Per-example gradients: a new frontier for understanding and improving optimizers

Anonymous authors
Paper under double-blind review

Abstract

Training algorithms in deep learning usually treat a mini-batch of samples as a single object; they average gradients over the mini-batch, and then process the average in various ways. Computing other statistics beyond the average may have been seen as prohibitively resource intensive in automatic differentiation (AD) frameworks. We show that this is not the case. Generally, gradient statistics can be implemented through a surgery of the AD graph, which, in some cases, incur almost no computational and memory overheads compared to the mini-batch gradient computation. Additionally, we show that in certain classes of models, including transformers, JAX's vectorization transformation offers a viable implementation for prototyping and experimentation. We then revise our understanding of two nonlinear operations in optimization through the lens of per-example gradient transformations. We first study signSGD and show that the optimal placement of the sign operation in the gradient processing chain is crucial to success and can be predicted with a simple signal-to-noise ratio argument. Next we study per-example variations of the Adam preconditioner, and show that optimization is best served when the preconditioner is dominated by the mean rather than the variance of the gradient distribution — in contrast to conventional wisdom. Overall we demonstrate that per-example gradient information enables new analyses and possibilities for algorithm design.

1 Introduction

The success of modern machine learning would not be possible without efficient methods of computing gradients of loss functions through neural networks, primarily through the invention of reverse-mode automatic differentiation (AD) (Linnainmaa, 1970). With the advent of accelerators like GPUs, practitioners are now able to approximate the expected gradients of a model's loss function on a dataset by averaging gradients on large batches of data. This has allowed model and data complexity to grow rapidly while maintaining tractability of training algorithms (Hoffmann et al., 2022).

A key feature of reverse-mode AD is that it never stores the gradients of individual elements of a batch. This increases the memory efficiency of the process, but makes it impossible to answer questions about the distribution of per-example gradients, or allow for optimizers to depend on higher order moments of the gradient distribution. This leaves a vast part of training algorithm design space inaccessible for researchers, particularly if they want to work at large scales.

Access to per-example gradient covariance statistics is useful for understanding deep learning; this information can be used to predict quantitative properties of loss trajectories at scale (Yin et al., 2018; Qiu et al., 2025). There is practical interest in per-example gradient transformations as well. There is some evidence that optimizers which have access to per-example gradients can have better stability and predictability than typical optimizers (Wang & Aitchison, 2024). The growing interest for distributed optimizers like DiLoCo (Douillard et al., 2023) calls for studying how optimizers may benefits from aggregating more than just the gradients (Zhang et al., 2023). This has led to attempts to efficiently compute per-example gradient norms (Kong & Munoz Medina, 2023; Bu et al., 2023), and to come up with ways to modify AD more generally to compute higher order moments of gradient distributions (Dangel et al., 2019).

Motivated by these examples, we study the technical challenges arising from computing generic gradient statistics. We show the following:

- Generic gradient statistics can be implemented with a *computational graph surgery*, and, in some cases, gradients' structure allows the approach to be as efficient as computing average gradients (Section 2.2).
- For sequence-based models, an implementation using just-in-time compiled JAX `vmap` vectorization transformation only incurs a moderate computational overhead, providing a path for prototyping (Section 2.3).

Through these technical findings *we do not claim that generic gradient statistics are inexpensive*, *we simply claim that they are not prohitively expensive.* We can then use gradient statistics to improve our understanding of two optimization algorithms:

- **SignSGD** (Section 3.1). We study a per-element version of SignSGD and find that the optimal positioning of the sign operation is as late as possible in the processing chain — a phenomenon which can be understood using a simple signal-to-noise ratio analysis.
- **Adam** (Section 3.2). We study Adam variants which operate on per-example statistics like the second moment. We provide, to the best of our knowledge, the first true implementation of these algorithms at scale, have the predicted scaling properties with batch size. We show that preconditioners which depend on the mean squared train faster and more stably than those which depend on the variance, in contrast to conventional wisdom.

Altogether, our results suggest that studying per-example gradients/gradient transformations is tractable even in modern settings and opens a new dimension for understanding and improving training algorithms.

2 Accessing mini-batch gradient statistics

2.1 Objective and challenges

Deep learning pipelines generally consist of optimizing the parameters θ of a model f as follows:

1. Fetch a mini-batch of samples $x_1, \ldots, x_B$ of size B:
   - `fetch_batch(train_data)` $= x_1, \ldots, x_B$
2. Compute mini-batch loss in an automatic differentiation (AD) framework.
   - `loss(params, batch)` $= \frac{1}{B}\sum_{i=1}^{B} f(\theta; x_i)$
3. Get the gradient of the mini-batch loss w.r.t. the parameters by gradient backpropagation.
   - `grad(loss)(params, batch)` $= \nabla \frac{1}{B}\sum_{i=1}^{B} f(\theta; x_i) = \frac{1}{B}\sum_{i=1}^{B} \nabla f(\theta; x_i)$
4. Feed the mini-batch gradient to an optimizer that returns directions along which parameters are updated.

As such, optimizers only have access to an estimate of the expectation of the gradients

$$\mathbb{E}[\nabla f(\theta; X)] \approx \frac{1}{B}\sum_{i=1}^{B} \nabla f(\theta; x_i), \tag{1}$$

where $x_1, \ldots, x_B$ are assumed to be i.i.d. samples of X. However, other statistics might be of interest for understanding existing algorithms, and to design new ones. We may want to approximate the expectation of an arbitrary function ϕ of the gradients in a minibatch as

$$\mathbb{E}[\phi(\nabla f(\theta; X))] \approx \frac{1}{B}\sum_{i=1}^{B} \phi(\nabla f(\theta; x_i)). \tag{2}$$

Specific examples include the elementwise square $\mathbb{E}[\nabla f(\theta; X)^2]$, the elementwise variance $\mathbb{E}[(\nabla f(\theta; X) - \mathbb{E}[\nabla f(\theta; X)])^2]$, and sign $\mathbb{E}[\text{sign}(\nabla f(\theta; X))]$.

If $\frac{1}{B}\sum_{i=1}^{B} \phi(\nabla f(\theta; x_i))$ is the gradient of some function `f`, it could be computed by automatic differentiation as `grad(f)(params, batch)`. However, this is generally not the case. With basic access to gradient oracle calls like `grad(loss)(params, sample)`, there are two simple ways to compute this statistic, both at the ends of a computation-memory trade-off.

1. **Memory intensive implementation.** Compute $\nabla f(\theta; x_1), \ldots, \nabla f(\theta; x_B)$ in parallel, apply ϕ to each in parallel, and then take the average. Assuming that the statistics we search are of lower or equal dimension than the gradients, i.e., $\dim(\phi(\nabla f(\theta; x_i))) \leq \dim(\nabla f(\theta; x_i))$, this requires BP memory storage, with P the dimension of the parameters, and can be prohibitive for some architectures and hardware.

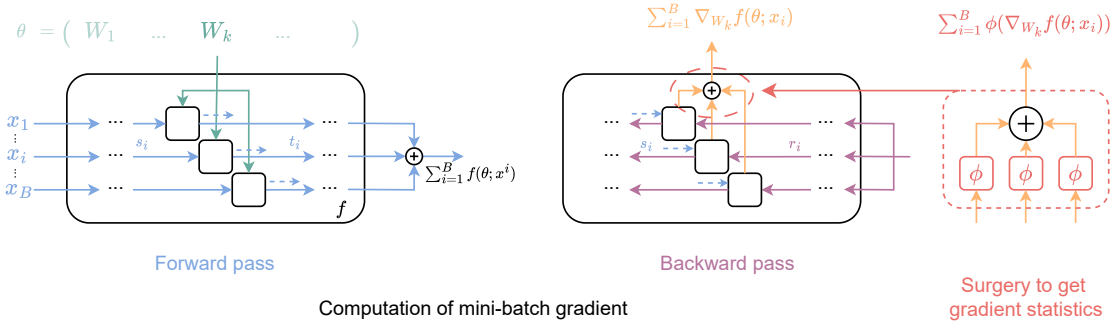

Figure 1: Computational graph of the mini-batch loss and the mini-batch gradient w.r.t. some intermediate weights. The forward pass has independent computational paths for each datapoint x_i and intermediate activation s_i, and the weights are essentially broadcasted to each computational path before the final loss merges them (left). In the backwards pass the residuals r_i move along the reversed computational paths and are similarly broadcast, and the merging of paths only happens at the end via sum reduction — the adjoint operation of the weight broadcasting. Computing gradient statistics of a function ϕ of the gradients can be done by injecting ϕ just before the final sum reduction.

2. **Computationally intensive implementation.** Accumulate the average $\bar{\phi}$ in a loop for $i = 1$ to B, computing the gradient, $\nabla f(\theta; x_i)$, applying $\phi(\nabla f(\theta; x_i))$, and updating $\bar{\phi} \leftarrow [(i-1)\bar{\phi} + \phi(\nabla f(\theta; x_i))]/i$. Assuming again $\dim(\phi(\nabla f(\theta; x_i))) \leq \dim(\nabla f(\theta; x_i))$, this prevents using more memory storage than the one needed to compute a gradient. However, it requires B more calls to a gradient oracle.

Efficiently implementing these non-linear averages requires a dive into the mechanics of AD, and the computational bottlenecks of deep learning architectures.

2.2 Gradient statistics by computational graph surgery

In the computation of the mini-batch gradient, most computations preserve per-example information, as can be seen in the generic computational graph (Figure 1). The averaging of the gradients over the batch is generally the last operation (illustrated in Figure 1, detailed in Appendix B.1). If we parse the computational graph of the minibatch gradient, we can "inject" the desired computation ϕ to the individual gradients before averaging. In the remainder of this section we describe an efficient way to perform this operation.

For certain operations ϕ can be injected with negligible overhead. Specifically, consider a dense layer parameterized by weights $W_k \in \mathbb{R}^{D \times D}$. In the computation of the loss $f(\theta; x_i)$, this layer acts on intermediate inputs s_i as $t_i^\top = s_i^\top W_k$. To compute the gradient of $f(\theta, x_i)$ w.r.t. W_k, reverse mode AD performs a backward pass through the transpose Jacobians that computes $r_i := \nabla_{t_i} f(\theta; x_i)$, the derivative of $f(\theta; x_i)$ w.r.t. the output of that layer. The gradient w.r.t. W_k is then computed by passing r_k through the adjoint operation of $W_k \mapsto s_i^\top W_k$. This gives $\nabla_{W_k} f(\theta; x_i) = s_i r_i^\top$ – a rank-one matrix – for a single example. For the mini-batch gradient we get

$$\nabla_{W_k} \sum_{i=1}^{B} f(\theta; x_i) = \sum_{i=1}^{B} s_i r_i^\top = SR, \tag{3}$$

where $S = (s_1, \dots, s_B)^\top \in \mathbb{R}^{B \times D}$ and $R = (r_1, \dots, r_B) \in \mathbb{R}^{D \times B}$, stack the intermediate inputs and intermediate derivatives computed in the independent computational paths illustrated in Figure 1. The required memory of this operation is $O(BD + D^2)$.

The rank-one nature of the individual gradients can be exploited by any "factorable" operation ϕ where $\phi(sr) = \phi(s)\phi(r)$. Examples of such functions include the sign function as well as any power $\phi(s) = s^\alpha$. Of particular interest is the elementwise square of the gradient which amounts to

$$\sum_{i=1}^{B} [\nabla_{W_k} f(\theta; x_i)]^2 = \sum_{i=1}^{B} (s_i r_i^\top)^2 = \sum_{i=1}^{B} (s_i)^2 (r_i)^{2\top} = S^2 R^2. \tag{4}$$

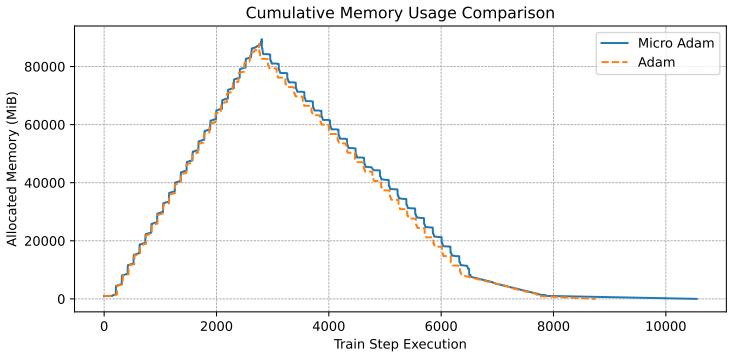

Figure 2: Memory footprint along program execution as reported by a code profiler of the train step of the usual ADAM algorithm and its per-example variant, MICROADAM (Section 3.2), for a 1.2B transformer in Nanodo (Liu et al., 2024). The peak memory corresponds to the accumulation of the memory during the forward pass of automatic differentiation necessary to compute gradients. We observe that the per-example variant may incur more operations (longer tail) that translate in longer execution time. But the peak memory is the same.

