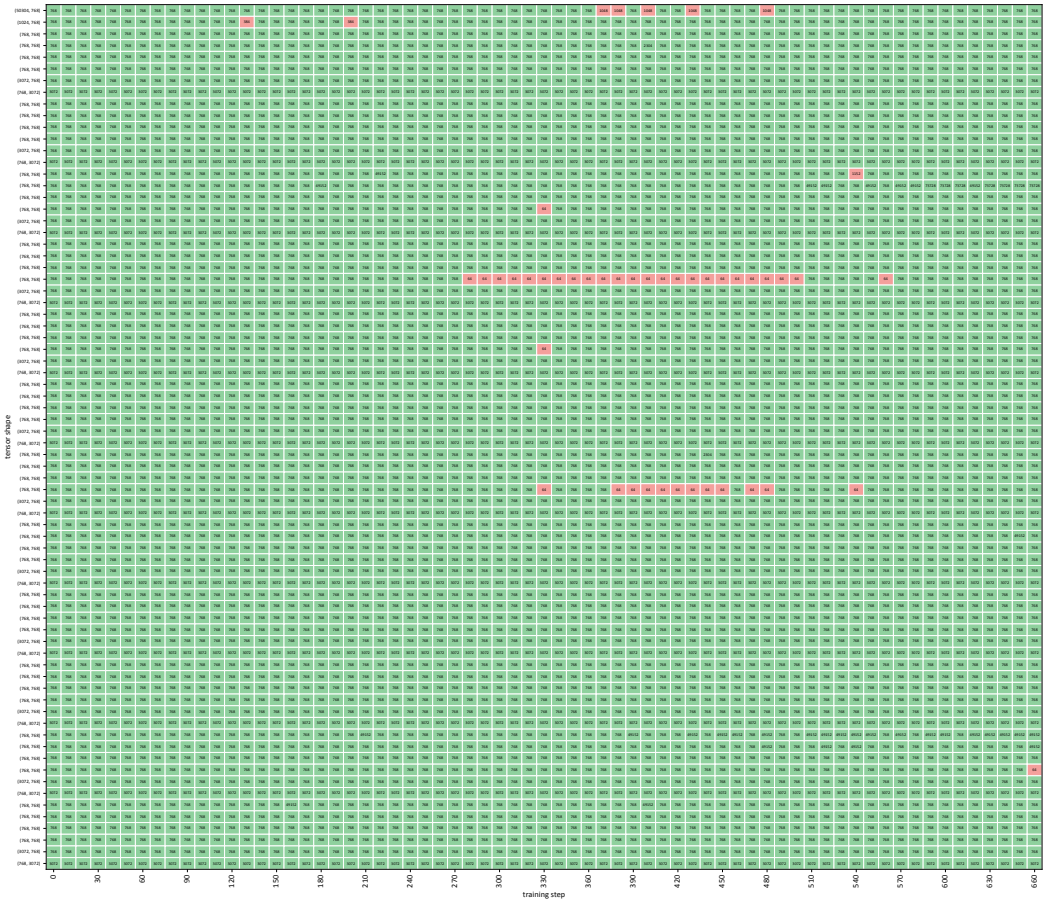
Because gradients are noisy, it is important to verify whether the partitioning inferred from the first-step gradients remains stable throughout training. In particular, we ask whether rerunning the partitioning algorithm at later steps recovers the same period for each tensor. To evaluate this, we train the GPT-2-125M and CNN models and rerun the partitioning algorithm periodically, comparing the period inferred for each tensor with its first-step period. Figures 9a and 9b report the results. Each entry is marked in green when the inferred period is identical to the first-step period or is an integer multiple of it. The latter case is also acceptable because the smaller first-step period yields a finer partition and cannot increase within-block heterogeneity. All other entries are marked in red. Overall, the vast majority of entries are green, indicating that the inferred periods remain relatively stable throughout training. Thus, the first-step partitioning appears to be a good proxy for the partitioning that would be obtained later, and repeated repartitioning is likely to provide only limited additional benefit. Despite the noise in the gradient estimates, both models exhibit largely consistent partitioning structures throughout training. Interestingly, the figures show that, although the automatic partitioning algorithm receives only a flattened tensor, it often recovers a period equal to one of the original tensor dimensions, or a combination of a few dimensions.

