ADAM-MINI: USE FEWER LEARNING RATES TO GAIN MORE

Anonymous authors
Paper under double-blind review

ABSTRACT

We propose Adam-mini, an optimizer that achieves on-par or better performance than AdamW with 50% less memory footprint. Adam-mini reduces memory by cutting down the learning rate resources in Adam (i.e., $1/\sqrt{v}$). By delving into the Hessian structure of neural nets, we find Adam's v might not function at its full potential as effectively as we expected. We find that $\geq 99.9\%$ of these learning rates in v could be *harmlessly* removed if we (1) carefully partition the parameters into blocks following our proposed principle on Hessian structure; (2) assign a single but good learning rate to each parameter block. We then provide one simple way to find good learning rates and propose Adam-mini. Empirically, we verify that Adam-mini performs on par or better than AdamW on various language models sized from 39M to 13B for pre-training, supervised fine-tuning, and RLHF. The reduced memory footprint of Adam-mini also alleviates communication overheads among GPUs, thereby increasing throughput. For instance, Adam-mini achieves 49.6% higher throughput than AdamW when pre-training Llama 2-7B on $2\times$ A800-80GB GPUs, which saves 33% wall-clock time for pre-training.

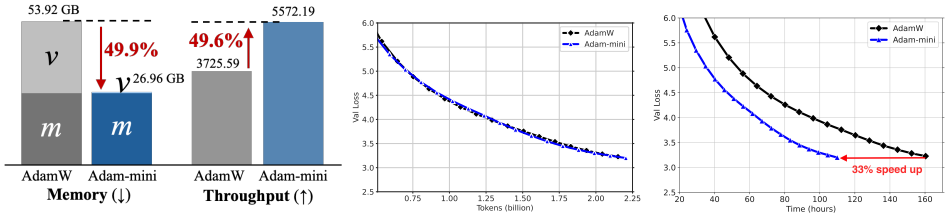
1 INTRODUCTION

Adam (Kingma & Ba, 2014) has become the de-facto optimizer for training large language models (LLMs) (e.g., (Vaswani et al., 2017; Achiam et al., 2023; Touvron et al., 2023; Team et al., 2023)). Despite its superior performance, Adam is expensive to use. Specifically, Adam requires the memory for its optimizer states: the first-order momentum m, and the second-order momentum v. These in total take at least $2\times$ the memory of the model size[1]. This memory consumption has become a major burden in LLM training. For instance, to train a 7B model, Adam alone requires about 56 GB for m and v, and with the gradients included, a total of 86 GB is needed. This is expensive even for cutting-edge graphics cards (e.g., A100-80GB). To support training, CPU-offload and optimizer state sharding (Rajbhandari et al., 2020) must be used in practice, which unfortunately increases the latency and slows down the training (Rajbhandari et al., 2021).

It is intriguing to design effective optimizers that require less memory. **First**, it lowers the threshold of training LLMs and encourages participation from more diverse researchers, especially those with limited GPU resources. **Second**, it requires fewer GPUs to train a model with a desired size, leading to substantial savings in both cost and energy. **Third**, it can ease the burden of CPU offloading and model sharding, which in turn, can enhance the throughput and accelerate the training process.

It is challenging to modify Adam without sacrificing its performance. One primary reason is that we still lack understanding of the role of Adam's m and v (Zhang et al., 2020; Kunstner et al., 2023). It remains uncertain which components in Adam are indispensable for superior performance, and which components could be re-designed or improved. One notable attempt is Adafactor (Shazeer & Stern, 2018), which cuts down memory by low-rank factorization on v. However, we find that Adafactor is not easy to tune and often performs worse than Adam (see evidence in (Luo et al., 2023) and Section 3.4). One possible reason is that the current v in Adam is crucial and cannot be simplified. This is possible as most existing Adam variants that attempt to modify v to varying extents have been reported to perform worse than Adam (Orabona, 2020). Another possible reason is that there is potential to cut down v, but Adafactor does not use the most suitable way: matrix factorization is a generic approach that could be applied broadly, but it does not leverage much problem-specific structure, thus it does not work well on specific neural-net tasks.

[1]We restate the update rules of Adam and AdamW in Appendix D.1.



(a) Memory (↓) and Throughput (↑) (b) Loss v.s. iteration (c) Loss v.s. time

Figure 1: Results for Llama 2-7B pre-training. (a) Adam-mini takes less memory and can reach higher throughput (# tokens per second). The throughput is tested on 2× A800-80GB GPUs. (b, c) Adam-mini performs on-par with AdamW, but takes 33% less time to process the same # tokens.

In this work, we find it is possible to significantly reduce the usage of v. Currently, Adam assigns an individual learning rate for each parameter, i.e., i-th parameter receives learning rate $\frac{\eta}{\sqrt{v_i}}$, where v_i is the i-th component of v. For a billion-parameter model, Adam requires billions of learning rates. We argue that it is possible to achieve on-par or better performance with much fewer learning rates. We first recall a classical result that the Hessian of neural nets is near-block-diagonal with several dense principle sub-blocks (Collobert, 2004). We then find that, for each of these dense sub-blocks, there exists a single high-quality learning rate that outperforms Adam, provided that we have enough resources to search it out. Since the number of dense sub-blocks is much fewer than the number of parameters, our findings imply that it is possible to achieve good performance with much fewer learning rates. The remaining question is how to find them efficiently.