Apart from an elementwise square of the matrices S, R, this operation has the same memory and computational complexity as computing the sum of gradients. It can take advantage of accelerators, and it does not suffer from the $O(BD^2)$ memory cost of the naive memory intensive implementation from Section 2.1.

The overall approach amounts to a *computational graph surgery* and can be implemented in JAX (Bradbury et al., 2018); we provide code and benchmarking in Appendix B. For each operation of interest ϕ, the method modifies the basic linear primitives in JAX. We note that a related approach was taken on a different problem (fast computation of NTK) in Novak et al. (2022). The specific case of efficient computation of squared gradients has been pointed out previously by Dangel et al. (2019), but their approach required overriding derived classes (specifically, base layers) in PyTorch (Paszke et al., 2019). In contrast our approach modifies basic framework operations, allowing automatic compatibility with downstream modules.

2.3 OPPORTUNITIES IN SEQUENCE-LEVEL ARCHITECTURES

While the computational graph surgery approach is demonstrably efficient, it may not be needed for tractable prototyping and exploration in certain settings. In particular, more naive implementations are viable in sequence to sequence models like transformers.

For dense layers taking vector inputs, processing a batch of data of size B with a layer of width D contributes to the memory cost of the backwards pass in two ways: a memory cost of BD from activations, and a cost of D^2 from the parameters. Historically, the parameter cost D^2 was often limiting, which meant that storing the individual gradients at a cost BD^2 was prohibitively expensive.

However, for sequence-based layers trained on sequences represented by L vectors of size D, the cost of storing the activations of the batch alone is BDL, while the parameters take D^2 space. If $L \geq D$, then the cost of storing B individual gradients is less than the memory cost of the activations! This does not apply to the embedding and final head layers but it still suggests that gradient statistics may be computable in these models. So to test the hypothesis, we implemented the elementwise square of the gradient of a transformer model in JAX using the automatic vectorization tool `vmap` to implement the per-example gradients statistics. We found that computing these statistics did not increase the peak memory usage (which was reached by checkpointing activations, see Figure 2) and it only added a small compute overhead ($\approx 17\%$) on a model of size 1.2B parameters with TPU v5. Here JAX may also benefit highly from the underlying XLA's optimizations (Appendix A) that can best trade-off computational and memory costs.

Overall, our results suggest that for transformer architectures and some other layers detailed in Appendix B.4, the `vmap` approach offers a reasonably efficient way to access non-linear gradient statistics and prototype new

optimizers or diagnostic tools. The computational graph surgery can complete the prototyping to generalize these optimizers to other architectures while keeping a small computational overhead.

3 New perspectives using mini-batch gradient statistics

In the remainder of the paper, we ask: what does access to this additional information actually buy us? We use the rapid prototyping `vmap` approach to study properties of optimization algorithms and their per-example counterparts, using methods and measurements enabled by our approach.

Many optimizers in deep learning can be written as processing batch gradients through a sequence of transformations $T = T_p \circ \ldots \circ T_1$ to define an update direction along which parameters are updated,

$$\theta_{t+1} = \theta_t - \eta T(\text{avg_grad}(\theta_t)). \tag{5}$$

Our methods unlock the potential for transformations that come after the gradient computation but before the averaging. Formally, we can change a training algorithm T to a new algorithm T' as follows:

$$T = T_2 \circ T_1 \circ \text{avg} \circ \text{grad} \rightarrow T' = T_2 \circ \text{avg} \circ T_1 \circ \text{grad},$$

where avg is the average over the gradients, and T_1 and T_2 are gradient transformations.

We focused on 2 settings: SignSGD and Adam. We chose these settings for their relevance to practical optimization strategies, and since we know they can be implemented efficiently using the computational graph surgery approach. In order to provide insights relevant for modern machine learning, all experiments were performed on a 151M parameter decoder-only transformer language model, trained on the C4 dataset (Raffel et al., 2020) using the Nanodo codebase (Liu et al., 2024). Unless otherwise specified, we use a batch size of 64, use a cosine learning rate schedule and a weight decay. Full experimental details can be found in Appendix E.

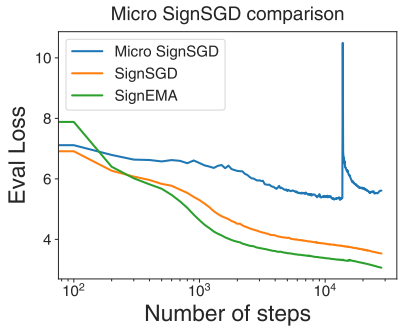

Figure 3: Learning curves for SignSGD variants at optimal learning rates, $\beta_1 = 0.9$. SignEMA has the best performance and MicroSignSGD has the worst performance. This suggests that the sign function needs to be applied as late as possible to prevent signal-to-noise ratio reduction for gradients of individual parameters.

3.1 Where should you place the sign in SignSGD?

The optimizer SignSGD (Bernstein et al., 2018) has been well studied in various contexts, both from a theoretical perspective (Compagnoni et al., 2024; Balles et al., 2020; Xiao et al., 2025) but also as a practical optimizer (Zhao et al., 2024). The most basic SignSGD update rule is given by instantiating the typical optimizer (5) with $T = \text{sign}$ the sign function applied elementwise. Theoretical work has shown that SignSGD is similar to RMSprop, effectively preconditioning by the square root of the diagonal of the Gauss Newton matrix (Xiao et al., 2025).

To make SignSGD a practical algorithm, practitioners add minibatching and momentum. A natural question is: what is the optimal order of operations? More concretely, we consider three operations: avg, which computes the empirical average of a set of minibatch gradients, EMA which takes an exponential moving average, and sign, applied to each parameter. The three algorithms we consider are:

$$\begin{aligned} \text{SignEMA} &= \text{sign} \circ \text{EMA} \circ \text{avg} \\ \text{SignSGD} &= \text{EMA} \circ \text{sign} \circ \text{avg} \\ \text{MicroSignSGD} &= \text{EMA} \circ \text{avg} \circ \text{sign} \end{aligned}$$

This covers all orders of the 3 operations, since avg and EMA commute due to their linearity.

MicroSignSGD is a new algorithm enabled by the per-example transforms. Here the sign operation is applied first, followed by the minibatch averaging, and then the momentum. SignEMA, a.k.a. signum, was found to be competitive with Adam in training large transformer models (Zhao et al., 2024).

After optimizing learning rates, we find that SignEMA trains best, followed by SignSGD (Figure 3). Both train much better than MicroSignSGD. In addition to training more slowly, the MicroSignSGD curves are noisier (including a late-time training spike). Our results suggest that applying the sign function as late as possible results in the fastest, most stable training.

We hypothesize that MicroSignSGD fails in part because any stronger preconditioning is outweighed by the amplification of noise in the per-example transformations. The sign function reduces SNR for distributions with low SNR, and increases SNR for distributions with high SNR (Appendix C.1). This theoretical analysis is consistent with our observations that sign should be applied after the maximal amount of averaging — corresponding to the largest variance reduction possible for the object being transformed.

3.2 Adam and per-example statistics

In this section, we analyze the statistics of the preconditioner in Adam on a per-example basis. The original Adam algorithm accumulates over time the squared average of gradients, denoted ν_{adam} below; we will contrast this measure with the average of squared gradients ν_{micro} (accessible via per-example methods):

$$\nu_{\text{adam}} = \left(\frac{1}{B}\sum_{i=1}^{B} g_i\right)^2, \quad \nu_{\text{micro}} = \frac{1}{B}\sum_{i=1}^{B} g_i^2, \quad \text{for } g_i = \nabla f(\theta; x_i). \tag{6}$$

Using ν_{adam} and ν_{micro} to construct new optimizers and measurements give us new insights on Adam.

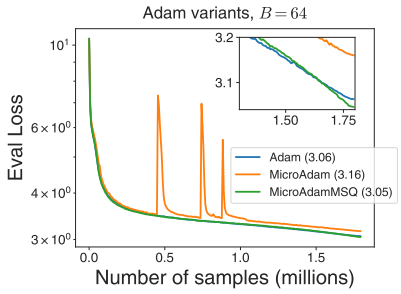

Figure 4: MicroAdam (orange) emphasizes variance information in preconditioner and generally trains less stably and more slowly than Adam (blue), while MicroAdamMSQ (green) emphasizes mean squared information and shows slight gains

3.2.1 Adam, MicroAdam, and the statistics of μ^2 and σ^2

If the gradients g_i are i.i.d., we can compute the expected values of ν_{adam} and ν_{micro} as follows:

$$\mathbb{E}[\nu_{\text{adam}}] = \mu^2 + \sigma^2/B, \qquad \mathbb{E}[\nu_{\text{micro}}] = \mu^2 + \sigma^2, \tag{7}$$

where μ and σ are the mean and standard deviation of the individual g_i.

This observation is the basis of the *square root learning rate rule*. If $\mu = 0$, then the preconditioner scales as $B^{-1/2}$, which suggests that scaling the learning rate as $\eta \propto B^{1/2}$ causes the loss trajectories of models at different batch sizes to become similar when plotted versus the total number of samples (compared to SGD's rule $\eta \propto B$). This equivalence becomes exact in certain limits (Xiao et al., 2025). This heuristic also applies to the scaling of the optimal learning rate with batch size, for small batch sizes (Shallue et al., 2019). We provide a simple and intuitive derivation of the rule in Appendix D.1.

Wang & Aitchison (2024) point out that if $\mu \neq 0$, Equation 7 may have a crossover effect at large batch sizes where ν_{adam} goes from being variance dominated to mean squared dominated. They proposed MicroAdam— an optimizer where the preconditioner ν_{adam} of Adam is replaced with the true second moment ν_{micro}. They provided numerical evidence that this approach could lead to more universal learning curves for training ResNet18 on CIFAR10, using device parallelism to approximate ν_{micro} with a per-device batch-size of 25.

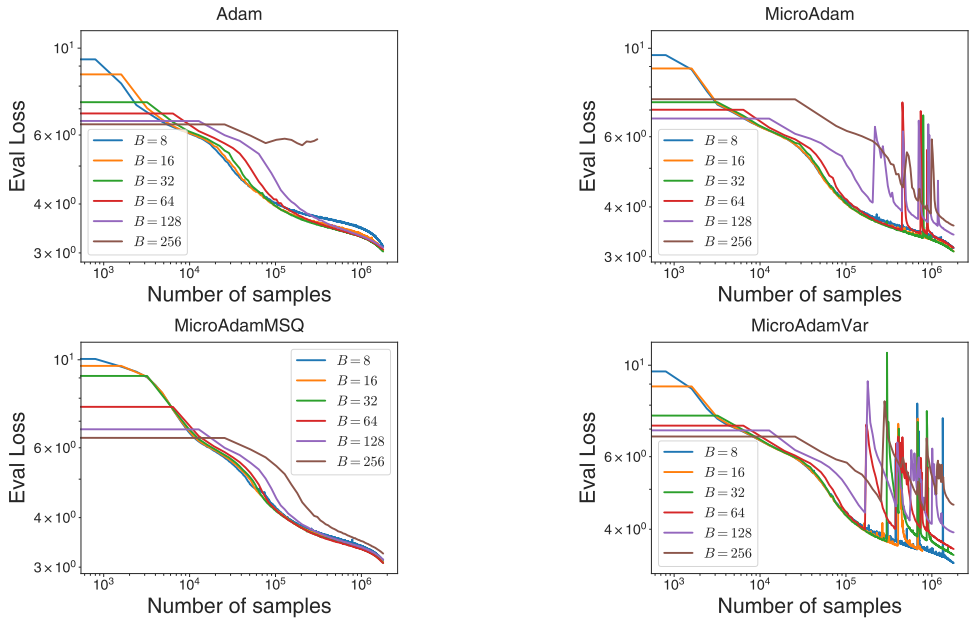

Figure 5: ADAM family variants trained at various batch sizes with their respective learning rate scaling rules. ADAM (top left) is trained with $\eta \propto \sqrt{B}$ and shows universal loss curves for intermediate batch size, but not for small or large batch size. MICROADAM (top right), MICROADAMMSQ (bottom left), and MICROADAMVAR (bottom right) all show universal scaling at small and intermediate batch sizes with $\eta \propto B$. ADAM family members with more σ^2 contribution to preconditioner suffer from stability issues, particularly MICROADAMVAR which *only* depends on σ^2. Learning rates are chosen to be close to optimal at $B = 64$.