C.3 Correct Partitioning

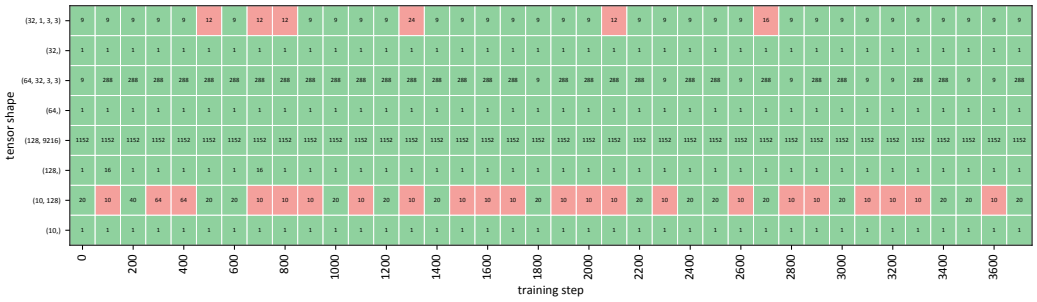
We ablate the importance of correctly aligning the partitions. Starting from the partition used by Gefen, we artificially perturb it in a few ways: we double the partition size, we shift the partition boundaries by half a period to deliberately create misalignment, and we randomize which weights in a given tensor belong to each partition. The last perturbation keeps both the number of partitions and their size exactly as chosen by Gefen, and replaces the consecutive coordinates of every partition with a random subset of the same size, drawn without replacement so that the partitions stay mutually exclusive and jointly cover all weights. The random assignment is drawn once and held fixed throughout training. As shown in Figure 10, all perturbations degrade performance compared with the original partition.

C.4 Dynamic Tree Quantization

We ablate Gefen's exact quantization of the first moment by replacing it with Dynamic Tree Quantization (DTQ), the scheme used by Adam8bit (Dettmers et al., 2021) (Section 2). Everything else is kept identical to Gefen: the first moment is quantized using the same partition as Gefen, with a single scale factor per partition, found from the second moment, exactly as in Gefen. The only change is that DTQ, rather than Gefen's exact dynamic programming quantization, is used to encode the normalized values within each partition. As shown in Figure 10, Gefen is almost always better than the DTQ variant. In addition, Gefen does not need the partition size as a hyperparameter, since it reuses the partition found automatically for the second moment.



(a) GPT-2.



(b) CNN-MNIST.

Figure 9: Partitioning stability across training steps.

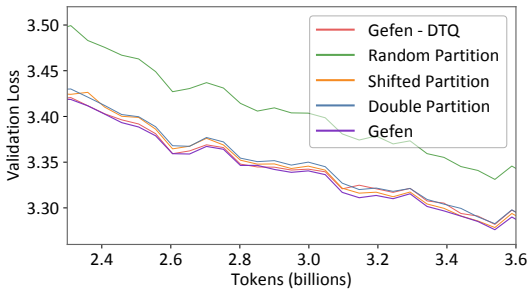


Figure 10: Partition perturbation and DTQ quantization ablations using GPT-2-125M trained on OpenWebText.

D Exact DP Quantization Codebook Learning: Details

The dynamic program Algorithm 3 exploits the fact that one-dimensional optimal quantization partitions the sorted input values into contiguous intervals. After histogramming the normalized gradients, we discard empty bins and sort the remaining bin centers m_i with counts c_i.

For any interval of bins $b_\ell, \ldots, b_r$, the weighted squared quantization error is:

$$C(b_\ell, b_r, k') = \sum_{i=b_\ell}^{b_r} c_i(m_i - \mu)^2 = S_2 - 2\mu S_1 + \mu^2 S_0$$

where $S_0 := \sum_{i=b_\ell}^{b_r} c_i$, $S_1 := \sum_{i=b_\ell}^{b_r} c_i m_i$, and $S_2 := \sum_{i=b_\ell}^{b_r} c_i m_i^2$ and μ is the value of the k'-th codebook entry. These are obtained in $O(1)$ time for any interval using prefix sums of c_i, $c_i m_i$, and $c_i m_i^2$.

For an unconstrained codebook entry, the optimal representative of the interval is its weighted mean $\mu = S_1/S_0$, and the interval cost is $S_2 - S_1^2/S_0$. For the forced endpoints, the first and last codebook entries are fixed to -1 and 1, respectively, so the cost for a fixed center a is $S_2 - 2aS_1 + a^2 S_0$.

Let $D_{b_r,k'}$ be the minimum squared error for representing the first b_r nonempty bins using k' codebook entries. The recurrence considers the left boundary b_ℓ of the final interval:

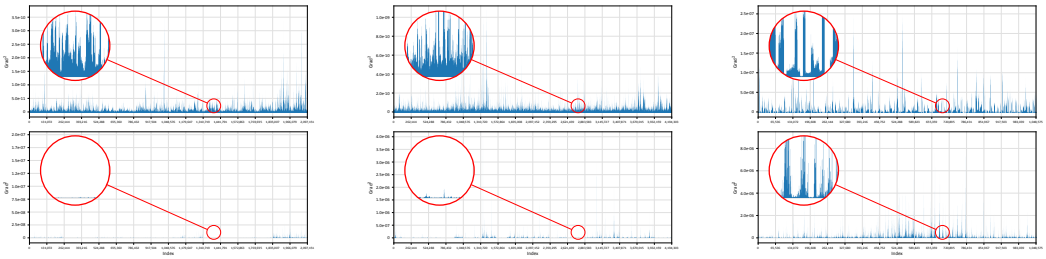
$$D_{b_r,k'} = \min_{b_\ell \le b_r} [D_{b_\ell - 1, k'-1} + C(b_\ell, b_r, k')]$$

This is correct by optimal substructure: in any optimal solution, the bins assigned to the last codebook entry form a suffix interval $b_\ell, \ldots, b_r$, and all earlier bins must themselves be optimally quantized by the previous $k' - 1$ entries. Conversely, combining any optimal prefix solution with the best cost for the final interval gives a valid k'-entry quantizer. Backtracking the minimizing split points therefore recovers the globally optimal codebook for the histogram. Since there are $O(bk)$ DP states and each state checks $O(b)$ split points, the running time is $O(b^2 k)$.