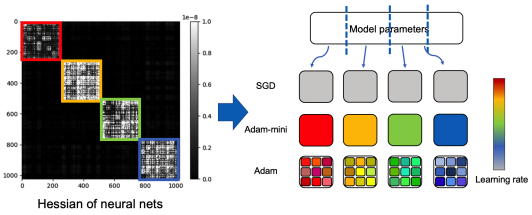


Figure 2: An illustration of Adam-mini. Adam-mini assigns learning rates (lrs) by Hessian structure. It uses more lrs than SGD but fewer than Adam.

We then propose a cheap and simple way to find good learning rates that are sufficient to perform on-par or better than Adam. We introduce the proposed design principle here: we first partition the gradient vector into B sub-vectors according to the dense Hessian sub-blocks, and call it g_b for $b \in \{1, \cdots, B\}$. For each g_b, we calculate the quantity below.

$$\boxed{v_b = (1-\beta_2) * \texttt{mean}(g_b \odot g_b) + \beta_2 * v_b, \quad b = 1, \cdots B.}$$

We then use $\eta/\sqrt{v_b}$ as the learning rate for the parameters in the block associated with g_b. Such design changes almost all Adam's v to a negligible amount of scalars and thus reduces the memory. We call the corresponding method Adam-mini. We provide a simple illustration in Figure 2 and relegate the complete form later in **Algorithm 2**. We summarize our main contribution as follows.

- **New optimizer.** We propose a new optimizer called Adam-mini. First, Adam-mini partitions the model parameters based on the principle we established upon the Hessian structure. Then, it chooses a single learning rate for each block using the average of Adam's v in that block. Adam-mini has the following advantages.
  - **Lightweightness:** By design, Adam-mini largely reduces the number of learning rates used in Adam. For mainstream LLMs, Adam-mini could cut down $\geq 99.9\%$ proportion of Adam's v, which saves 50% of the memory cost of Adam.
  - **Effectiveness:** Despite the memory cut down, we empirically verify that Adam-mini performs on par or even better than AdamW (Loshchilov & Hutter, 2017) on various language models sized from 39M to 13B, including pre-training, supervised fine-tuning (SFT), and reinforcement learning from human feedback (RLHF). Adam-mini also performs similarly to Adam on non-LLM tasks such as training diffusion models, vision models, and graph neural networks.
  - **Efficiency:** Adam-mini can reach higher throughput than AdamW. We observe that Adam-mini reaches 49.6% higher throughput of AdamW when pre-training Llama 2-7B on $2\times$ A800-80GB, which saves 33.1% wall-clock time for pre-training. The efficiency comes from two factors. First, Adam-mini does not introduce extra computation in per-step updates. Second, the memory cut-down allows larger batch sizes per GPU, and at the same time, it eases the burden of communication among GPUs, which is usually a major overhead.
- **Generic partition principle.** A key component in Adam-mini is the strategy for parameter partition. We propose to partition parameters based on the smallest dense sub-block in Hessian. This principle can apply to generic problems with block diagonal Hessian: we find that more learning rates do not necessarily bring extra gain for these problems. In particular, for the problem associated with each dense sub-block, a single (but good) learning rate suffices to bring better performance.

- **Hessian structure and partition principle of Transformers.** We empirically apply the above principle to Transformers. We find that Transformer Hessian's smallest dense blocks are: (1) `query`, `key` by heads; (3) `value`, `attn.proj` and `mlp` by output neuron. We emphasize that our Hessian-based partition principle is crucial for good performance, as naive or default partitions (e.g. partitioning by layers) would cause training instability on LLMs.

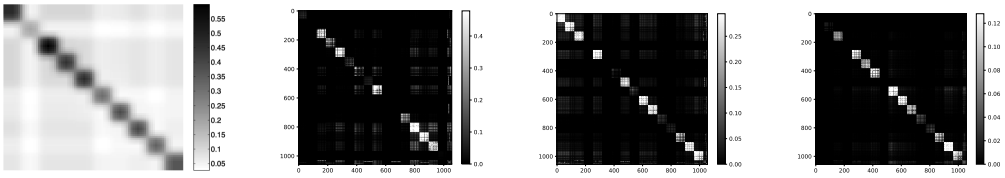
2 METHOD

2.1 MOTIVATIONS AND OBSERVATIONS

Now we discuss our observations that motivate the design of Adam-mini. [2] We start by investigating the role of Adam's v and explore possibilities for improvement. In Adam, v provides an individual learning rate for each parameter, i.e., i-th parameter receives the learning rate $\frac{\eta}{\sqrt{v_i}}$, where v_i is the i-th component of v. Very recently, Zhang et al. (2024) pointed out that such design is crucial for modern architectures such as Transformers. This is because these models often exhibit Hessian-block heterogeneity, i.e., the Hessian of different parameter blocks have dramatically different eigenvalue distributions (We restate their findings in Appendix D.2). This phenomenon suggests that different parameter blocks need different learning rates. This can be provided by Adam's v.

The findings in (Zhang et al., 2024) suggest that it is necessary to use a different learning rate for *each block*. Nonetheless, Adam does much more than that: it assigns an individual learning rate not just for each block, but for *each parameter*. Note that the number of parameters (could be billions) is much larger than the number of blocks (usually hundreds). This begs the question:

(Q1) Is it necessary to use a customized learning rate for each parameter?
If not, how much can we save?



(a) Exact Hessian of a MLP (Collobert, 2004) (b) Hessian of a MLP at 1% step (c) Hessian of a MLP at 50% step (d) Hessian of a MLP at 100% step

Figure 3: The near-block-diagonal Hessian structure of neural nets. (a) is the Hessian of an MLP after 1 training step reported in (Collobert, 2004). (b,c,d): the Hessians of an MLP that we observe along the training. We find the near-block-diagonal structure maintains throughout training.