We took advantage of our methodology to evaluate MICROADAM without approximation in our language model setting. Our aim was to evaluate both the batch size scaling behavior, as well as the overall performance. We first tuned the learning rate at batch size $B = 64$ and found that MICROADAM trains poorly compared to ADAM with instabilities (Figure 4). The training spikes could be mitigated by gradient clipping, but the final eval loss values were worse than ADAM (3.06 vs 3.10, Appendix D.2).

We then evaluated the batch size scaling behavior of MICROADAM. Using the optimal learning rate η^* at $B = 64$, we set the learning rate for other batch sizes using the predicted linear scaling rule $\eta = \eta^* B/64$. The resulting loss curves were plotted as a function of number of samples processed, where we see that for $B \in [8, 64]$ the learning curves follow the same universal form for much of training (Figure 5, top right). This is in contrast to the case of ADAM where we get universal curves for the square root learning rate rule $\eta = \eta^* \sqrt{B/64}$ (Figure 5, top left). We note that for ADAM, the smallest batch size $B = 8$ behaves in a non-universal way. This shows that the batch size scaling results previously approximated in Wang & Aitchison (2024) with "micro-batches" hold for a true implementation of MICROADAM as well. We note that the non-universality at larger B is observed across all the algorithms and is due to the transition to the large learning rate regime where non-linear effects become prominent (Cohen et al., 2021; Agarwala & Pennington, 2024; Cohen et al., 2024).

The results shown in Figure 4 suggest that increasing the effect of the σ^2 term by switching from ν_{adam} to ν_{micro} is detrimental to neural network training on this workload. This suggests that information from μ^2 may be more important to capture in the preconditioner. However, the square root learning rate scaling rule which holds for ADAM is motivated by the assumption that the variance dominates the preconditioner and $\mu^2 \ll \frac{1}{B}\sigma^2$ (Appendix D.1). How do we resolve these seemingly inconsistent observations?

We can measure μ^2 and σ^2 directly by constructing unbiased estimators using ν_{adam} and ν_{micro}:

$$\mathbb{E}\left[\frac{1}{1-B^{-1}}\left(\nu_{\text{adam}} - \frac{1}{B}\nu_{\text{micro}}\right)\right] = \mu^2, \qquad \mathbb{E}\left[\frac{1}{1-B^{-1}}\left(\nu_{\text{micro}} - \nu_{\text{adam}}\right)\right] = \sigma^2. \tag{8}$$

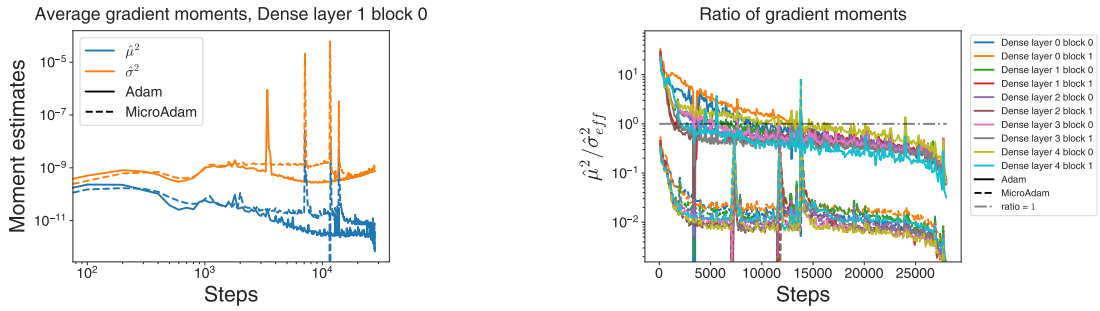

Figure 6: Estimator $\hat{\mu}^2$ is typically less than $\hat{\sigma}^2$ for both ADAM and MICROADAM, and quantities are similar for each algorithm at early times (left). However, the ratio of the μ^2 and σ^2 terms in ν_{adam} is greater than 1 at the start of training, and remains $O(1)$ until the end of training (right). In contrast, for MICROADAM the μ^2 information is always significantly smaller than the σ^2 term. Curves represent estimators using EMAs of ν_{adam} and ν_{micro}, averaged over all parameters in a given layer.

The latter is the traditional unbaised estimator of the variance using the sample variance. Therefore, if we can compute an estimate of $\mathbb{E}[\nu_{\text{adam}}]$ and $\mathbb{E}[\nu_{\text{micro}}]$ during training, we can directly compare μ^2 and σ^2.

Using EMA estimators, we measured μ^2 and σ^2 for both ADAM and MICROADAM. We used the same β_2 as the training algorithm so that ν_{adam} and ν_{micro} could be used for the training steps in their respective algorithms. We found that at early training times, ADAM and MICROADAM had similar values of $\hat{\mu}^2$ and $\hat{\sigma}^2$ (Figure 6, left). In both cases $\hat{\sigma}^2 > \hat{\mu}^2$, and the gap increased during training. In order to understand the relative importance of μ^2 and σ^2, we define $\sigma^2_{\text{eff}} = \frac{1}{B}\sigma^2$ for ADAM, and $\sigma^2_{\text{eff}} = \sigma^2$ for MICROADAM. The ratio $\hat{\mu}^2/\hat{\sigma}^2_{\text{eff}}$ gives a better sense of the dependence of ν_{adam} and ν_{micro} on the two terms, and reveals that at early times, ν_{adam} has a larger contribution from μ^2 than σ^2 (Figure 6, right). This is in contrast to MICROADAM which is dominated by σ^2 for most of training.

The trend of large μ^2 in ADAM is consistent across models of different sizes trained with different B (Figure 7). We found that for ADAM the ratio $\mu^2/\sigma^2_{\text{eff}}$ had a similar range across batch sizes when training with the square root scaling rule. We hypothesize that the dynamics induce a nontrivial relationship between μ^2, σ^2 and B, which can preserve the usefulness of the square root scaling rule. We leave detailed characterization and a true causal explanation of this phenomenon to future work.

3.2.2 A NEW FAMILY OF ADAM METHODS

The results of the previous section suggest that μ^2 information is better than σ^2 information in the ADAM preconditioner. This raises two additional questions: does a preconditioner based on σ^2 in (8) perform even worse than MICROADAM, and does one based on μ^2 in (8) perform better than ADAM?

We implemented these optimizers, which we call MICROADAMVAR and MICROADAMMSQ respectively, using linear combinations of ν_{adam} and ν_{micro} as we had for the measurements. Full pseudocodes of ADAM variants are presented in Appendix F. We found that indeed MICROADAMVAR had similar stability issues to and even worse performance than MICROADAM (Figure 5, bottom right). We also found that MICROADAMMSQ tended to destabilize in only a few hundred steps, since the estimator μ^2 in 8 is not guaranteed to be non-negative (ν_{adam} can be 0 for a finite sample, while ν_{micro} is always positive).

We solved the convergence issue by adding filtering to ν, so that the preconditioner is $\sqrt{\text{ReLU}(\nu)} + \epsilon$, with a larger $\epsilon = 10^{-6}$. Applying the ReLU only before applying the preconditioner maintains the unbiased nature of the estimator. This was combined with gradient clipping to obtain an algorithm which trains smoothly with slightly better performance than ADAM at batch size $B = 64$ (loss of 3.05 vs 3.06, Figure 4, left). It lagged slightly behind ADAM for most of training and then surpasses it at the end. As expected, MICROADAMMSQ shows universal curves over a large range of batch sizes with the scaling rule $\eta \propto B$ using the same methodology as for MICROADAM (Figure 5, bottom left, using $0.5\eta^* B$ where η^* is optimal at $B = 64$). There was 20% more wall-clock time with our `vmap` implementation.

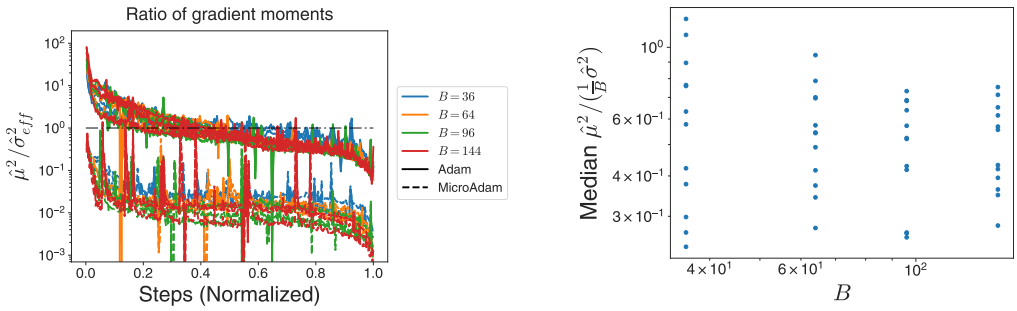

Figure 7: Ratios of $\hat{\mu}^2$ to $\hat{\sigma}^2_{eff}$ are consistent for models across different batch sizes with μ^2 dominating at early times for ADAM (left). For ADAM, median ratio of $\hat{\mu}^2/\frac{1}{B}\hat{\sigma}^2$ in different layers does not show much dependence on batch size (right), suggesting that the statistics of μ^2 must have a nontrivial relationship with σ^2/B to dominate the dynamics yet induce the square root learning rate heuristic.

A natural question is: why is ADAM the best stable member of this extended ADAM family? We observe that the ADAM preconditioner is the unique linear combination of ν_{adam} and ν_{micro} that minimizes the σ^2 term while maintaining non-negativity. This suggests that improvements to ADAM along these lines would require emphasizing the μ^2 information without instability due to small or negative contributions to the estimator. Our experiments with MICROADAMMSQ demonstrated one mitigation strategy but others (such as averaging ν_{micro} and ν_{adam} across blocks of parameters) are worth exploring.

4 DISCUSSION

One key finding of our methodological work is that the complexity of modern architectures and datasets actually *helps* methods that process gradient per-example. This reflects a broader point that there are many potential opportunities to improve training algorithms by exploiting dimensions which are underutilized but not limiting in terms of memory or compute. Our computational graph surgery approach also demonstrates that further improving the implementation of these methods is possible in some cases. That approach reflects the flexibility that differentiable programming languages like JAX offer with their native tracing of programs. We can use this approach on any factorable functions, including sign and power functions, and indeed any linear combination of factorable functions. This gives us access to a very broad set of per-example statistics.

Our experimental results show that access to per-example gradient statistics can dramatically improve our understanding of optimization in deep learning. Our most surprising result was that MICROADAM has stability and speed issues. By using the *measurements* enabled by our methods we were able to uncover evidence that the ADAM preconditioner is better when it is dominated by the mean squared rather than the variance — despite conventional wisdom like the explanation of the square root scaling rule. This is consistent with previous work showing that ADAM performs better with less noisy gradients (Kunstner et al., 2023), and overall suggests that constructing better estimators for this information might help improve ADAM.

Altogether, these results suggest that per-example gradient measurements and transformations are an exciting new area for optimization research. We can now test ideas about manipulating distributions of gradients in modern settings (Zielinski et al., 2020). We hope our methods will be used to develop new perspectives and paradigms in deep learning training algorithms in the near future.

Reproducibility statement. All details on architecture, dataset and optimizers are presented in Appendix E. Pseudo-code for the Adam variants is presented in Appendix F. The experiments are based on publicly available code `https://github.com/google-deepmind/nanodo`. The authors plan to directly contribute to that codebase by means of including "`vmap`" versions of the presented algorithms to reproduce all experiments and go beyond.

References

Atish Agarwala and Jeffrey Pennington. High dimensional analysis reveals conservative sharpening and a stochastic edge of stability. *arXiv preprint arXiv:2404.19261*, 2024.

Lukas Balles, Fabian Pedregosa, and Nicolas Le Roux. The geometry of sign gradient descent. *arXiv preprint arXiv:2002.08056*, 2020.

Jeremy Bernstein, Yu-Xiang Wang, Kamyar Azizzadenesheli, and Animashree Anandkumar. signsgd: Compressed optimisation for non-convex problems. In *International conference on machine learning*, pp. 560–569. PMLR, 2018.

Mathieu Blondel and Vincent Roulet. The elements of differentiable programming. *arXiv preprint arXiv:2403.14606*, 2024.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: Autograd and XLA. http://github.com/google/jax, 2018. Accessed: 2025-08-27.

Zhiqi Bu, Jialin Mao, and Shiyun Xu. Scalable and efficient training of large convolutional neural networks with differential privacy. *Advances in Neural Information Processing Systems*, 35:38305–38318, 2022.

Zhiqi Bu, Yu-Xiang Wang, Sheng Zha, and George Karypis. Differentially private optimization on large model at small cost. In *International Conference on Machine Learning*, pp. 3192–3218. PMLR, 2023.