E Pretraining vs. Finetuning

Pretraining is substantially more compute intensive than finetuning, and is therefore the primary focus of this work. Nevertheless, it is useful to examine whether the gradient structure that motivates our second-moment compression also appears during finetuning. We find that the squared-gradient structure in finetuning differs markedly from that observed in pretraining. During pretraining, the Hessian of a parameter tensor, or the squared-gradient proxy used in our method, reveals architectural structure: the periodicity underlying our second-moment sharing is visually apparent and can be detected reliably. In contrast, this periodicity is much less pronounced during finetuning. One possible explanation is that pretraining starts from randomly initialized weights, whereas finetuning begins from parameters already located near low-loss regions of the optimization landscape, which may alter the local curvature and gradient statistics. Figure 11 illustrates this behavior for two architectures and three parameter tensors. The top row shows squared gradients at the beginning of

pretraining, while the bottom row shows squared gradients during finetuning. When zoomed in (Figure 12), the pretraining examples exhibit clear periodicity, in this case with period 2048, matching the hidden dimension of the corresponding networks. The finetuning examples, however, do not exhibit the same visible periodic structure. Since our second-moment sharing relies on periodicity across squared-gradient blocks, we focus this paper on the compute-intensive pretraining setting. By contrast, our first-moment quantization does not rely on grouping and applies equally well in both pretraining and finetuning.



(a) Qwen3-1.7B 27th-K (b) Llama-3.2-1B 7th-Q (c) Llama-3.2-1B 13th-V

Figure 11: Squared-gradient structure in pretraining and finetuning. Each panel compares the same parameter tensor at the beginning of pretraining (top) and during finetuning (bottom). The magnified regions show that the periodic structure used for second-moment sharing is clear in pretraining but substantially less pronounced in finetuning.

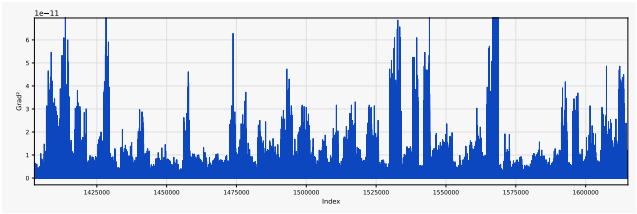


Figure 12: Zoomed-in squared-gradient structure for Qwen3-1.7B 27th layer, attention key parameter. Every 2048 parameters, corresponding to the hidden dimension of the network, exhibit similar squared-gradient magnitudes. This periodicity is used for second-moment sharing in Gefen.

F Distributability of Optimizers

Not every optimizer[1] is compatible with every distributed-training mechanism. In some cases, the limitation is inherent to the optimizer design, while in others it reflects implementation difficulty. As shown in Table 3, AdamW and Gefen are practical, general-purpose optimizers that support the common methods for distributed training.

We evaluate multiple common distributed-training settings. DDP refers to standard PyTorch `DistributedDataParallel`, where each rank holds a full model replica and gradients are all-reduced. FSDP refers to PyTorch FSDP with full sharding, applied after the model has been materialized on CUDA and before the optimizer is constructed. FSDP2 Sharded Init refers to the TorchTitan-style composable FSDP path, where the model is constructed on the meta device, sharded with `fully_shard`, materialized with `to_empty(device="cuda")`, initialized after sharding, and then passed to the optimizer. DeepSpeed ZeRO-3 (Rajbhandari et al., 2020) refers to ZeRO stage 3 applied to a normally materialized CUDA model. Tensor Parallelism partitions individual model tensors and their computations across ranks.

[1]For this comparison, we used `torch.optim.AdamW` for AdamW, `thu-ml/low-bit-optimizers` for Adam4bit, `bitsandbytes-foundation/bitsandbytes` for Adam8bit, `zyushun/Adam-mini` for Adam-mini, `jettify/pytorch-optimizer` for SM3 and Adafactor, and `torch.optim.Muon` for Muon.

Table 3: Compatibility of optimizers with common distributed-training configurations.