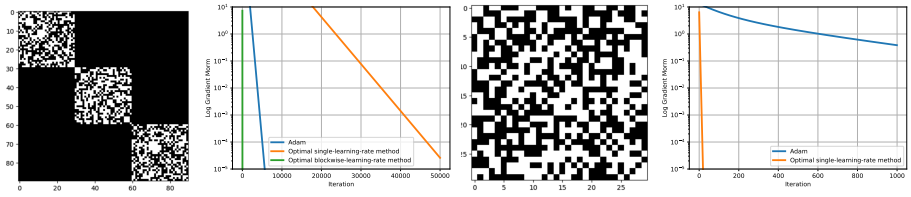
To answer **(Q1)**, we delve into the Hessian structures of neural networks. First, we recall an important (but often overlooked) result: the Hessian of neural nets is near-block-diagonal. This is an old result that has been reported for at least two decades; see, e.g., (Collobert, 2004, Section 7). The authors also provide theoretical justification. Consider a standard supervised learning problem: minimizing $\ell(f(\theta, x), y)$ where $\ell(\cdot,\cdot)$ is the Cross-Entropy (CE) loss, $f(\theta, x) = \sum_{i=1}^{n} v_i \phi(w_i^\top x)$ is an 1-hidden-layer neural network with input $x \in \mathbb{R}^d$, weight $w_i \in \mathbb{R}^d$, $v_i \in \mathbb{R}$, and label $y \in \{0, 1\}$, then the off-diagonal-block Hessian elements would contain

$$\frac{\partial^2 \ell(f(\theta, x), y)}{\partial w_i \partial w_j} = p(x)\,(1 - p(x)) v_i v_j \phi'\left(w_i^\top x\right) \phi'\left(w_j^\top x\right) x x^\top \quad \text{for } i \neq j, \tag{1}$$

where $p(x) = 1/(1 + \exp(-y f(\theta, x)))$ and $\phi'(\cdot)$ is the derivative of $\phi(\cdot)$. Since the training objective is to maximize $p(x)$, the term $p(x)\,(1 - p(x))$ will quickly shrink to zero. This term will push the Hessian to near-block-diagonal structure where each block corresponds to one output neuron. The authors report that this can happen just after 1 training step, as restated in Figure 3 (a). We also numerically reproduce this result on a small MLP in Figure 3 (b,c,d), and show that the near-block-diagonal Hessian structure maintains along training.

Case study I: random quadratic problems. With the above observation in mind, we now explore **(Q1)** on generic optimization problems with block-diagonal Hessian. We consider the random quadratic minimization problem $\min_w \frac{1}{2} w^\top H w$ where the Hessian H is a random positive definite (PD) matrix and is visualized in Figure 4 (a). We compare the coordinate-wise learning-rate method,

[2] All experimental details in Section 2 are shown in Appendix E.2.



(a) Hessian matrix (b) Total loss (c) Dense sub-block (d) Sub-problem loss

Figure 4: **(a):** The Hessian of a three-block random quadratic problem. **(b):** Training curves for the problem associated with the full Hessian in (a). The optimal single (blockwise) learning rate is chosen based on the full (blockwise) Hessian in (a). **(c):** The 1st dense Hessian sub-blocks in (a). **(d):** Training curves for the new problem associated with the Hessian in (c).

i.e., Adam, with the single-learning-rate method, i.e., gradient descent (GD). We choose quadratic minimization because the optimal learning rate has a close form. We have the following findings.

- **(1):** as shown in Figure 4 (a) and (b), Adam outperforms the optimal single-learning-rate method. This is expected since Adam deploys different learning rates to different parameters.
- **(2):** as shown in Figure 4 (c) and (d), we consider a new problem whose Hessian is a dense sub-block of (a). We consider the optimal single learning-rate method for this new problem and find it outperforms Adam, even though Adam assigns much more learning rates. Similar phenomena apply to all the three sub-blocks of (a).
- **(3):** If we collect these optimal learning rates in **(2)** and apply them to a "blockwise" version of GD, it would be faster than Adam on the original problem (the green line in Figure 4 (b)).

In summary, for generic problems with block-diagonal Hessian, we find that more learning rates do not necessarily bring extra gain. In particular, **for each dense sub-block, a single (but good) learning rate suffices to bring better performance** than using tens or hundreds more.

More discussions on case study I. Why would this happen? We provide one possible explanation from a linear algebra perspective. Adam can be viewed as a diagonal preconditioned method , i.e., at the t-th step:

$$w_{t+1} = w_t - \eta_t D_t m_t, \tag{2}$$

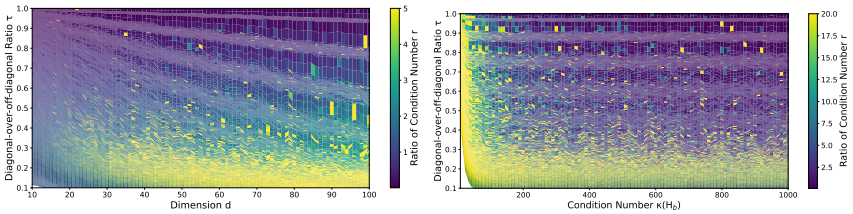
where $D_t = \mathrm{Diag}(1/\sqrt{v_t})$ is a diagonal matrix, m_t is the 1st-order momentum, w_t and η_t are model parameters and learning rate. However, Adam may not be an optimal preconditioner and thus cannot effectively reduce the condition number of the dense sub-matrix. In the field of optimization, the effectiveness of a diagonal preconditioner D is often measured by "how much is $\kappa(DH)$ reduced over $\kappa(H)$", where H usually refers to the Hessian matrix and $\kappa(\cdot)$ is the condition number (smaller is better). Unfortunately, there is no guarantee of $\kappa(DH) \leq \kappa(H)$ and this inequality often requires strict assumptions on both D and H. For instance, $\kappa(DH)$ would be small if H is close to diagonal and D is a cleverly designed compressor of H (Forsythe & Straus, 1955; Young, 1954; Sun & Ye, 2021; Qu et al., 2022).