Jeremy Cohen, Simran Kaur, Yuanzhi Li, J Zico Kolter, and Ameet Talwalkar. Gradient descent on neural networks typically occurs at the edge of stability. In *International Conference on Learning Representations*, 2021.

Jeremy M. Cohen, Alex Damian, Ameet Talwalkar, Zico Kolter, and Jason D. Lee. Understanding optimization in deep learning with central flows, 2024. URL https://arxiv.org/abs/2410.24206.

Enea Monzio Compagnoni, Tianlin Liu, Rustem Islamov, Frank Norbert Proske, Antonio Orvieto, and Aurelien Lucchi. Adaptive methods through the lens of sdes: Theoretical insights on the role of noise. *arXiv preprint arXiv:2411.15958*, 2024.

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

Arthur Douillard, Qixuan Feng, Andrei A Rusu, Rachita Chhaparia, Yani Donchev, Adhiguna Kuncoro, Marc'Aurelio Ranzato, Arthur Szlam, and Jiajun Shen. Diloco: Distributed low-communication training of language models. *arXiv preprint arXiv:2311.08105*, 2023.

Milan Ganai, Haichen Li, Theodore Enns, Yida Wang, and Randy Huang. Target-independent xla optimization using reinforcement learning. *arXiv preprint arXiv:2308.14364*, 2023.

Andreas Griewank and Andrea Walther. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM, 2008.

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

Weiwei Kong and Andres Munoz Medina. A unified fast gradient clipping framework for dp-sgd. *Advances in Neural Information Processing Systems*, 36, 2023.

Frederik Kunstner, Jacques Chen, Jonathan Wilder Lavington, and Mark Schmidt. Noise is not the main factor behind the gap between sgd and adam on transformers, but sign descent might be. *arXiv preprint arXiv:2304.13960*, 2023.

Seppo Linnainmaa. *The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors*. PhD thesis, Master's Thesis (in Finnish), Univ. Helsinki, 1970.

Peter J. Liu, Roman Novak, Jaehoon Lee, Mitchell Wortsman, Lechao Xiao, Katie Everett, Alexander A. Alemi, Mark Kurzeja, Pierre Marcenac, Izzeddin Gur, Simon Kornblith, Kelvin Xu, Gamaleldin Elsayed, Ian Fischer, Jeffrey Pennington, Ben Adlam, and Jascha-Sohl Dickstein. Nanodo: A minimal transformer decoder-only language model implementation in JAX., 2024. URL http://github.com/google-deepmind/nanodo.

Zakaria Mhammedi, Andrew Hellicar, Ashfaqur Rahman, and James Bailey. Efficient orthogonal parametrisation of recurrent neural networks using householder reflections. In *International Conference on Machine Learning*, pp. 2401–2409. PMLR, 2017.

Roman Novak, Jascha Sohl-Dickstein, and Samuel S Schoenholz. Fast finite width neural tangent kernel. In *International Conference on Machine Learning*, pp. 17018–17044. PMLR, 2022.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Soumith Chintala, and Ronan Collobert. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pp. 8024–8035. 2019.

Shikai Qiu, Lechao Xiao, Andrew Gordon Wilson, Jeffrey Pennington, and Atish Agarwala. Scaling collapse reveals universal dynamics in compute-optimally trained neural networks. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=Fvq9ogLnLN.

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.

Christopher J. Shallue, Jaehoon Lee, Joseph Antognini, Jascha Sohl-Dickstein, Roy Frostig, and George E. Dahl. Measuring the effects of data parallelism on neural network training. *Journal of Machine Learning Research*, 20(112):1–49, 2019. URL http://jmlr.org/papers/v20/18-789.html.

Xi Wang and Laurence Aitchison. Batch size invariant adam. *arXiv preprint arXiv:2402.18824*, 2024.

Ke Liang Xiao, Noah Marshall, Atish Agarwala, and Elliot Paquette. Exact risk curves of signsgd in high-dimensions: quantifying preconditioning and noise-compression effects. In *Forty-second International Conference on Machine Learning*, 2025.

Dong Yin, Ashwin Pananjady, Max Lam, Dimitris Papailiopoulos, Kannan Ramchandran, and Peter Bartlett. Gradient diversity: a key ingredient for scalable distributed learning. In *International Conference on Artificial Intelligence and Statistics*, pp. 1998–2007, 2018.

Yijia Zhang, Yibo Han, Shijie Cao, Guohao Dai, Youshan Miao, Ting Cao, Fan Yang, and Ningyi Xu. Adam accumulation to reduce memory footprints of both activations and gradients for large-scale dnn training. *arXiv preprint arXiv:2305.19982*, 2023.

Rosie Zhao, Depen Morwani, David Brandfonbrener, Nikhil Vyas, and Sham Kakade. Deconstructing what makes a good optimizer for language models. *arXiv preprint arXiv:2407.07972*, 2024.

Piotr Zielinski, Shankar Krishnan, and Satrajit Chatterjee. Weak and strong gradient directions: Explaining memorization, generalization, and hardness of examples at scale. *arXiv preprint arXiv:2003.07422*, 2020.

A Just-in-time compilation, tracing, and code optimization

A simple implementation of estimators like $\frac{1}{B}\sum_{i=1}^{B}\phi(\nabla f(\theta; x_i)) \approx \mathbb{E}[\phi(\nabla f(\theta; X))]$ would consist in (i) computing B gradients, (ii) applying the non-linear operation ϕ, (iii) averaging the result. In JAX, computing the average elementwise square gradient would look like this:

```
import jax
import jax.numpy as jnp

B, d = 10, 4
fun = lambda w, x: jnp.sum(jnp.dot(x, w))
w, xs = jnp.ones((d, d)), jnp.ones((B, d))
indiv_grads = jax.vmap(jax.grad(fun), in_axes=(None, 0), out_axes=0)(w, xs)
avg_sq_grads = jnp.mean(indiv_grads**2, axis=0)
```

An immediate potential issue with this approach is the memory cost: computing B gradients would require B times the space necessary to store the parameters of the network. In many architectures such a cost is prohibitive.

However, not only such an implementation may be too naive, but it is also a naive view of how code is compiled in deep learning frameworks, like JAX (Bradbury et al., 2018) or Pytorch (Paszke et al., 2019). Such frameworks offer *just-in-time compilations (jit)* techniques that can tailor the compilations to a given backend and optimize the implementation to some extent. Below is the "jitting" for the average elementwise square.

```
def compute_avg_sq_grads_(w, xs):
    indiv_grads = jax.vmap(jax.grad(fun), in_axes=(None, 0), out_axes=0)(w, xs)
    return jnp.mean(indiv_grads**2, axis=0)
compute_avg_sq_grads = jax.jit(compute_avg_sq_grads_)
```

Just-in-time compilation of a program $\mathcal{P}$ takes typical inputs of P, i.e., inputs with the shape and type that $\mathcal{P}$ will be run with, and lists all operations involved in $\mathcal{P}$ in order with the size and types of their inputs and outputs. Such a process is called *tracing* the program. In JAX, we get then access to the computational graph of $\mathcal{P}$ in the form of of a *jaxpr* that lists operations as JAX primitives in the topological order of the computational graph. Provided with such a computational graph in terms of some primitives, we can define a new computational graph to compute for example gradients of the outputs of $\mathcal{P}$ w.r.t. its inputs. Below is the jaxpr for the average elementwise square.

```
print(jax.make_jaxpr(compute_avg_sq_grads_)(w, xs))
```

```
{ lambda ; a:f32[4,4] b:f32[10,4]. let
    c:f32[10,4] = dot_general[
      dimension_numbers=(([1], [0]), ([], []))
      preferred_element_type=float32
    ] b a
    _:f32[10] = reduce_sum[axes=(1,)] c
    d:f32[4] = broadcast_in_dim[broadcast_dimensions=() shape=(4,)] 1.0
    e:f32[4,10,4] = dot_general[
      dimension_numbers=(([], []), ([], []))
      preferred_element_type=float32
    ] d b
    f:f32[10,4,4] = transpose[permutation=(1, 2, 0)] e
    g:f32[10,4,4] = integer_pow[y=2] f
    h:f32[4,4] = reduce_sum[axes=(0,)] g
    i:f32[4,4] = div h 10.0
  in (i,) }
```

The computational graph leaves also room for optimizations and tailored implementations for given hardwares. For that, jaxprs are first converted in *high-level operations (HLO)* code, that are intermediate representations. They are independent of the framework (JAX, Pytorch, Tensorflow, etc...), so lower level than

JAX. But they are also independent of the hardware, so higher level than targeted code for e.g. GPUs. Below is the hlo code for the average elementwise square.

```
print(compute_avg_sq_grads.lower(w, xs).as_text())
```

```
module @jit_compute_avg_sq_grads_ attributes {mhlo.num_partitions = 1 : i32, mhlo.num_replicas
    = 1 : i32} {
nc.func public @main(%arg0: tensor<10x4xf32> {mhlo.layout_mode = "default"}) ->
    (tensor<4x4xf32> {jax.result_info = "", mhlo.layout_mode = "default"}) {
%cst = stablehlo.constant dense<1.000000e+00> : tensor<f32>
%0 = stablehlo.broadcast_in_dim %cst, dims = [] : (tensor<f32>) -> tensor<4xf32>
%1 = stablehlo.dot_general %0, %arg0, contracting_dims = [] x [], precision = [DEFAULT,
    DEFAULT] : (tensor<4xf32>, tensor<10x4xf32>) -> tensor<4x10x4xf32>
%2 = stablehlo.transpose %1, dims = [1, 2, 0] : (tensor<4x10x4xf32>) -> tensor<10x4x4xf32>
%3 = stablehlo.multiply %2, %2 : tensor<10x4x4xf32>
%cst_0 = stablehlo.constant dense<0.000000e+00> : tensor<f32>
%4 = stablehlo.reduce(%3 init: %cst_0) applies stablehlo.add across dimensions = [0] :
    (tensor<10x4x4xf32>, tensor<f32>) -> tensor<4x4xf32>
%cst_1 = stablehlo.constant dense<1.000000e+01> : tensor<f32>
%5 = stablehlo.broadcast_in_dim %cst_1, dims = [] : (tensor<f32>) -> tensor<4x4xf32>
%6 = stablehlo.divide %4, %5 : tensor<4x4xf32>
return %6 : tensor<4x4xf32>
```

HLO code can then be parsed to find simplifications or optimizations. Note that finding the high-level optimal implementation of a program may not be possible. The optimization of the implementation is done by parsing the graph with a series of simple rules (some available publicly on the OpenXLA project) that search for specific simplifications. The order in which the simplifications are done may even have an impact on the final implementation (Ganai et al., 2023).

In our case of interest, i.e., computing (2), a compiler may know the memory limits of the given hardware and accumulate the average by computing gradients one by one rather than computing all gradients at once. This would amount to *fuse* some operations in the graph (like the computations associated to `e`, `f`, `g`, `h` in the jaxpr above). This would implement a computationally intensive approach outlined in Section 2.1 but at the scale of the leaf of the computational graph. The trade-off memory vs time cost may further be optimized by the compiler by computing micro-batch of gradients at a time. Once the HLO code is optimized it is further converted in low-level code that further optimizes the code for the given hardware backend, see the documentation of the OpenXLA project https://www.openxla.org/xla/ for further explanations.

To summarize, the actual implementation in frameworks like JAX involves many more moving parts than what a high-level python code looks like. An implementation using `vmap` typically already uses the trace of the computations of the original functions to modify appropriately the code for the needs of the user. For sequence-level models, the performance of the `vmap` approach only incurs a small computational overhead as illustrated in Figure 2. However, we have also evidence that a jitted `vmap` may not provide the best implementation in some other cases, see Figure 8. In general, a computational graph surgery can ensure efficient implementations. Some simplifications found in the computations of gradient statistics — like for computing the sum of square gradients in Section 2.2 — could also be implemented as parts of the optimization of the HLO code to further improve the native performance of `vmap`.

B Computational graph surgery

B.1 From broadcasting weights, to reducing gradients, and injecting new statistics

We formalize Figure 1 to identify where, in the computational graph of the average gradients (1), is performed the mean reduction.

We analyze the loss $f(\theta; x_i)$ of a network with no shared weights schematized in Figure 1. We do not restrict ourselves to feed-forward networks. We examine the computation of the gradient of the sum of the losses, $\sum_{i=1}^{B} f(\theta; x_i)$. In the computation of the loss $f(\theta, x_i)$, we assume that the weight W_k of the kth operation

acts on the intermediate representation s_i of the input x_i through some bilinear operation ℓ to output

$$t_i = \ell(s_i, W_k).$$

For dense layers, ℓ is a vector-matrix product, for convolutional layers, it's a convolution, etc... Some sequence of operations later, the loss associated to this sample is computed and, as all samples in the mini-batch are processed, the average loss is computed, see the forward pass in Figure 1.