Optimizer	DDP	FSDP	FSDP2 Sharded Init	DeepSpeed ZeRO-3	Tensor Parallelism
AdamW	✓	✓	✓	✓	✓
Adam4bit	✓	✓	✗	✓	✗
Adam8bit	✓	✓	✗	✓	✗
Adam-mini	✓	✗	✓	✓	✓
SM3	✓	✓	✗	✓	✗
Adafactor	✓	✓	✗	✓	✗
Gefen	✓	✓	✓	✓	✓

G Extended Related Work

Second-Moment Compression. A complementary line of work reduces optimizer memory by compressing how second-moment statistics are stored. **Adafactor** (Shazeer & Stern, 2018) replaces a full $n \times m$ second-moment tensor with factored row and column accumulators (roughly $n + m$ state), then reconstructs elementwise scales from these marginals. **SM3** (Anil et al., 2019) keeps per-axis accumulators (for matrices, row-wise and column-wise maxima), forms each element's preconditioner from the elementwise minimum across axes, and scales updates by the inverse square root of that value (with ϵ in practice), yielding a memory footprint similar to Adafactor. **Lion** (Chen et al., 2023) is a sign-momentum optimizer discovered through symbolic program search, and its design rationale is less transparent. However, these methods rely on structural compression assumptions (factorizability in Adafactor, axis-wise summarization in SM3) that may not match every layer or training regime, and they often require learning-rate retuning relative to AdamW. These limitations make them less convenient as drop-in replacements, and they have already been found to underperform **Adam-mini**.

Other Optimizers. **Muon** (MomentUm Orthogonalized by Newton-Schulz) was recently introduced as an optimizer for hidden-layer matrix parameters (Jordan, 2024). Its update starts from momentum and then applies matrix orthogonalization using a few Newton–Schulz iterations, essentially equalizing singular values (Björck & Bowie, 1971; Higham, 2008). Muon uses about half the optimizer-state memory of AdamW because it tracks only the first-moment estimates (momentum). However, it is not a complete general-purpose drop-in optimizer: it is not intended for non-matrix tensors, usually handles input/output embeddings and classifier heads with a different optimizer (such as AdamW), requires manually selecting which parameters are handled by Muon and which by a general-purpose optimizer, and introduces additional optimizer-specific hyperparameters such as the number of Newton–Schulz iterations and its polynomial coefficients. In addition, it may have difficulty fine-tuning models pre-trained with AdamW (Liu et al., 2025), and its optimal learning rate is often different from that of AdamW. Muon is complementary to Gefen and can work in tandem with it, as shown in Appendix I. **Apollo** (Zhu et al., 2024) projects gradients, only for parameters selected by the user, into a low-dimensional subspace and then performs AdamW iteration, thereby reducing optimizer state memory. However, similar to Adam-mini, its implementation relies on manual if/else rules based on module names (e.g., applying their method only to "attn" and "mlp" in the parameter name), while requiring additional optimizer-specific hyperparameters from the user beyond AdamW, including the desired low-rank dimension, scaling factors, and learning-rate retuning, making it less convenient as a drop-in replacement. **Shampoo** (Gupta et al., 2018) is a structure-aware preconditioned optimizer that maintains matrix preconditioners for each tensor dimension, enabling richer second-order-style updates than diagonal adaptive methods. However, its goal is improved preconditioning rather than AdamW-style optimizer-state compression, and the additional matrix states and inverse-root computations make it less directly comparable to lightweight drop-in AdamW replacements. **Sophia** (Liu et al., 2024) is a stochastic second-order optimizer that divides an exponential moving average of gradients by an exponential moving average of lightweight diagonal Hessian estimates, and then applies elementwise clipping to the resulting update, for faster convergence. However, Sophia has the same optimizer-state memory footprint as AdamW, since it stores both first-moment and Hessian-preconditioner states, and it introduces additional optimizer-specific hyperparameters such as the Hessian update frequency and the clipping parameter; in contrast, we seek a general-purpose optimizer with lower memory footprint and no additional user-tuned hyperparameters beyond AdamW.

Gradient Communication Compression. Another complementary direction reduces distributed-training communication rather than optimizer-state memory. **1-bit Adam** (Tang et al., 2021) compresses Adam's gradient communication by transmitting one bit per coordinate, corresponding to the sign of the gradient, $\mathrm{sign}(g_t)$, together with error compensation, using a warmup stage to stabilize Adam's variance state before switching to compressed communication. However, each GPU still maintains Adam's full first- and second-moment states, so the per-GPU optimizer-state memory footprint remains roughly $2\times$ the parameter size. This line of work targets the network bottleneck in data-parallel training, whereas Gefen targets the memory footprint of optimizer states. Consequently, 1-bit Adam and Gefen are largely orthogonal: the communication compression of 1-bit Adam should compose naturally with the optimizer-state compression in Gefen, allowing the two techniques to work in tandem.