Here, we numerically explore the effectiveness of Adam's preconditioner within each dense Hessian sub-block. We generate a random dense PD matrix $H_b \in \mathbb{R}^{d\times d}$ and use it as a proxy for the dense Hessian sub-block of neural nets in Figure 3. We define $D_{\text{Adam}} = \mathrm{Diag}(1/\sqrt{v})$, where $v = g \odot g$, $g = H_b x \in \mathbb{R}^d$, and each entry $x_i \sim \mathcal{N}(0, 1/\sqrt{d})$ follows Xavier initialization. We explore the interplay between the following two metrics:

$$\tau = \frac{\sum_i |H_{b,i,i}|}{\sum_{i,j} |H_{b,i,j}|}, \quad r = \frac{\kappa(D_{\text{Adam}} H_b)}{\kappa(H_b)}, \tag{3}$$

where $\tau \in [0, 1]$ is the "diagonal-over-off-diagonal ratio", and we use it to measure how dense H_b is (H_b is pure diagonal when $\tau = 1$). $r \geq 0$ measures the effectiveness of Adam's preconditioner D_{Adam} when operating on the Hessian-block H_b (the smaller the better). We investigate the change of r when changing the structure of H_b, including changing τ, dimension d, and also $\kappa(H_b)$. We emphasize that for a fixed d or $\kappa(H_b)$, we change τ by only rotating the eigenvectors, but not changing the eigenvalues of H_b. This ensures τ is the only changing factor in the experiments.

We summarize the key findings in Figure 5: for H_b with most dimension d and $\kappa(H_b)$, r decreases as $\tau \to 1$. That is, D_{Adam} is effective when H_b is close to diagonal, and $D_{\textbf{Adam}}$ **is not so effective when** H_b **is dense.** This aligns with the convergence rates in Figure 4. It is intriguing to provide a lower bound on $\kappa(D_{\text{Adam}} H_b)$ to ground the observation in Figure 5, and we are not aware of any existing lower bound of this kind. Note that it is rather difficult to characterize $\kappa(D_{\text{Adam}} H)$, partially because the extreme eigenvalues are neither sub-additive nor sub-multiplicative (Kittaneh, 2006). We leave it as an important but challenging future direction. **To summarize, for the dense Hessian-blocks, it is possible to outperform Adam with only one good learning rate.**



(a) r v.s. dimension d (b) r v.s. dimension $\kappa(H_b)$

Figure 5: The effectiveness of Adam's preconditioner D_{Adam} on different matrix structures of H_b. (a): for most dimension d, r is large when τ is small (r and τ are defined in Eq. (3)). This indicates that Adam might not be so effective when H_b is dense. We fix $\kappa(H_b) = 500$ here. (b): We use the same setups as (a), except that we fix the dimension $d = 50$ and change the x-axis to $\kappa(H_b)$.

Case study II: Transformers. The above analysis suggests there is room to cut down the number of learning rates. We also observe similar phenomena in Transformers. We consider a 4-layer Transformer and under the PyTorch default partition, and we randomly choose one parameter block as the "left-out" block and change the coordinate-wise learning rate to a single-learning rate counter-part. We use Adam for the rest of the blocks. We grid-search the learning rate for the left-out block and apply the cosine decay schedule like the rest of the blocks. We report the best result and call this method "Adam (leave-one-out)". Figure 6 shows that Adam (leave-one-out) can achieve similar or better performance than Adam. A similar phenomenon is also observed when we randomly leave out up to three blocks and search three learning rates. We do not explore the possibility of leaving more blocks out since the cost of grid search grows exponentially.

To summarize this section, we find that it is possible to reach similar or better performance with much fewer learning rates than Adam. The remaining issue is how to find them without grid-search. In the next part, we propose a simple and effective method called Adam-mini, which could bring comparable or even better performance than Adam, but with 99.9% fewer learning rates.

2.2 Proposed Method: Adam-mini

We now introduce Adam-mini. We will first state the **"general principled form"** of Adam-mini and then introduce the **"the realization"** of Adam-mini on specific architectures. In this section, we present the general form of Adam-mini in **Algorithm 1**. Following this general principled form, Adam-mini will have different realizations on different architectures, and the concrete example on Transformers is shown in Appendix B. As shown in **Algorithm 1**, Adam-mini contains two steps.