Computing $\nabla_{W_k} \sum_{i=1}^{B} f(\theta, x_i)$, i.e., the gradient of the sum of the losses w.r.t. the weight W_k, amounts then to computing the gradient of a function of the form

$$W_k \mapsto \sum_{i=1}^{B} h_i(\ell_i(W_k)),$$

where $\ell_i = \ell(s_i, \cdot)$ and h_i denotes subsequent operations on s_i, that may depend on the sample x_i (if for example the sample actually consists of an input/output pair). This evaluation consists essentially in B independent computational paths that are linked at the operational index k by the weight W_k, and at the end by the reduction, see the forward pass in Figure 1.

Linear operations generally define a batch dimension along which inputs s_i are treated by the same right operand W_k. This can be formalized by introducing the broadcast operation

$$\in\!\!\!\!- : W \mapsto (W, \ldots, W),$$

that broadcasts the weights to the computation of the linear transformation of each input. We can then rewrite the computation of the gradient of the sum w.r.t. $W = W_k$ as

$$\nabla_W \left(\sum_{i=1}^{B} f(\theta; x_i) \right) = \nabla_W \left(\sum_{i=1}^{B} h_i(\ell_i(W)) \right) = \nabla \left(\oplus \circ h_{/\!/} \circ \ell_{/\!/} \circ \in\!\!\!\!- \right)(W),$$

where

$$\oplus : (u_1, \ldots, u_B) \mapsto \sum_{i=1}^{B} u_i,$$

is the sum reduction, $\ell_{/\!/} : (W, \ldots, W) \mapsto (\ell_1(W), \ldots, \ell_B(W))$ denotes the execution of the linear operations on the replicas of the weights and the different inputs, and $h_{/\!/} : (t_1, \ldots, t_B) \mapsto (h_1(t_1), \ldots, h_B(t_B))$ denotes the rest of the computation.

Computing the gradient amounts to composing the Vector-Jacobian Products (VJP) of the operations, i.e., the adjoint[1] of their linear approximations on their inputs, in reverse order. Namely, following e.g. Blondel & Roulet (2024, Chapter 2),

$$\nabla \left(\oplus \circ h_{/\!/} \circ \ell_{/\!/} \circ \in\!\!\!\!- \right)(W) = \partial \left(\oplus \circ h_{/\!/} \circ \ell_{/\!/} \circ \in\!\!\!\!- \right)(W)^* 1 = \partial \in\!\!\!\!-(W)^* \partial \ell_{/\!/}(V)^* \partial h_{/\!/}(t)^* \partial \in\!\!\!\!-(u)^* 1,$$

where, for a function f, $\partial f(x)$ denotes its linearization around an input x, for a linear operator a, a^* denotes its adjoint, and we used the shorthands $V = \in\!\!\!\!-(W), t = \ell_{/\!/}(V), u = h_{/\!/}(t)$.

Since the broadcast and reduce-sum operations are already linear, their VJPs consist simply in their adjoint operations, i.e., $\in\!\!\!\!-(W)^* = \in\!\!\!\!-^*$ and $\oplus(V)^* = \oplus^*$. Moreover, one easily verifies that broadcast and reduce-sum are adjoint to each other, i.e., $\in\!\!\!\!-^* = \oplus$, and $\oplus^* = \in\!\!\!\!-$. One also easily observes that the linearizations $H_{/\!/} := \partial h_{/\!/}(t)^*, L_{/\!/} := \partial \ell_{/\!/}(V)^*$, of, respectively, $h_{/\!/}$ and $\ell_{/\!/}$ on their inputs, form independent paths of computations just as $h_{/\!/}$ and $\ell_{/\!/}$ did in their forward pass. The sum of gradients w.r.t. W is then computed as

$$\nabla \left(\oplus \circ h_{/\!/} \circ \ell_{/\!/} \circ \in\!\!\!\!- \right)(W_k) = \in\!\!\!\!-^* \circ L_{/\!/}^* \circ H_{/\!/}^* \circ \oplus^* 1 = \oplus \circ L_{/\!/}^* \circ H_{/\!/}^* \circ \in\!\!\!\!- 1.$$

We see —as illustrated in Figure 1— that the sum reduction happens theoretically as the last operation in the computation of the sum of the gradients. This is not due to the sum reduction of the losses (that is now

[1] For a linear operator $a : \mathcal{E} \mapsto \mathcal{F}$ from a Euclidean space $\mathcal{E}$ to an Euclidean space $\mathcal{F}$, equipped with respective inner products $\langle \cdot, \cdot \rangle_{\mathcal{E}}$ and $\langle \cdot, \cdot \rangle_{\mathcal{F}}$, the adjoint of a is the unique operator, denoted a^* such that for any $u, v \in \mathcal{E} \times \mathcal{F}$, $\langle a(u), v \rangle_{\mathcal{F}} = \langle u, a^*(v) \rangle_{\mathcal{E}}$.

a first broadcast operation in the computation of the gradient), but is due to the adjoint operator of the underlying broadcast of the weights in the forward pass.

To compute estimators of the form $1/B \sum_{i=1}^{B} \phi(\nabla f(\theta; x_i))$ as in (2), it suffices a priori to inject the nonlinear function just before the last operation as

$$\sum_{i=1}^{B} \phi(\nabla f(W; z_i)) = \oplus \circ \phi \circ L_{/\!/}^{*} \circ H_{/\!/}^{*} \circ \in 1.$$

In practice, we do not have direct access to a computation graph of the sum of the gradient of the form $\oplus \circ L_{/\!/}^{*} \circ H_{/\!/}^{*} \circ \in 1$. The reduction is performed as part of the VJP of $W_k \mapsto (\ell(s_1, W_k), \ldots, \ell(s_B, W_k))$. Diving into some specific forms show what opportunities exist for some linear operations as in dense layers, as presented in Appendix B.5. The generic viewpoint presented above lays down the foundations for a generic implementations of estimators (2) in a differentiable programming framework like JAX (Bradbury et al., 2018) by parsing the jaxpr of the gradient as explained in the next section.

B.2 Jaxpr surgery

As mentioned in Appendix A, JAX can and does trace the code, either to transform it (to provide the computational graph of the gradient) or to compile it efficiently. For our purposes, it means that we have access to jaxprs as the one below.

```
import jax
import jax.numpy as jnp
import jax.random as jrd

B, d = 10, 4
fun = lambda w, x: jnp.sum(jnp.dot(x, w))
w, xs = jrd.normal(jrd.key(0), (d, d)), jrd.normal(jrd.key(1), (B, d))
loss = lambda w, x: jnp.sum(jnp.dot(x, w)**2, axis=-1)
sum_loss = lambda w, xs: jnp.sum(loss(w, xs))
sum_grad_jaxpr = jax.make_jaxpr(jax.grad(sum_loss))(w, xs)

print('Original jaxpr of sum of grads:\n', sum_grad_jaxpr)
```

```
Original jaxpr of sum of grads:
 { lambda ; a:f32[4,4] b:f32[10,4]. let
    c:f32[10,4] = dot_general[
      dimension_numbers=(([1], [0]), ([], []))
      preferred_element_type=float32
    ] b a
    d:f32[10,4] = integer_pow[y=2] c
    e:f32[10,4] = integer_pow[y=1] c
    f:f32[10,4] = mul 2.0 e
    g:f32[10] = reduce_sum[axes=(1,)] d
    _:f32[] = reduce_sum[axes=(0,)] g
    h:f32[10] = broadcast_in_dim[broadcast_dimensions=() shape=(10,)] 1.0
    i:f32[10,4] = broadcast_in_dim[broadcast_dimensions=(0,) shape=(10, 4)] h
    j:f32[10,4] = mul i f
    k:f32[4,4] = dot_general[
      dimension_numbers=(([0], [0]), ([], []))
      preferred_element_type=float32
    ] j b
    l:f32[4,4] = transpose[permutation=(1, 0)] k
  in (l,) }
```

Jaxprs encode a list of operations in the topological order of the computational graph. We can then encode computational graph surgeries by parsing the jaxpr to find where reduction happens as done below.

```
import copy
```

```python
import jax.extend as jex

def collect_reduce_ops(
    sum_grad_jaxpr: jex.core.Jaxpr,
) -> tuple[list[jex.core.JaxprEqn], list[jex.core.JaxprEqn]]:
  """Parse the jaxpr to collect last reduce operations before computing sum of gradients."""

  # Some primitives can safely be ignored
  invariant_primitives = ['transpose', 'reshape']

  # For this example, we simply treat 'dot_general' to showcase the overall implementation
  # Our library can handle other of the main reducing primitives like
  # 'reduce_sum', 'conv' etc...
  reduce_primitives = ['dot_general']

  reduce_ops = []

  # We track the reduce ops through the variables output when computing gradient
  grad_outvars = copy.copy(sum_grad_jaxpr.outvars)

  for eqn in sum_grad_jaxpr.eqns[::-1]:
    primitive_name = eqn.primitive.name
    outvar = eqn.outvars[0]
    if outvar in grad_outvars:
      if primitive_name in invariant_primitives:
        # Add invar to list of variables tracked to find reduce ops
        invar = eqn.invars[0]
        grad_outvars.append(invar)
      if primitive_name in reduce_primitives:
        # Primitive found
        reduce_ops.append(eqn)
  return reduce_ops
```

Once we collected the operations we will change in the computational graph, we can simply parse the jaxpr as if it was normally evaluated and inject the transforms we want.

```python
from typing import Callable, Any

def jaxpr_surgery(
    sum_grad_jaxpr: jex.core.ClosedJaxpr,
    reinterpreter: Callable[[jex.core.JaxprEqn, jax.Array], jax.Array],
) -> Callable[..., jax.Array]:

  reduce_ops = collect_reduce_ops(sum_grad_jaxpr.jaxpr)
  def reinterpreted_sum_grad(
      params: list[jax.Array], *extra_fun_args: Any
  ) -> jax.Array:
    flat_params, params_treedef = jax.tree.flatten(params)
    flat_extra_fun_args = jax.tree.leaves(extra_fun_args)
    env = {}

    def read(var: jex.core.Var):
      if isinstance(var, jex.core.Literal):
        return var.val
      return env[var]

    def write(var: jex.core.Var, val: jax.Array):
      env[var] = val

    jaxpr = sum_grad_jaxpr.jaxpr
    consts = jaxpr.constvars
```

```
    args = flat_params + list(flat_extra_fun_args)

    jax.util.safe_map(write, jaxpr.invars, args)
    jax.util.safe_map(write, jaxpr.constvars, consts)

    for eqn in jaxpr.eqns:
      if eqn in reduce_ops:
        # Reinterpret the reduce operation
        invals = jax.util.safe_map(read, eqn.invars)
        ans = reinterpreter(eqn, invals)
      else:
        # Usual pass on the jaxpr graph
        subfuns, bind_params = eqn.primitive.get_bind_params(eqn.params)
        invals = jax.util.safe_map(read, eqn.invars)
        ans = eqn.primitive.bind(*subfuns, *invals, **bind_params)
      if eqn.primitive.multiple_results:
        jax.util.safe_map(write, eqn.outvars, ans)
      else:
        write(eqn.outvars[0], ans)

    flat_grads_like = jax.util.safe_map(read, jaxpr.outvars)
    return jax.tree.unflatten(params_treedef, flat_grads_like)

  return reinterpreted_sum_grad
```

The core and growing part of the library is then to implement reinterpreters that, for each possible reduction leading to a sum of gradients, implements the desired statistics. Below we present a simple and brief implementation for the sum of square gradients.

```
def sum_square_reinterpreter(eqn: jex.core.JaxprEqn, invals: tuple[jax.Array, ...]) ->
    jax.Array:
  if eqn.primitive.name == 'dot_general':
    lhs_contracting_dims, rhs_contracting_dims = eqn.params['dimension_numbers'][0]
    if len(lhs_contracting_dims) == len(rhs_contracting_dims) == 1:
      # This is the dense layer case operating on vectors
      # We can simply square the inputs and apply the original matrix product
      sq_invals = [inval**2 for inval in invals]
      sum_sq_sgrads = eqn.primitive.bind(*sq_invals, **eqn.params)
    else:
      # Reinterpreter can be extended for more generic cases of dot-general
      # as in sequence base model.
      raise NotImplementedError
    return sum_sq_sgrads
  else:
    # Reinterpreter can be extended for other operations like convolution.
    raise NotImplementedError()
```