Zeroth-Order Optimizers. Zeroth-order (ZO) optimizers estimate search directions from function evaluations rather than backpropagated gradients, often using simultaneous perturbation or finite-difference estimators (Spall, 1992). This makes them attractive for memory-constrained fine-tuning because they avoid storing activations for the backward pass. For example, **MeZO** (Malladi et al., 2023) adapts ZO-SGD to fine-tune language models using only forward passes, and follow-up methods such as **ZO-AdaMU** (Jiang et al., 2024) and **Sparse MeZO** (Liu et al., 2026) improve the stability, convergence speed, or parameter efficiency of this approach. However, ZO methods optimize through noisy low-dimensional projections of the gradient and therefore trade memory savings for weaker or less robust optimization compared with standard backpropagation-based AdamW in the general-purpose training regimes considered here. We therefore view them as an important but distinct direction: they reduce memory by changing the optimization oracle itself, whereas Gefen preserves the AdamW training interface and targets AdamW-level performance with a smaller optimizer-state footprint.

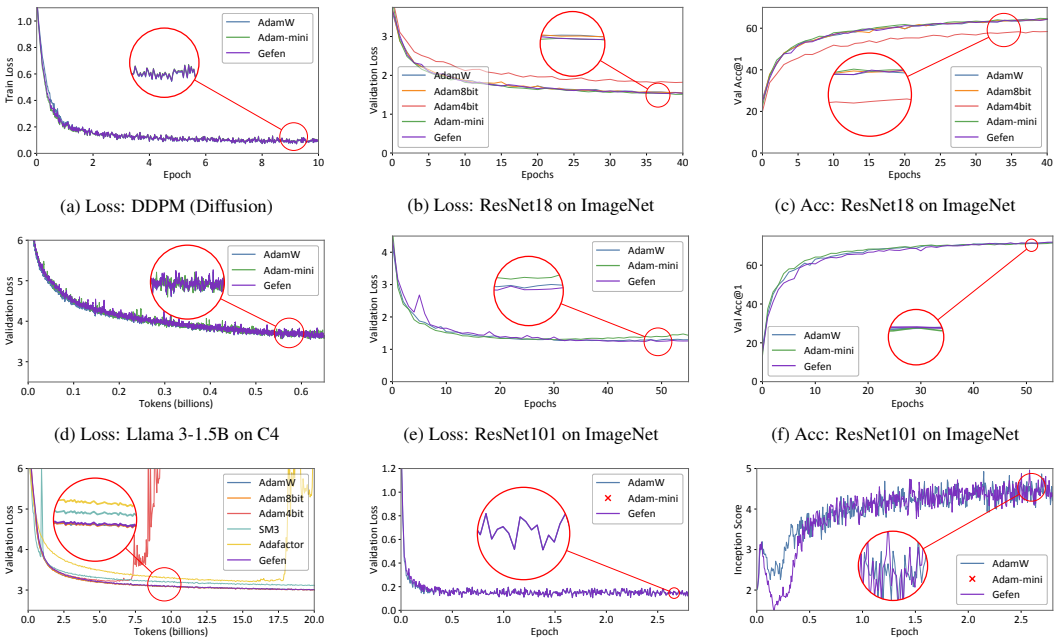
Distributed Training. Large-scale training commonly has several forms of parallelism. In this work, we focus on two common distributed-training schemes: DDP (Distributed Data Parallel) and FSDP (Fully Sharded Data Parallel). Data-parallel methods such as DDP (Li et al., 2020) replicate the model on each worker and synchronize gradients after the backward pass. Sharded data-parallel methods such as FSDP (Zhao et al., 2023) and DeepSpeed-ZeRO (Rajbhandari et al., 2020) reduce per-GPU memory by partitioning some or all of the parameters, gradients, and optimizer states across workers, at the cost of additional communication and implementation constraints. Model-parallel techniques, including tensor parallelism (Shoeybi et al., 2019) and pipeline parallelism (Huang et al., 2019), split the model computation itself across devices and are often combined with data parallelism for very large models. These techniques address memory and throughput at the system level, whereas Gefen reduces the optimizer state that each training configuration must store or shard. Therefore, optimizer-state compression is complementary to distributed-training schemes: it can reduce the memory pressure within DDP, lower the amount of optimizer state handled by sharded methods, and, as shown in our experiments, enable larger per-GPU microbatches in both FSDP and DDP settings.

H Additional Experiments

In Figure 13, we present additional plots for the experiments reported in Figure 3. Gefen performs on par with AdamW across the settings. In the ResNet18 and GPT-2-125M experiments, 4-bit quantization negatively affects performance. Furthermore, it can be seen that not all existing methods perform equally well with AdamW's learning rate. In the GPT-2-125M experiment, SM3 and Adafactor perform worse than AdamW when using the same learning rate. Adam-mini cannot be evaluated in all experiments because it relies on hard-coded tensor names and therefore does not support every model.

In the diffusion task DiT-XL/2 on ImageNet, Gefen reduces the AdamW memory state from 5720MiB to 722MiB.

For GPT-2-125M, the optimizer step takes 11 ms with AdamW and 3 ms with Gefen; for Llama 3-1.5B, it takes 115 ms with AdamW and 29 ms with Gefen. This reduction in latency occurs because Gefen writes less optimizer-state data to memory.