Algorithm 1 Adam-mini (General form)

Input weight-decay coefficient λ and current step t
Partition params into `param_blocks` by **Principle 1** in Section 2.3
for `param` in `param_blocks` **do**
  $\texttt{g} = \texttt{param.grad}$
  $\texttt{param} = \texttt{param} - \eta_t * \lambda * \texttt{param}$
  $\texttt{m} = (1-\beta_1) * \texttt{g} + \beta_1 * \texttt{m}$
  $\hat{\texttt{m}} = \frac{\texttt{m}}{1-\beta_1^t}$
  $\texttt{v} = (1-\beta_2) * \texttt{mean}(\texttt{g} \odot \texttt{g}) + \beta_2 * \texttt{v}$
  $\hat{\texttt{v}} = \frac{\texttt{v}}{1-\beta_2^t}$
  $\texttt{param} = \texttt{param} - \eta_t * \frac{\hat{\texttt{m}}}{\sqrt{\hat{\texttt{v}}}+\epsilon}$
end for

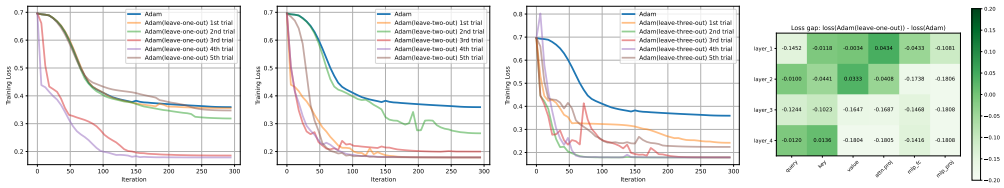
Step 1 Partition the model parameters into blocks by Hessian structure. We discuss **Principle 1** later in Section 2.3. For different architectures, the principle will be realized in different forms; see **Algorithm 3**: "Partition for non-Transformers". and **Algorithm 3**: "Partition for Transformers".

Step 2. For each parameter block, we use a single learning rate. To efficiently choose a suitable learning rate in each block, Adam-mini simply replaces $\texttt{g} \odot \texttt{g}$ in vanilla Adam by its mean value. We adopt the moving average on these mean values as in Adam.

A simple example of Adam-mini. We use a simple example to illustrate the key design of Adam-mini. For a problem with 5 parameters $w \in \mathbb{R}^5$, Adam and Adam-mini both perform $w = w - u \odot m$, where m is the 1st-order momentum and u has different forms as follows:

- For Adam: $u_{\text{Adam}} = \left(\frac{\eta}{\sqrt{v_1}}, \frac{\eta}{\sqrt{v_2}}, \frac{\eta}{\sqrt{v_3}}, \frac{\eta}{\sqrt{v_4}}, \frac{\eta}{\sqrt{v_5}}\right)$.
- For Adam-mini: suppose the partition is $(1,2,3)$ and $(4,5)$ then

$$u_{\text{mini}} = \left(\frac{\eta}{\sqrt{(v_1+v_2+v_3)/3}}, \frac{\eta}{\sqrt{(v_1+v_2+v_3)/3}}, \frac{\eta}{\sqrt{(v_1+v_2+v_3)/3}}, \frac{\eta}{\sqrt{(v_4+v_5)/2}}, \frac{\eta}{\sqrt{(v_4+v_5)/2}}\right).$$



(a) Randomly leave one block out (b) Randomly leave two blocks out (c) Randomly leave three blocks out (d) Performance gap

Figure 6: (a) (b) (c) Adam (leave-x-out) can reach a similar or better performance than Adam for all randomly picked left-out blocks. $x = 1, 2, 3$. (d) The performance gap between Adam and Adam (leave-one-out) for all possible blocks. We find Adam (leave-one-out) always performs on par with Adam, and for most blocks, Adam (leave-one-out) performs better.

Note that the number of effective elements u_{mini} equals the number of blocks, which could be significantly smaller than that of u_{Adam}, which equals the number of parameters. For LLMs, this will free $\geq 99.9\%$ elements in v.

2.3 Principle for the Partition Strategy

We now discuss how to choose the parameter partition for Adam-mini. A straightforward way is to use PyTorch default partition. Unfortunately, we find that the PyTorch default partition does not always work well on larger-scaled tasks including Transformer training. In particular, we find that Adam-mini encounters training instability on 1B models (see Figure 7 (d)). We suspect this is because the default PyTorch partition did not fully capture the Hessian structure. We propose a general principle in **Principle 1** below.

Principle 1: We should partition parameters into blocks, such that each parameter block is associated with the smallest dense sub-block in Hessian.

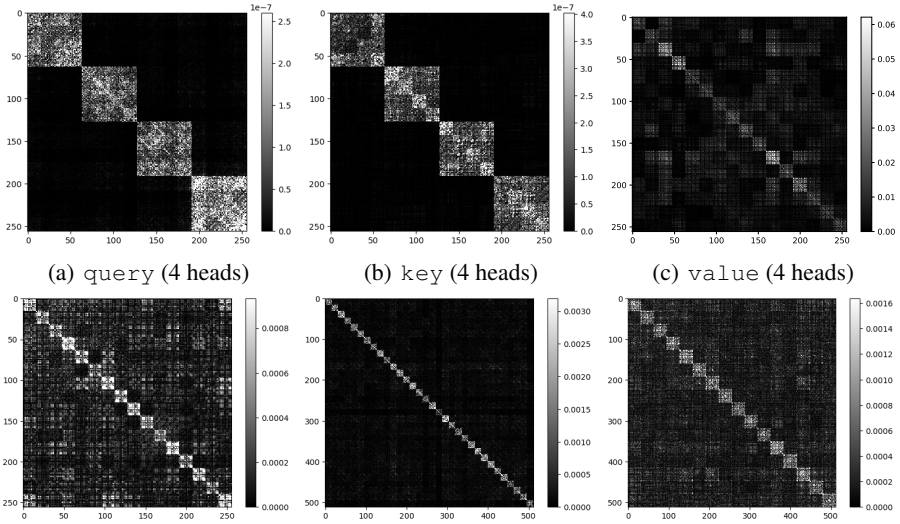
Principle 1 comes from the analysis in Section 2.1: it is possible to harmlessly reduce the number of Adam's learning rates within each dense Hessian block. However, if the partition is too coarse and violates **Principle 1**, we might accidentally remove some crucial learning rates and oversimplify the problem, causing training failure. It is important to follow **Principle 1** since it is necessary to use (at least) one distinct learning rate for each Hessian block (as evident in Appendix D.2).