We can then instantiate our desired "sum of square gradients" oracle and compare to the vmap implementation.

```
compute_sum_sq_grads = jaxpr_surgery(sum_grad_jaxpr, sum_square_reinterpreter)
print('Modified Jaxpr:\n', jax.make_jaxpr(compute_sum_sq_grads)(w, xs))

def compute_sum_sq_grads_via_vmap(w, xs):
  return jnp.sum(jax.vmap(jax.grad(loss), (None, 0))(w, xs)**2, axis=0)
print('Jaxpr of vmap implementation:\n', jax.make_jaxpr(compute_sum_sq_grads_via_vmap)(w, xs))

print(
  'Do implementations match?\n',
  jnp.all(jnp.equal(compute_sum_sq_grads(w, xs), compute_sum_sq_grads(w, xs)))
)
```

```
Modified Jaxpr:
 { lambda ; a:f32[4,4] b:f32[10,4]. let
    c:f32[10,4] = dot_general[
      dimension_numbers=(([1], [0]), ([], []))
      preferred_element_type=float32
    ] b a
    d:f32[10,4] = integer_pow[y=2] c
    e:f32[10,4] = integer_pow[y=1] c
    f:f32[10,4] = mul 2.0 e
    g:f32[10] = reduce_sum[axes=(1,)] d
    _:f32[] = reduce_sum[axes=(0,)] g
    h:f32[10] = broadcast_in_dim[broadcast_dimensions=() shape=(10,)] 1.0
    i:f32[10,4] = broadcast_in_dim[broadcast_dimensions=(0,) shape=(10, 4)] h
    j:f32[10,4] = mul i f
    k:f32[10,4] = integer_pow[y=2] j
    l:f32[10,4] = integer_pow[y=2] b
    m:f32[4,4] = dot_general[
      dimension_numbers=(([0], [0]), ([], []))
      preferred_element_type=float32
    ] k l
    n:f32[4,4] = transpose[permutation=(1, 0)] m
  in (n,) }
Jaxpr of vmap implementation:
 { lambda ; a:f32[4,4] b:f32[10,4]. let
    c:f32[10,4] = dot_general[
      dimension_numbers=(([1], [0]), ([], []))
      preferred_element_type=float32
    ] b a
    d:f32[10,4] = integer_pow[y=2] c
    e:f32[10,4] = integer_pow[y=1] c
    f:f32[10,4] = mul 2.0 e
    _:f32[10] = reduce_sum[axes=(1,)] d
    g:f32[4] = broadcast_in_dim[broadcast_dimensions=() shape=(4,)] 1.0
    h:f32[1,4] = broadcast_in_dim[broadcast_dimensions=(1,) shape=(1, 4)] g
    i:f32[10,4] = mul h f
    j:f32[10,4,4] = dot_general[
      dimension_numbers=(([], []), ([0], [0]))
      preferred_element_type=float32
    ] i b
    k:f32[10,4,4] = transpose[permutation=(0, 2, 1)] j
    l:f32[10,4,4] = integer_pow[y=2] k
    m:f32[4,4] = reduce_sum[axes=(0,)] l
  in (m,) }
Do implementations match?
 True
```

The code above can be extended to a library and benchmarked against vmap.

B.3 Jaxpr surgery vs vmap

As explained in Appendix A, the native `vmap` transformation in JAX can benefit from optimizations at the HLO code level or even at a lower level of hardware. So even though the jaxprs of the vmap and the jaxpr surgery in the example above differ, the real performance of these methdos can only be judged in the final code execution.

To compare the jaxpr surgery and the `vmap` approaches, we implement the MicroAdam variant of Adam that uses the average of square gradients rather than the square average of gradients (see Section 3.2). We implement it on MLPs with constant width (hidden dimension) across layers and measure the execution times of both approaches as depth, width or batch size vary, see Figure 8. For moderate batch sizes $B = 64$, the jaxpr surgery is moderately faster than the `vmap` implementation across widths and depths (Figure 8

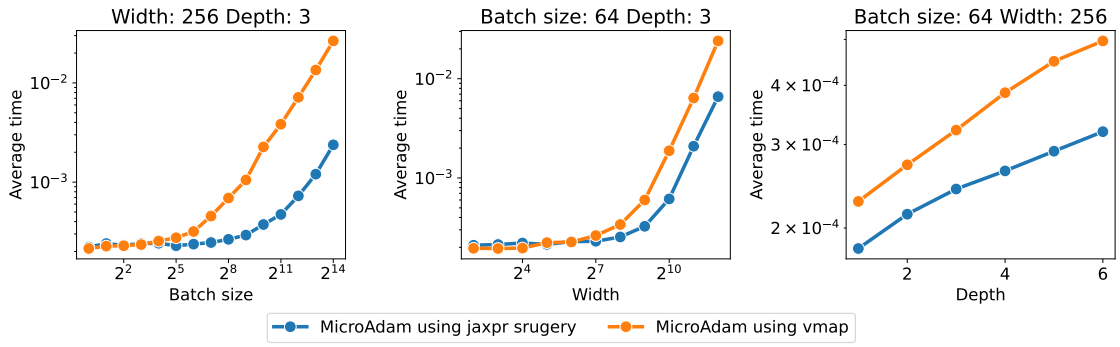

Figure 8: Execution times of MicroAdam with `vmap` or using a jaxpr surgery. The jaxpr surgery is generally faster and in particular scale much better with the batch size (note the log scale on the vertical time axis).

middle and right panels) The jaxpr surgery can be an order of magnitude faster for batch sizes larger than $2^8 = 256$ (Figure 8 left panel). Both implementations follow similar trends with increasing batch size.

B.4 Memory bottlenecks in deep networks

Section 2.3 summarized memory footprints of gradient computations for dense layers applied to either vector inputs or sequences of vectors. The remark can be summarized and generalized as a simple fact.

Fact B.1. *In a network, for any parameterized operation processing inputs representations $s \in \mathbb{R}^R$ using parameters $w \in \mathbb{R}^P$, if the dimension of the input is greater than the dimension of the parameters, i.e., $R > P$, then the cost of storing B individual gradients of the loss of the network w.r.t. w is no larger than the checkpointing cost necessary for computing the mini-batch gradients. So if the memory used to store the checkpoints is freed to store the B individual gradients, the peak memory usage of computing the mini-batch gradients is the same as the memory cost to compute the individual gradients and then averaging them.*

Proof. Computing the mini-batch gradient requires checkpointing B inputs s at a cost BD. Storing B gradients take BP memory. Memory allocations along program execution are well explained by e.g. Griewank & Walther (2008, Chapter 4). □

Fact B.1 may still lower bound the memory necessary for computing the mini-batch loss. Some non-parameterized operations may require extensive additional memory. This is typically the case for attention blocks in transformers that require a priori BL^2 additional memory to checkpoint the inputs of the activation module.

We can parse common layers through this fact.

1. For dense layers operating on vectors, i.e., of the form $t^\top = s^\top W$, with $t \in \mathbb{R}^D, s \in \mathbb{R}^D$, $W \in \mathbb{R}^{D\times D}$, Fact B.1 does not apply as we have $R = D < D^2 = P$.
2. For dense layers operating on sequences of vectors, i.e., of the form $T = SW$, for $T \in \mathbb{R}^{L\times D}, S \in \mathbb{R}^{L\times D}$, $W \in \mathbb{R}^{D\times D}$, as long as L the length of the sequence is less than D, the width, Fact B.1 applies.
   - This may apply for example in the dense layers of the multihead attention and the mlp blocks of a transformer.
   - This would not apply to the embedding layer and the final head of a transformer whose dimensions depend on the vocabulary that generally has a very large size.
3. Convolutional layers operating on images of size HWC for H, W, C the height, width and number of channels of the image. The parameters are filters of size K^2C^2, with K the kernel size. If $HW > KC$, Fact B.1 applies.

The first point of Fact B.1 is easy to check. The second point depends on the compilation of the program and compiler optimizations (Appendix A). Fact B.1 simply hints that computations involving individual gradients *may* not be prohibitive for some architectures given an efficient just-in-time compiler.

Figure 2 shows the memory allocated along program execution for the ADAM and MICROADAM algorithms on a transformer. We observe that the peak memory is the same. It corresponds well to the peak activation required for reverse-mode automatic differentiation (compare to Figure 8.4 in (Blondel & Roulet, 2024)).

The unchanged peak memory usage may be surprising as individual gradients of the embedding layer may still account for an important part of the computational cost. However, in any case, just-in-time compilation in JAX (Appendix A) may automatically trade-off memory and computational usages.

B.5 MINI-BATCH GRADIENT STRUCTURE

Tensor contractions. We keep the framework outlined in B.1, i.e., we consider networks where weights define a linear operation on intermediate inputs and output $t_i = \ell(s_i, W_k)$. Here, we generalize the matrix-vector product case presented in Section 2.2 and consider generic tensor contractions.

Forward operation. Dropping the index k of the operation in the weight $W = W_k$, and denoting $T = (t_1, \dots, t_B), S = (s_1, \dots, s_B)$, the batch operation reads $T = \ell(S, W)$ where T, S, W are all multi-axes arrays. The tensor contraction consists in summing products of S and W along a set K of *contraction* axes, and doing this for all "free indexes" of S and W. Formally, following Einstein summation notations we get, for any $i_1, \dots, i_m$ and $j_1, \dots, j_n$ taken along a set of axes I and J respectively,

$$T^{i_1,\dots,i_m}_{j_1,\dots,j_n} = \sum_{k_1,\dots,k_p} S^{i_1,\dots,i_m}_{k_1,\dots,k_p} W^{k_1,\dots,k_p}_{j_1,\dots,j_n}, \tag{9}$$

with $m = |I|, n = |J|, p = |P|$. The axes I along which $i_1, \dots, i_m$ are taken can be seen as the "data parallel" indexes along which the content of the intermediate representations S is processed. In particular, i_1 will denote the mini-batch axis along which we compute the independent computational paths of the losses. The axes J along which $j_1, \dots, j_n$ are taken can be seen as the "new dimensions" of the intermediate representations.

Examples. For a dense layer operating on vectors, we have $m = n = p = 1$, and for all i, j,

$$T^i_j = \sum_k S^i_k W^k_j.$$

There, i is the index of the batch, k sums over the hidden dimension common to the input and the matrix, and j indexes the dimension of the new representations.

For dense layers operating on sequences, we have $m = 2, n = p = 1$, and for all i, l, j,

$$T^{il}_j = \sum_k S^{il}_k W^k_j.$$

There i, j, k play the same role as above, but the operation also has a "length" axis with l denoting the index in the sequence on which the operation is applied.

For convolutions, several implementations pass by folding/unfolding the images as explained by Bu et al. (2022). Essentially, for an input image $s_i \in \mathbb{R}^{H \times W \times C}$ convolved with a filter $w \in \mathbb{R}^{K \times K \times C \times C}$, the image is unfolded into $\sigma_i \in \mathbb{R}^{HW \times CK^2}$, and the weights into $\omega \in \mathbb{R}^{CK^2 \times C}$ such that the matrix product $\tau_i = \sigma_i \omega \in \mathbb{R}^{HW \times C}$ corresponds to the convolution at each output index (variable heights, width, channels, and kernel sizes can be treated with the same logic, see Bu et al. (2022)). The output image is obtained by folding τ_i into $t_i \in \mathbb{R}^{H \times W \times C}$. For a batch of examples, the core operation remains a tensor contraction

$$\tau^{il}_j = \sum_k \sigma^{il}_k \omega^k_j,$$

where i is still the index in the batch, while l is the spatial position after unfolding, k denotes the index of the receptive fields of the image and the filter concatenated along the channels of the inputs, and j denotes the output channel dimension.

Mini-batch gradient computation. During backpropagation, as explained in Appendix B.1, the gradients of the loss w.r.t. T are computed giving a multi-index array R sharing the same indexes as T. One verifies then that the adjoint $\ell(S, \cdot)^*$ of the tensor contraction, give a final sum of gradients as $\sum_{i=1}^{B} \nabla_W \ell(\theta; x_i) = G$ where

$$G^{k_1,\ldots,k_p}_{j_1,\ldots,j_n} = \sum_{i_1,\ldots,i_m} S^{i_1,\ldots,i_m}_{k_1,\ldots,k_p} R^{i_1,\ldots,i_m}_{j_1,\ldots,j_n}, \tag{10}$$

is computed as a tensor contraction over the free axes I of S and R.

Opportunities in vectorized intermediate representations. For dense layers operating on vectors the final operation (10) reads

$$G^k_j = \sum_i S^i_k R^i_j,$$

that can be written as a sum of rank-one vectors. As explained in 2.2 we can exploit this structure for factorable operations like the elementwise square.

Potential challenges in sequence-level operations. For dense layers operating on sequence of vectors, the final operation (10) reads

$$G^k_j = \sum_i \sum_l S^{i,l}_k R^{i,l}_j.$$

In other words, in this case, the per-sample gradients are not rank-one vectors and require a sum across the length of the sequence. We cannot then simply square S and R to obtain for example the sum of square gradients.