(a) Loss: DDPM (Diffusion)
(b) Loss: ResNet18 on ImageNet
(c) Acc: ResNet18 on ImageNet
(d) Loss: Llama 3-1.5B on C4
(e) Loss: ResNet101 on ImageNet
(f) Acc: ResNet101 on ImageNet
(g) Loss: GPT-2-125M on OpenWebText
(h) Loss: DiT-XL/2 on ImageNet
(i) Inception Score: DiT-XL/2 on ImageNet

Figure 13: Training curves of DDPM on Bollywood-Celebs, ResNet18 and ResNet101 on ImageNet, GPT-2-125M on OpenWebText, Llama 3-1.5B on C4, and DiT-XL/2 on ImageNet. Gefen performs on par with AdamW across the settings.

I Extensions to Gefen

Gefen modifies the AdamW state representation, and is therefore complementary to several methods that reduce memory along different axes. We list several natural extensions below.

1. **Combining with Muon.** Muon, discussed in Appendix G, stores momentum for matrix parameters and then applies Newton–Schulz iterations to obtain a pseudo-orthogonalized update. Since Muon does not maintain an AdamW-style second moment accumulator, a natural combination with Gefen would focus on quantizing Muon's momentum buffer, while leaving the pseudo-orthogonalization step unchanged. A concrete implementation and evaluation of this combination is provided in Appendix J.

2. **Combining with GaLore.** GaLore (Zhao et al., 2024) is a memory-efficient training method for LLMs. Given a gradient matrix G_t, GaLore projects it into a compact gradient R_t using low-rank projection matrices, applies an optimizer in this compact space, and then projects the resulting update back to the original parameter space. Since this projection is orthogonal to how optimizer statistics are represented, one could combine Gefen with GaLore when GaLore is used with AdamW-style optimizer states, by compressing the first and second moment estimates maintained inside the projected optimization step.

3. **Combining with Sophia.** Sophia, discussed in Appendix G, replaces AdamW's second moment estimate with a lightweight diagonal Hessian estimate and a clipped update. A Gefen variant could potentially quantize Sophia's gradient momentum and store its diagonal Hessian estimator compactly across blocks, while keeping Sophia's Hessian-based preconditioner and clipping rule unchanged.

J Gefen-Muon

To demonstrate that Gefen can be combined with other optimization methods, we provide a PyTorch implementation with CUDA kernels for Gefen-Muon.

The idea is straightforward: take Gefen and apply the orthogonalization step from Muon. This yields a one-line code-change drop-in replacement for Muon, while reducing Muon's persistent memory state by $4\times$ and decreasing peak memory by approximately 38% compared with Muon as the model gets larger (Table 4). For the orthogonalization step, we use the official PyTorch implementation of Muon.

Table 4: Memory footprint normalized by parameter memory of Gefen-Muon vs Muon.

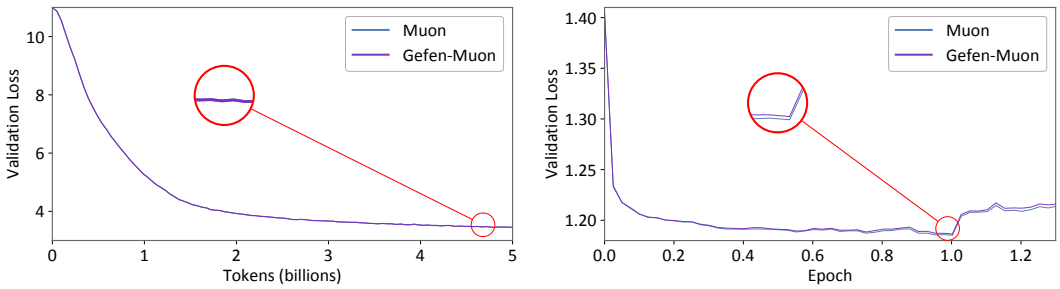
Optimizer	CNN (1.2M)		GPT-2 (125M)	
	Persistent mem	Peak mem	Persistent mem	**Peak mem**
Muon	x1.00	x3.12	x1.01	x1.63
Gefen-Muon	x0.26	x3.21	**x0.25**	**x1.01**

For simplicity, and to demonstrate that Gefen-Muon is a drop-in replacement, we use the same learning rate for Muon and Gefen-Muon. We also apply the same optimizer to all tensor types in the network by flattening tensors with more than two dimensions and expanding one-dimensional tensors into two dimensions.

Table 5: CNN on MNIST.

Optimizer	Validation Loss ↓
Muon	$0.0354_{\pm 0.0005}$
Gefen-Muon	$0.0356_{\pm 0.0005}$

As shown in Figure 14, the curves of Gefen-Muon and Muon are nearly identical in both pretraining and finetuning, suggesting that Gefen-Muon preserves Muon's optimization behavior while substantially reducing memory usage. The left panel shows GPT-2-125M pretraining on OpenWebText, and the right panel shows Qwen3-1.7B finetuning on Alpaca. We also compared the methods using CNN on MNIST as shown in Table 5. Full experimental settings are provided in Appendix M.