Does the PyTorch default partition follow **Principle 1**? To find out, we explore the Hessian of a small Transformer as in Figure 7. Under the default PyTorch partition, we compute the exact Hessian at initialization for each parameter block. We find there are three classes of Hessian sub-blocks.

- **Class 1: `query` and `key`.** For `query` and `key`, the Hessian sub-block itself has a block-diagonal structure and consists of smaller dense matrices. We empirically find that the number of small dense sub-blocks equals the number of heads in multi-head attention.
- **Class 2: `attn.proj` and MLPs.** For `attn.proj`, `mlp.fc_1`, and `mlp.proj`, the Hessian sub-block has a block-diagonal structure and the number of small dense sub-blocks equals the number of output neuron. This observation suggests that the calculation in Eq. (1) not only holds in pure MLPs, but also can generalize to various building blocks in Transformers.
- **Class 2: `value`.** For `value`, the structure of Hessian sub-blocks seems less clear. It seems to have the hint of 16 diagonal blocks (16 is the number of output neurons), but the pattern is less obvious. This Hessian structure is significantly different from that of `query` and `key`, although they all consist of four heads. The Hessian entries of `value` are also about 10^5 larger than those of `query` and `key` [3]. One possible reason is that `value` is positioned outside the softmax operator in the self-attention design, while `query` and `key` are not.

Based on the above findings, we find that the PyTorch default partition is indeed not the best fit for Transformers. By **Principle 1**, `query` and `key` should be further partitioned by heads; `value`, `attn.proj`, and MLPs should be partitioned by output neurons; `embed_layer` and `output_layer` should be partitioned by tokens. As for `value`, the Hessian shows the hint of 16 diagonal blocks (where 16 is the number of output neurons), but the pattern is less clear. Our experiments show that "partition `value` by output neurons" works well in general, yet there are also some special cases where it is better to "treat `value` as a whole" (see discussions in Appendix C.5). By default, we will partition `value` by output neurons.

[3] This might be one source of the heterogeneity of Hessian eigenvalues as reported by (Zhang et al., 2024).



(a) `query` (4 heads) (b) `key` (4 heads) (c) `value` (4 heads)

(d) `attn.proj` (16 neurons) (e) `mlp.fc_1` (32 neurons) (f) `mlp.proj` (16 neurons)

Figure 7: (a-f): The initial Hessian of different parameter blocks in a small Transformer on Openwebtext. Here, neuron refers to the "output neuron". We find that these Hessian sub-blocks (except for `value`) have near-block-diagonal structure and consists of smaller dense matrices. Different parameter blocks have different numbers of small dense matrices.

We then introduce the resulting **Algorithm 3**: "Partition for Transformers". As shown in Figure 7 (d). This strategy indeed stabilizes the training and boosts the performance.

2.4 Some Characteristics of Adam-mini and Discussions

Memory cut down. Adam-mini reduces the number of learning rates from the number of model parameters to the number of total number of blocks by our partition strategies. As a result, Adam-mini cuts down more than 99.9% of Adam's v, which saves 50% of Adam's memory.

Higher throughput. Adam-mini can reach a higher throughput than AdamW, especially under limited GPU resources. There are two reasons. First, Adam-mini does not introduce extra computation in its update rules. The averaging operation in **Algorithm 1** incurs negligible cost and it significantly reduces the number of vector-square-root and vector-division operations in AdamW. Second, thanks to the memory cut-down, Adam-mini can support larger batch sizes per GPU. It also reduces the communication among GPUs, which is known to be a major overhead (Rajbhandari et al., 2021).

Owing to these properties, Adam-mini could reduce the overall time for pre-training. We provide evidence in Table 2. When pre-training Llama 2-7B on $2\times$ A800-80GB GPUs, we find Adam-mini could reach 49.6% higher throughput than AdamW. This translates to $\mathbf{33.1\%}$ **reduction of wall-clock time** on processing the same amount of tokens for pre-training.

Why using average v as learning rates. In Line 9 of **Algorithm 1**, we use the average of v in a block as the learning rate for that block. We choose such a design due to the following reasons.

- **First: grid-search is too expensive.** Optimal blockwise learning rates can be powerful (as evident in Figure 6), but they are too expensive to search. Such a searching procedure is not scalable.
- **Second: average of v can be borrowed from Adam.** Compared to search all the learning rates from scratch, it is much easier to "borrow" them from the current design of Adam, and the average of v is the most natural quantity to "borrow". Although the average of v might not be so powerful as grid-searching, it is much more simple to use. As a result, we find such design helps Adam-mini to be as effective as Adam (though not significantly surpassing it).
- **Third: average of v keeps us close to Adam.** For neural nets, we find that the average of v is a good representative for the whole v in the block, and can help Adam-mini keep close to Adam. The reason comes from backpropagation (BP) rule: for one data sample, the gradient of the weight matrix $W \in \mathbb{R}^{d\times d}$ can be expressed as $G := \frac{\partial \ell}{\partial W} = ez^\top \in \mathbb{R}^{d\times d}$, where e is certain BP error vector and z is the input feature to the current weight. For all entries in the i-th row of G, they all share the same BP error term e_i, which is usually non-negligible when $G \neq 0$. Therefore, G usually has similar entries within a row (which associates with the same output neuron), and its mean value can be a good representative of the whole row. As a result, we find that Adam-mini's trajectory closely resembles that of Adam (see the curves in Figure 9 and Figure 10). **One resulting advantage is that Adam-mini can maintain the scaling laws of LLMs trained by Adam, while substantially saving the training cost (see evidence in Figure 12).**

Table 1: Memory cost of AdamW v.s. Adam-mini. Calculation is based on `float32`, which is a standard choice for optimizer states.