One may think that $\sum_i \sum_l (S^{i,l}_k)^2 (R^{i,l}_j)^2$ represent sum of square gradients per-sample per-token; however for architectures like transformers this is not a well-posed quantity since the computations of per-token losses do not define independent computational paths (the activation heads mix the contribution of each token). That sum instead represents a per-sample per-activation gradient.

Affine operations. Note that affine operations (like adding an offset) can essentially be cast in terms of tensor contractions between the offset b and some constant vector of ones. We can then make similar observations as in the generic tensor case: if the offset is applied on several "data parallel" axes (like both the batch and some length axis) then the gradient is obtained through several reductions. If there is only one axis, we can compute e.g. average square of gradients for offsets easily too.

Case of preprocessed weights. For some operations weights are preprocessed before being applied to the inputs. For example, weight matrices may be orthonormalized before being applied (Mhammedi et al., 2017). Operations on the inputs take then the form

$$\ell(s_i, W_k) \quad \text{for } W_k = p(V_k),$$

where V_k are actually the weights we are optimizing for. Such situations generally cover cases where the weights are not applied linearly on inputs, as it can be the case for the scaling of a normalization layer.

If the weights are preprocessed, following the perspective taken in Appendix B, the mini-batch gradient with respect to V_k read

$$\nabla \left(\oplus \circ h_{/\!/} \circ \ell_{/\!/} \circ \in \circ p \right) (V_k) = P^* \circ \oplus \circ L^*_{/\!/} \circ H^*_{/\!/} \circ \in 1,$$

where $P = \partial p(V_k)$ is the linearization of p on V_k.

To average a function ϕ of the gradients, we need to first apply the adjoint of the preprocessing on each individual gradients and then average. In other words, the resulting computations would be

$$\sum_{i=1}^{B} \phi(\nabla_{V_k} f(\theta; x_i)) = \oplus \circ \phi \circ P^*_{/\!/} \circ L^*_{/\!/} \circ H^*_{/\!/} \circ \in 1,$$

where $P^*_{/\!/}$ applies the adjoint P^* on all incoming individual gradients computed through $L^*_{/\!/} \circ H^*_{/\!/} \circ \in 1$. Injecting the non-linear operation ϕ requires then to modify the adjoint P^* to operate on individual gradients, and we may not use some simplifications like applying ϕ to the inputs s_i, r_i of the reduction as explained in Section 2.2.

Tied weights. Tied weights, i.e., weights that are applied at different operations, may still be treated by a jaxpr surgery, albeit at a potentially non-negligible cost.

Consider for example an RNN whose weight W is shared among the time-steps k. The resulting mini-batch gradient is then summed both along the batch axis and along the operations in the computational graph, and take the form $\sum_i \nabla_W f(\theta, x_i) = \sum_i \sum_k s_k^i (r_k^i)^\top$ for s_k^i the intermediate representations of x_i along the time steps k and r_k^i the gradients with respect to the output of each application of W.

When computing the mini-batch gradient, the sum over the batch is computed at each index and the sum over the time steps is accumulated during the backward pass. Namely, the gradient of the mini-batch is naturally computed as $\sum_k \sum_i s_k^i (r_k^i)^\top$.

Getting access to the individual gradients require then to remove the sum over the batch in the computational graph of the gradient to back-propagate the B individual gradients up to the first time step. At that point, the ϕ operation can be applied and a final sum over i can be performed.

Note that, if the RNN processes vectors, the intermediate gradients $s_k^i (r_k^i)^\top$ are rank-one vectors whose nature can be exploited. In particular, for any operations that factor over the rank-one structure such as taking the square, we can bookkeep the decomposition in rank-one vectors to potentially avoid some memory overload.

C SignSGD

C.1 The sign function and signal to noise ratio

Applying the sign function on a random variable X can dramatically change the signal to noise ratio (SNR) as defined by $\mathrm{SNR}_X \equiv |\mu_X|/\sigma_X$. The SNR of $\mathrm{sign}(X)$ is

$$\mathrm{SNR}_{\mathrm{sign}(X)} = \frac{|\mathbb{E}[\mathrm{sign}(X)]|}{\sqrt{\mathrm{Var}[\mathrm{sign}(X)]}} = \frac{|2p-1|}{2\sqrt{p(1-p)}}, \tag{11}$$

where p is the probability that $X > 0$.

Note that the ratio $\mathrm{SNR}_X/\mathrm{SNR}_{\mathrm{sign}(X)}$ depends heavily on the distribution — $\mathrm{SNR}_{\mathrm{sign}(X)}$ is sensitive to the details of tails, and imbalance like discrepancy between the median and mean. Regardless, it is informative to analyze some simple cases.

Consider the case where $X \sim \mathcal{N}(r, 1)$. We have $\mathrm{SNR}_X = r$. For $0 < r \ll 1$, we have:

$$p = \frac{1}{2} + \frac{r}{\sqrt{2\pi}} + O(r^2), \quad \mathrm{SNR}_{\mathrm{sign}(X)} = \frac{r}{\sqrt{2\pi}} + O(r^2). \tag{12}$$

Therefore the sign operation reduces the SNR in this limit. In general, for distributions whose PDF can be defined as differentiable functions $\rho(x-r)$ for some small parameter r, where $\rho(x)$ has median at 0, we have:

$$\mathrm{SNR}_{\mathrm{sign}(X)} = \rho(0)r + O(r^2). \tag{13}$$

If the distribution $\rho(x)$ has 0 mean as well, then the distribution $\rho(x-r)$ has mean r. The change in SNR due to the sign function can be written as:

$$\frac{\mathrm{SNR}_{\mathrm{sign}(X)}}{\mathrm{SNR}_X} = \frac{\rho(0)}{\sigma_X} + O(r^2) \tag{14}$$

This suggests that for narrower distributions (in the sense of $\rho(0)/\sigma_X$), sign has a worse effect on SNR than for broader distributions — which is consistent with general intuitions and observations about the sign function in optimization.

In the opposite limit where $r \gg 1$ we have:

$$p = 1 - \frac{\exp(-r^2/2)}{r}(1 + O(r^{-2})), \quad \mathrm{SNR}_{\mathrm{sign}(X)} = \sqrt{r}\exp(r^2/4). \tag{15}$$

Here SNR is improved by sign if $r > \sqrt{2\log(r)}$.

Our analysis suggests that application of sign can reduce SNR when SNR is already low, and increase it when SNR is high, which means one must be careful about where to apply it in gradient processing. If gradients are near-Gaussian, it suggests that it is better to apply sign after as much averaging as possible to reduce the SNR of the random object that is pasing through sign. This is consistent with the observation that SignEMA was the best performing algorithm, and MicroSignSGD was the worst.

D Adam variant details

D.1 Argument for the square root scaling rule

In this section we present a basic argument for the proposed $\eta \propto \sqrt{B}$ scaling heuristic that is often used for Adam. We start with the scaling rule $\eta \propto B$ for SGD. The basic setup is as follows: we consider a series of gradients g_t presented in a training loop. (We will focus on the case of a single parameter at the time but the argument generalizes immediately to multiple parameters.) We assume that the gradients are sampled i.i.d. from a distribution with mean μ and variance σ^2.

Consider taking T steps of SGD with batch size B with learning rate η. The mean and the variance of the total update $u_T(\eta, B)$ is given by:

$$\mathbb{E}[u_T(\eta, B)] = T\eta\mu, \ \mathrm{Var}[u_T(\eta, B)] = T\eta^2 \frac{\sigma^2}{B}. \tag{16}$$

We can then ask the following question: is there some learning rate scaling rule such that the first two moments are identical for a fixed number of samples $K \equiv TB$? That is, we want:

$$\mathbb{E}[u_{K/B}(\eta(B), B)] = m(K), \ \mathrm{Var}[u_{K/B}(\eta(B), B)] = V(K). \tag{17}$$

The answer is linear scaling of learning rate with batch size. Selecting $\eta = aB$, we have:

$$\mathbb{E}[u_{K/B}(aB, B)] = aK\mu, \ \mathrm{Var}[u_{K/B}(aB, B)] = a^2 K\sigma^2. \tag{18}$$

If we think of modeling SGD as a random stochastic process, this ensures that we have two processes with equal statistics in a sample-to-sample comparison.

We can now provide a similar analysis for Adam. We ignore momentum for now and model the Adam update distribution as follows: we assume that the gradient distribution is stationary, and the preconditioner well-approximates the second moment of this distribution. That is, the preconditioner magnitude is $\mu^2 + \sigma^2/B$. Note that the batch size appears in this calculation.

Under these assumptions, the modified gradients $\tilde{g}_t$ have central moments

$$\mathbb{E}[\tilde{g}_t] = \frac{\mu}{\sqrt{\mu^2 + \sigma^2/B}}, \ \mathrm{Var}[\tilde{g}_t] = \frac{\sigma^2}{\mu^2 + \sigma^2/B}. \tag{19}$$

The denominators come from the assumption that the preconditioner is well-captured by the second moment $\mathbb{E}[g_t^2]$.

The update central moments are

$$\mathbb{E}[u_T(\eta, B)] = T\eta \frac{\mu}{\sqrt{\mu^2 + \sigma^2/B}}, \ \mathrm{Var}[u_T(\eta, B)] = T\eta^2 \frac{\sigma^2}{B\mu^2 + \sigma^2}. \tag{20}$$

We can ask the same question about finding a learning rate scaling rule that causes moments to match for equal number of samples. In general this would involve a complicated rule depending on the gradient statistics; however this simplifies if $\mu^2 \ll \sigma^2/B$. In this case we have:

$$\mathbb{E}[u_T(\eta, B)] \approx T\eta\sqrt{B}\frac{\mu}{\sigma}, \ \mathrm{Var}[u_T(\eta, B)] \approx T\eta^2. \tag{21}$$

Here the *square root scaling rule* $\eta \propto a\sqrt{B}$ gets us

$$\mathbb{E}[u_{K/B}(aB, B)] = aK\frac{\mu}{\sigma}, \ \mathrm{Var}[u_{K/B}(aB, B)] = a^2 K, \tag{22}$$

independent of B. Interestingly this is a normalized version of the moments for SGD. The other limit is $\mu^2 \gg \sigma^2/B$. Then we have

$$\mathbb{E}[u_T(\eta, B)] \approx T\eta, \ \mathrm{Var}[u_T(\eta, B)] \approx T\eta^2 \frac{\sigma^2}{B\mu^2}. \tag{23}$$

This leads to a linear scaling rule $\eta \propto aB$:

$$\mathbb{E}[u_{K/B}(aB, B)] = aK, \ \mathrm{Var}[u_{K/B}(aB, B)] = a^2 K \frac{\sigma^2}{\mu^2}, \tag{24}$$

unlike the small μ^2 limit.

The square root scaling rule can be made more formal in various stochastic differential equation (SDE) limits of training dynamics. Currently such limits have been formally derived for SignSGD and not Adam, but the same square root learning rule applies in both cases. In such a limit one can derive an SDE in continuous time whose dynamics can be mapped onto the discrete stochastic dynamics such that averages of observables match between the two descriptions, with error shrinking with some small parameter (learning rate in the approach of Compagnoni et al. (2024), inverse dimension in Xiao et al. (2025)).

In our training setup, we can see that the square root learning rule induces similar learning curves in terms of number of samples over some range of B (Figure 9). We first found the best learning rate $\eta^*_{B_0}$ for $B_0 = 64$. We then use the square root scaling rule to generate learning rates for other batch sizes with $0.5\eta^*_{B_0}$ (Figure 9, left), and $\eta^*_{B_0}$ (Figure 9, right). We see that there is a crossover from small $B = 8$ to the larger batch sizes where the learning curve is quite different. From $B = 16$ to $B = 128$ the intermediate to late time learning curves are similar, while for larger batch sizes (and therefore learning rates) the curves become non-universal and eventually diverge. Smaller learning rates show better agreement as predicted by the theory; the heuristic (and the SDE equivalents) break down if the gradient distribution changes over a small number of steps. This can happen because of effects like feature learning, or even more simply by making significant progress towards the objective.