(a) Pretraining GPT-2-125M on OpenWebText. (b) Finetuning Qwen3-1.7B on Alpaca.

Figure 14: Muon and Gefen-Muon exhibit nearly identical validation-loss curves in both pretraining and finetuning.

K The Need for Moments

To demonstrate the necessity of the first- and second-moment estimates, we compare the performance of AdamW with that of AdamW without moving averages when training a CNN on MNIST. Specifically, we set both β_1 and β_2 to zero, effectively reducing AdamW to SignSGD. To ensure a fair comparison, we sweep over multiple learning rates and report its best-performing configuration. The results are shown in Figure 15. Even at its optimal learning rate, SignSGD cannot match the performance of AdamW, demonstrating the importance of maintaining moment estimates.

L Limitations

The partition is inferred from gradients at the beginning of training and is then held fixed. Although we find this partition to be stable in the settings we evaluate, distribution shifts or substantially different training regimes could change the relevant structure later in training.

Our empirical evaluation focuses primarily on pretraining models. In particular, the periodic squared-gradient structure used for second-moment sharing is clearly visible in the pretraining settings we study, but is less apparent in finetuning (Appendix E). Fully characterizing how the periodic gradient structure evolves in the transition from pretraining to finetuning, including the settings and

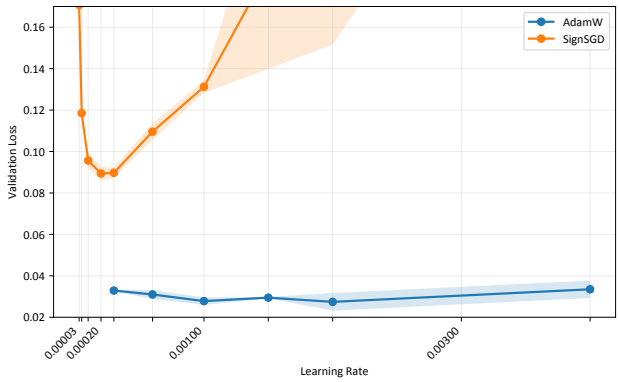


Figure 15: Comparison of AdamW and SignSGD when training a CNN on MNIST. SignSGD is obtained by setting $\beta_1 = \beta_2 = 0$, and its learning rate is selected by sweeping over multiple values. Lines show the mean across three consecutive random seeds, while shaded bands represent the mean $\pm$ one standard deviation.

stages of training in which this transition occurs, remains an interesting open question. The first-moment quantization component does not depend on this periodic structure and remains applicable when second-moment sharing is not.

M HYPERPARAMETERS

Llama Pre-training. For Figure 3 we pre-trained Llama 3-1.5B using the TorchTitan codebase (Liang et al., 2025) on the C4 dataset (Raffel et al., 2020). We use weight decay coefficient $\lambda = 0.0$, $\epsilon = 10^{-8}$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, and learning rate $= 10^{-4}$. For the learning-rate schedule, we use warmup for 1% of the total training steps, followed by linear decay. This matches the default scheduler used in the TorchTitan codebase. For Llama 3-1.5B, we use sequence length 2048 and global batch size 256, with per-GPU batch size 2 and 64 gradient accumulation steps on 2 GeForce RTX 3090 GPUs. For Llama 8B, we use global batch size 128 and microbatch size 1 on 4 NVIDIA H100 GPUs with 80 GiB VRAM. In the same Llama 8B configuration, Gefen also fits on 4 NVIDIA RTX A6000 GPUs with 48 GiB VRAM, whereas AdamW runs out of memory.

GPT-2 training on OpenWebText. We use the nanoGPT codebase[2] to train GPT-2-125M on OpenWebText (Figure 3). We use the recommended hyperparameters: `seq_len = 1024`, batch size $= 480$, weight decay coefficient $\lambda = 0.1$, $\epsilon = 10^{-8}$, $\beta_1 = 0.9$, and $\beta_2 = 0.95$. We use a cosine-decay learning-rate schedule with 2000 warmup iterations. For GPT-2-small, we use the peak learning rate recommended by Liu et al. (2024), which is reported to be optimal based on grid search. The chosen peak learning rate is 6×10^{-4}. The minimum learning rate is 3×10^{-5}. For SM3, this learning rate caused divergence, so we switched to a more stable learning rate of 0.225. For Adafactor and SM3, we added momentum with the same value for a fair comparison. In general, SM3 and Adafactor use different learning rates than AdamW, which makes them less suitable as drop-in replacements, but we report them for completeness. Better-tuned learning rates for SM3 and Adafactor may bring these optimizers closer to AdamW.