Model	Optimizer	Memory (GB)
GPT-2-1.5B	AdamW	12.48
GPT-2-1.5B	Adam-mini	6.24 (**50%** ↓)
Llama 2-1B	AdamW	8.80
Llama 2-1B	Adam-mini	4.40 (**50%** ↓)
Llama 2-7B	AdamW	53.92
Llama 2-7B	Adam-mini	26.96 (**50%** ↓)
Llama 2-13B	AdamW	104.16
Llama 2-13B	Adam-mini	52.08 (**50%** ↓)

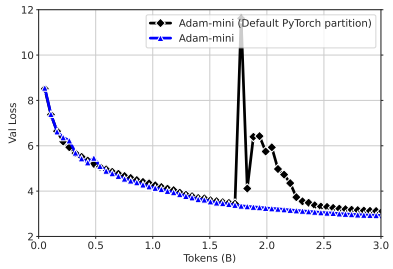


Figure 8: Training curves on Llama 2-1B. When using the PyTorch default partition, Adam-mini could suffer loss spikes. The spike disappears when we use the partition strategy in **Algorithm 3**.

Table 2: Throughput (↑) test on 2× A800-80GB GPUs for Llama 2-7B pre-training. ✗means out of memory. GPU hours (↓) to pre-train Llama 2-7B with the optimal token amount by Chinchila's law.

Optimizer	bs_per_GPU	total_bs	Throughput (↑)
Adam-mini	4	256	5572.19 (↑ **49.6%**)
AdamW	2	256	✗
AdamW	1	256	3725.59

Optimizer	# Tokens (B)	GPU hours (h) (↓)
AdamW	1	74.56
Adam-mini	1	49.85 (↓ **33.1%**)
AdamW	70	5219.16
Adam-mini	70	3489.55 (↓ **33.1%**)
AdamW	140	10438.32
Adam-mini	140	6979.10 (↓ **33.1%**)

Has room to improve. Adam-mini designs the learning rate for each dense Hessian sub-block using the average of Adam's v in that block. Such a design achieves cheap computation, but it might not be optimal. We believe there is great room to improve the learning rate design. As shown in Figure 4, we can reach much faster convergence if we utilize more information in the dense block to design the learning rate (e.g., using eigenvalues of each block), However, such a design requires expensive computation. We leave it as an important future direction.

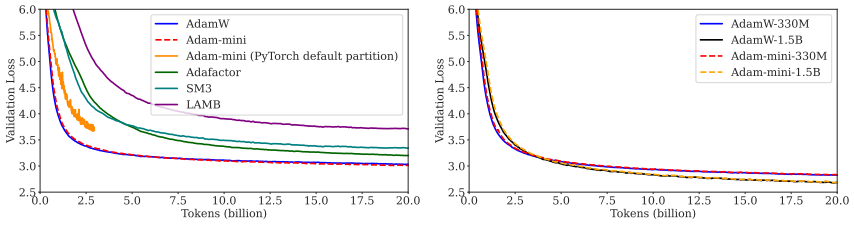
3 Experiments

We now verify the efficacy of Adam-mini on two types of neural-net tasks: (1) LLM tasks including pre-training, supervised fine-tuning (SFT), and reinforcement learning from human feedback (RLHF). (2) Non-LLM tasks including vision, graph, and diffusion model training. **Due to the limited space, we primarily focus on LLM tasks in this section, and we relegate the non-LLM tasks to Appendix C.4.** All LLM experiments are conducted on four NVIDIA A800-80GB GPUs and the rest are conducted on four V100 GPUs. All the experimental details are explained in Appendix E.1.

3.1 Pre-training

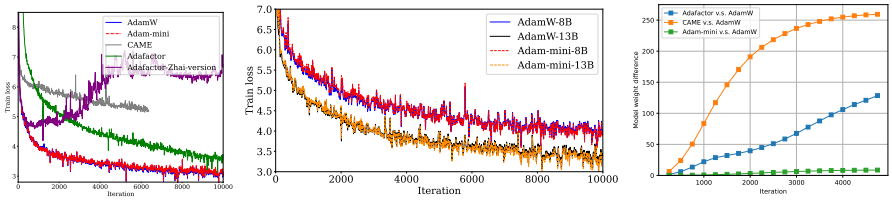
Setups. We pre-train LLMs including GPT-2 series and Llama series. We train these models on mainstream English Corpus from scratch. In particular, We train GPT-2 (Radford et al., 2019) series (125M to 1.5B) on Openwebtext (Gokaslan et al., 2019). We train Llama series (20M to 13B) (Touvron et al., 2023) on C4 (Raffel et al., 2020). We compare Adam-mini with AdamW (Loshchilov & Hutter, 2017) as well as popular memory-efficient methods including Adafactor (Shazeer & Stern, 2018), CAME (Luo et al., 2023), and SM3 (Anil et al., 2019). For Adafactor and SM3, we incorporate momentum with $\beta_1 = 0.9$ to ensure a fair comparison with other methods. We tune the learning rate for all methods, using the same tuning budget for each, and report the best performance.

GPT-2 series. Figure 9 (b) shows the loss curves for GPT-2 sized from 125M to 1.5B. We find that Adam-mini performs similarly to AdamW with less memory, while other methods perform worse. In Figure 9 (a), we include results for *Adam-mini (embd_blocks_removed)*, which sets the `embd_blocks` $= \emptyset$. That is, we use one single learning rate for the whole embedding (output) layer. We find that Adam-mini (embd_blocks_removed) performs poorly, as expected from the analysis in Section 2.1. We stopped the trial since it shows clear unstable behavior.