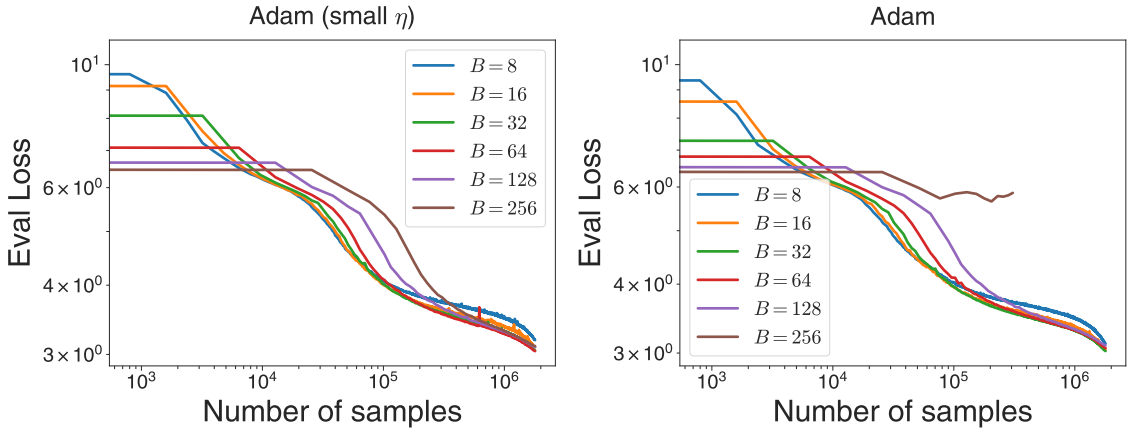

Figure 9: Square root learning rate scaling provides similar learning curves for a wide range of B in transformer language model training. Small batch size ($B = 8$) shows crossover behavior. Large batch size leaves the universal regime. Learning rate at $B = 64$ was set to half the optimal rate in the left plot, and at the optimal rate in the right plot. Smaller learning rates show better universal behavior.

The observation that the square root rule works in practice suggests that $\mu^2 \gg \sigma^2/B$. However, our experiments suggest that $\mu^2 \sim \sigma^2/B$ across various batch sizes, preserving the square root scaling rule without a variance-dominated preconditioner. Understanding this mystery fully is left as an open question.

D.2 Stabilizing variance-dominated Adam variants with gradient clipping

We found that gradient clipping could stabilize both MicroAdam and MicroAdamVar with appropriate choice of clipping thresholds. For $B = 64$ we swept clipping thresholds by factors of 10 and found that a

threshold of 10^{-2} removed most training spikes and led to the best eval metrics at the end of training. We then repeated the batch size sweep experiments from Figure 4 to confirm that the learning curves across batch sizes were similar and that training spikes were removed at most scales (Figure 10). However the clipping made the correspondence between the different batch sizes worse, and the final eval losses were still worse than Adam (3.10 for MicroAdam, 3.20 for MicroAdamVar, versus 3.06 for Adam).

Indeed, the fact that MicroAdamVar was the worst algorithm even after fixing stability issues further suggests that emphasizing the variance information in the Adam preconditioner is generally detrimental for training.

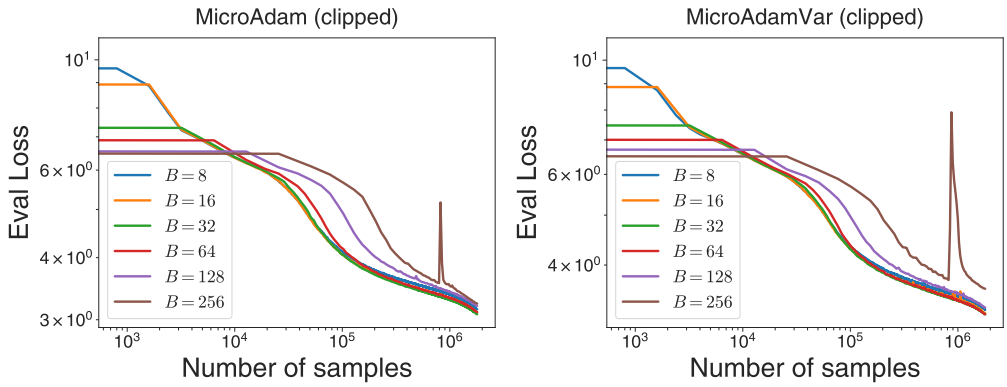

Figure 10: Training spikes in MicroAdam (left, best eval 3.10) and MicroAdamVar (right, best eval 3.20) can be mitigated with strong gradient clipping with threshold 10^{-2}. However these methods still have worse final eval loss compared to Adam (3.06). This provides further evidence that emphasizing the variance term of the Adam preconditioner leads to worse training outcomes.

D.3 Adam μ^2 and σ^2 statistics

In order to further understand the consistency of the square root learning rule with our observation that μ^2 seems to dominate the preconditioner at early times, we measured μ^2 and σ^2 using the estimators from 8 for models trained with varying batch size. Plotting the ratio of $\hat{\mu}^2$ to $\hat{\sigma}^2_{eff}$ (σ^2/B for regular Adam, σ^2 for MicroAdam) we see that at all batch sizes we studied, μ^2 dominates at early times for Adam (Figure 7, left). The ratios and their dynamics were somewhat consistent even for Adam, where $\hat{\sigma}^2_{eff}$ has explicit dependence on B.

To further quantify this, we took the median ratio for Adam for each layer at each batch size, and found that the range of values was consistent across batch sizes (Figure 7, right). This is consistent with the idea that indeed the statistics of μ^2 must have a nontrivial relationship with σ^2/B to dominate the dynamics yet induce the square root learning rate heuristic. More study across a broader set of batch sizes and architectures is needed to firmly establish this phenomenon.

E Experimental details

We train a standard decoder-only transformer from the Nanodo codebase (Liu et al., 2024).

E.1 Dataset and tokenizer

We train on the C4 dataset (Raffel et al., 2020), using use the SentencePiece tokenizer from Raffel et al. (2020).

E.2 Architecture

The architecture consists in N identical transformer blocks. Each block consists in a multi-head causal dot-product self-attention layer followed by a MLP layer. Layer normalization is applied to both the input of the attention and the input of the MLP layer. The MLP layer uses a gelu activation function. The model employs a learned positional embedding. The model uses an independent layer to map back to the vocabulary space. So in plain code, the architecture reads

```
def Transformer(V, L, D, H, F)(x_L):
    x_LxD = Embedder(D)(x_L)
    y_LxD = PositionalEmbedder(D)([0, ...,L])
    x_LxD = x_LxD + y_LxD
    for i in range(N):
        y_LxD = LayerNorm()(x_LxD)
        y_LxD = MultiHeadAttention(H)(y_LxD)
        x_LxD = y_LxD + x_LxD
        y_LxD = LayerNorm()(x_LxD)
        y_LxD = MLP(F)(y_LxD)
        x_LxD = y_LxD + x_LxD
    x_LxD = LayerNorm()(x_LxD)
    x_LxV = Linear(V)(x_LxV)
    return x_LxV

def MLP(F)(y_LxD):
    y_LxF = Linear(F)(y_LxV)
    y_LxF = gelu(y_LxF)
    y_LxD = Linear(D)(y_LxF)
    return y_LxD
```

The key parameters are given below. The number of the backbone parameters (i.e., ignoring embedding and output layers) is 151 million.

1. the size of the vocabulary V, fixed by the C4 dataset (i.e., $V = 32101$)
2. the length of the sequences $L = 2024$
3. the number of blocks, $N = 12$
4. the hidden dimension $D = 64H$
5. the number of heads in the attention layer $H = 16$
6. the hidden dimension in the MLP $F = 4D$

For the memory comparisons in Figure 2, we used the same architecture but with $N = 24$, $H = 32$.

E.3 Optimization generic details

Below, we present generic hyperparameters' settings. Specific setups are detailed in

Number of steps. In all experiments, the number of steps is fixed as (following Nanodo's recipe (Liu et al., 2024) which follows Chinchilla's (Hoffmann et al., 2022) scaling factor $c = 20$)

$$K = \lfloor cP/T \rfloor,$$

where $P = 12ND^2 + VD$ is the number of parameters (ignoring the final head), and $T = LB$ is the number of tokens per batch.

Batch size. If not specified the batch size is fixed at

$$B = 64.$$

For the memory comparison in Figure 2, we used $B = 256$.

Schedule and peak learning rate. We use a cosine decay schedule with warmup starting at 0, ending at 0. We use 1000 steps for the warmup. If not specified, the learning rate is set to the base learning rate of Nanodo, i.e.

$$\eta = 2/1024.$$

Weight decay. All implementations use weight decay, that is all implementations of Adam and all the SignSGD implementations. The weight decay is fixed at

$$\omega = 8/K$$

for K the number of steps presented above. This weight decay is not multiplied by the varying learning rate.

E.4 Experimental specific details.

SignSGD. For SignEMA, we used an EMA with decaying parameter $\beta = 0.9$. The learning rates were searched around the base learning rate η defined above in a grid $\{10^{0.25i}\eta \mid -6 \leq i \leq 1\}$. The best learning rates found were $3.47 \cdot 10^{-3}$ for MicroSignSGD, $1.10 \cdot 10^{-3}$ for SignSGD, $3.47 \cdot 10^{-4}$ for SignEMA.

Adam and MicroAdam. In all experiments, we ran Adam and its variants with EMA parameters $\beta_1 = 0.9$ and $\beta_2 = 0.95$ for respectively the first and second moments running estimators

For the scaling experiments (Figure 4, 4), as the batch size varies, the total number of steps vary according to the rule $K = \lfloor cP/T \rfloor$ with $T = LB$. The number of warmup steps is multiplied by $B/1024$ as it had been selected for batch size of 1024. As the base learning rate was chosen for a batch size $B = 1024$, the learning rate is adjusted as $\eta_\gamma = \gamma\eta$ with a factor $\gamma = B/1024$ for all variants except Adam that uses $\gamma = \sqrt{B/1024}$. This same factor is used to adjust the weight decay as $\omega_B = \omega_\gamma$.

Although not reported here, we tried varying β_2 for MicroAdam without success.

MicroAdamVar algorithm We implemented MicroAdamVar by using two EMA estimators ν_{adam} and ν_{micro}, and combining them in accordance with Equation 8. We then carried out a similar bath size scaling experiment to Figure 4.

F Detailed pseudocode

For completeness we detail the pseudocode for the MicroAdam family of algorithms in Algorithm 1.

Disclosures

A Large Language Model (LLM) has been used to reformat the paper before the submission and remove part of the content that the authors considered not mature enough for review. No LLM has been used to generate ideas, content, nor code.

Algorithm 1 MicroAdam algorithms

Inputs
Initial parameters θ_0,
Exponential Moving Average (EMA) decay parameters β_1, β_2
Learning rate schedule $(\eta_k)_{k\geq 0}$
Small ε to avoid division by 0
Preconditioner computation MicroAdamX
Initialization
EMA of gradients $\mu_0 = 0$
EMA of preconditioner $\nu_0 = 0$
for $t \in \{1, ..., T\}$ **do**
  Fetch samples $x_1^{(t)}, \ldots, x_B^{(t)}$
  Compute average gradient $\bar{g}_t = \frac{1}{B}\sum_{i=1}^{B} \nabla f(\theta_{t-1}; x_i^{(t)})$
  Update EMA of gradients $\mu_t = \beta_1 \mu_{t-1} + (1-\beta_1)\bar{g}_t$
  if MicroAdamX = MicroAdam **then**
    Compute preconditioner as average element-wise square
    $\nu = \frac{1}{B}\sum_{i=1}^{B} g_i^2$ for $g_i = \nabla f(\theta_k; x_i^{(t)})$
  else if MicroAdamX = MicroAdamVar **then**
    Compute preconditioner as element-wise estimate variance
    $\nu = \frac{B}{1-B}\left(\frac{1}{B}\sum_{i=1}^{B} g_i^2 - \left(\frac{1}{B}\sum_{i=1}^{B} g_i\right)^2\right)$ for $g_i = \nabla f(\theta_k; x_i^{(t)})$
  else if MicroAdamX = MicroAdamMSQ **then**
    Compute preconditioner as element-wise estimate variance
    $\nu = \frac{B}{1-B}\left(\left(\frac{1}{B}\sum_{i=1}^{B} g_i\right)^2 - \frac{1}{B}\sum_{i=1}^{B} g_i^2\right)$ for $g_i = \nabla f(\theta_k; x_i^{(t)})$
  end if
  Update EMA of preconditioner $\nu_t = \beta_2 \nu_{t-1} + (1-\beta_2)\nu$
  Apply bias corrections on both EMAs: $\hat{\mu}_t = \mu_t/(1-\beta_1^t)$, $\hat{\nu}_t = \nu_t/(1-\beta_1^t)$
  if MicroAdamX = MicroAdamMSQ **then**
    $\hat{\nu}_t \leftarrow \max(0, \hat{\nu}_t)$
  end if
  Define update direction $u_t = \hat{\mu}_t/\sqrt{\varepsilon + \hat{\nu}_t}$
  Apply update $\theta_t = \theta_{t-1} - \eta_t u_t$
end for