ResNet. For Figure 3 we use the official PyTorch implementation codebase[3] to train ResNet18 (He et al., 2016) on ImageNet (Deng et al., 2009). We use a cosine-decay learning-rate schedule, 90 epochs, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$, and weight decay 10^{-4}. For ResNet18, we use batch size $= 256$ and peak learning rate $= 0.005$.

[2] nanoGPT repository

[3] PyTorch torchvision ResNet implementation

CNN for MNIST. We use the official PyTorch implementation codebase[4] to train a CNN on MNIST on an RTX PRO 6000 Blackwell Max-Q GPU (Tables 1 and 2). The network consists of two convolutional layers with ReLU activations, followed by max pooling and dropout, and then two fully connected layers ending in a 10-class digit classifier. We use batch size $= 64$, test batch size $= 1000$, and train for 4 epochs. For all optimizers, we use $\beta_1 = 0.9$, $\beta_2 = 0.999$, and learning rate $= 0.001$, except for SM3 and Adafactor, for which we report the best result from learning rates in $\{0.001, 0.0006\}$. We use a `StepLR` scheduler with step size 1 and $\gamma = 0.7$, decaying the learning rate after each epoch. Standard deviations are computed over three consecutive seeds. For Muon, which only supports two-dimensional parameter tensors, we presented each model parameter to the optimizer as a 2D view: convolutional kernels and other higher-dimensional tensors were flattened across all dimensions except the output dimension, while one-dimensional tensors were expanded with a singleton dimension. It is possible that other learning rates would produce stronger results at 4 epochs for some methods, but our goal is to test AdamW compatibility; exhaustively tuning all learning rates for all methods is outside the scope of this paper.

DDPM. For Figure 3 we train an unconditional DDPM (Ho et al., 2020) on the Bollywood-Celebs dataset using the Hugging Face Diffusers codebase[5], specifically the `UNet2DModel`. The image size is 64, and the training objective is to predict the added noise with mean-squared error. We use base channel width 64, two layers per block, and U-Net channel multipliers $(1, 2, 4, 8)$, corresponding to block widths $(64, 128, 256, 512)$. The downsampling blocks are two standard convolutional blocks followed by two attention blocks, and the upsampling path uses the corresponding attention and convolutional blocks in reverse order. We use a diffusion process with 1000 training timesteps and the squared-cosine beta schedule. We train for 10 epochs with batch size 128, learning rate 5×10^{-5}, cosine learning-rate decay, weight decay 0, $\epsilon = 10^{-8}$, $\beta_1 = 0.9$, and $\beta_2 = 0.999$.

DiT-XL/2 on ImageNet. For the DiT-XL/2 experiment in Figure 3 and Appendix H, we train a class-conditional DiT-XL/2 on ImageNet at 256×256 resolution using the Hugging Face Diffusers implementation. Images are center-cropped, randomly flipped, and encoded into 32×32 latents by the EMA variant of the Stable Diffusion VAE. The transformer uses patch size 2, 28 layers, 16 attention heads with head dimension 72, and AdaLN-Zero conditioning over the 1000 ImageNet classes. We train for 10 epochs with effective global batch size 16, learning rate 10^{-4}, weight decay 0, $\epsilon = 10^{-8}$, $\beta_1 = 0.9$, and $\beta_2 = 0.999$, using bfloat16 precision and no learning-rate schedule. The diffusion process uses 1000 timesteps with a linear beta schedule, and the model is trained to predict the added noise using mean-squared error.

Qwen3-4B pre-training. For Figure 3, we pre-train Qwen3-4B (Yang et al., 2025) on the FineWeb-Edu dataset (Penedo et al., 2024) using the Axolotl codebase (Axolotl, 2023). We train with sequence length 2048, global batch size 128, and learning rate 3×10^{-4}. We use a cosine learning-rate schedule with 10 warmup steps. Training uses bfloat16 precision and FlashAttention 2. We use a per-GPU microbatch size of 1 and 32 gradient accumulation steps on four NVIDIA H100 GPUs.

Qwen3-1.7B finetuning on Alpaca. For the finetuning experiment in Figure 14, we train Qwen3-1.7B-Base (Yang et al., 2025) on the Alpaca dataset using the Axolotl codebase (Axolotl, 2023). We train with sequence length 2048, global batch size of 128, and learning rate 5×10^{-5}. We use a constant learning-rate schedule with 5 warmup steps, bfloat16 precision, and FlashAttention 2.

Throughput. For FSDP in Figure 2 we used Llama 3-1.5B on C4 with the default sequence length 2048, global batch size 256, and two RTX 3090 GPUs. In this setting, AdamW and Adam-mini support a microbatch size of 1, whereas Gefen supports a microbatch size of 2. For DDP in Figure 2 we used GPT2-1B on OpenWebText with the default sequence length 1024, global batch size 120, and two RTX 3090 GPUs. In this setting, AdamW cannot fit a microbatch size of 1, Adam-mini supports a microbatch size of 1, and Gefen supports a microbatch size of 2.

[4] PyTorch MNIST example implementation

[5] Hugging Face Diffusers repository