(a) GPT-2-125M (b) GPT-2-330M and 1.5B

Figure 9: For GPT-2 series pre-training, Adam-mini performs similarly to AdamW with 50% less memory, while other methods perform worse.



(a) Llama 2-1B (b) Llama 3-8B and Llama 2-13B (c) Trajectory comparison

Figure 10: (a, b): Training curves of Llama 2-1B, Llama 3-8B, and Llama 2-13B. Adam-mini performs on par or better than AdamW with about 50% less memory, while other methods perform worse. In (a), we stopped CAME due to unexpected infrastructure break-down and we cannot afford to re-run. (c): Adam-mini can generate similar trajectories to AdamW (in terms of the ℓ-2 distance of model checkpoints).

Llama series. Figure 10 shows the loss curve for pre-training Llama 2-1B, Llama 3-8B and Llama 2-13B. We also train Llama 2-7B as shown in Figure 1 (c) in Section 1. We find that Adam-mini performs on par with AdamW with about 50% less memory, while other methods perform worse.

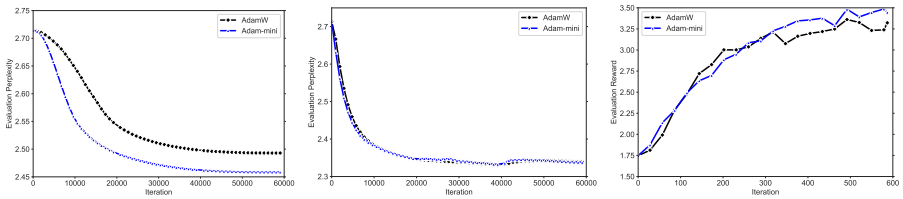
Sensitivity analysis. On GPT-2-125M pre-training task, we test the sensitivity of Adam-mini to hyperparameters. We report the validation loss after training with 2.5B tokens (by Chinchilla's law). As shown in Figure 12 (b), Adam-mini seems not overly sensitive to hyperparameters.

Trajectory comparison. On a small Transformer, Adam-mini generates similar trajectories to that of AdamW, while other methods cannot. This can be seen in Figure 10 (c) and the detailed description is in Appendix E. This might because Adam-mini makes fewer modifications over AdamW.

3.2 Supervised Fine-tuning and RLHF

We now run Adam-mini on downstream tasks including SFT and RLHF. We use the Llama 2-7B pretrained model (Touvron et al., 2023) for our study. We use the `ultrafeedback` dataset and implement the RLHF workflow from (Ouyang et al., 2022). We use ReMax (Li et al., 2023), a memory-efficient alternative to PPO (Schulman et al., 2017), to optimize the preference reward.

We evaluate the alignment performance in terms of chat ability using the MT-Bench (Zheng et al., 2024), where GPT-4 assesses multi-turn chatting capabilities and assigns a score from 0 to 10 (higher is better). Our results, presented in Table 3, demonstrate that Adam-mini can outperform AdamW.



(a) SFT (LoRA) (b) SFT (c) RLHF

Figure 11: Training curves of SFT (LoRA), SFT, and RLHF when aligning Llama 2-7B. Adam-mini reaches better performance (smaller perplexity, higher reward) than AdamW with less memory.

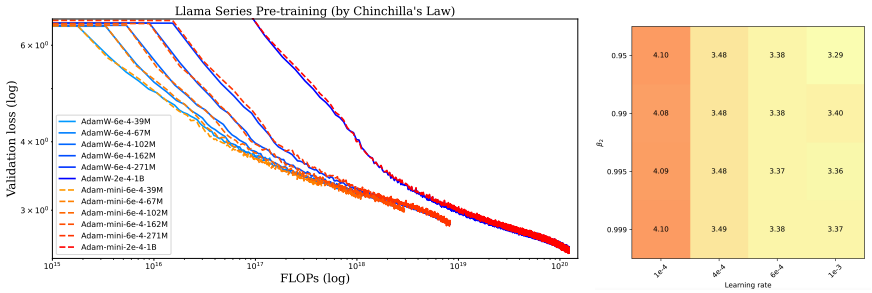
Table 3: Averaged GPT-4 evaluation score (↑) of SFT and RLHF on the MT-Bench.

	SFT (LoRA)		SFT		RLHF	
	AdamW	Adam-mini	AdamW	Adam-mini	AdamW	Adam-mini
MT-Bench	4.23	**4.41**	5.37	**5.40**	5.54	**5.68**

3.3 Scaling Laws of Adam-mini

We now show the efficacy of Adam-mini through scaling law experiments. We use C4 dataset to pre-train the Llama 2 architecture from 39M to 1B. For the model with size n_{param}, we train the model with about $20 * n_{\text{param}}$ tokens, which is suggested to be the optimal amount by Chinchilla's law (Hoffmann et al., 2022). The largest-scaled experiment we conducted is Llama 2-1B pre-training with 26.2B tokens, which takes about 170 GPU hours on $4\times$ A800-80GB GPUs. The total running time for the scaling law experiments is about 300 GPU hours.

As shown in Figure 12, Adam-mini consistently performs similarly to AdamW for all models. We also present the final validation perplexity and find that **Adam-mini reaches a lower perplexity than AdamW for all models** (see Table 5 in Appendix C). These scaling law experiments can serve as additional evidence that Adam-mini can be scaled up to larger models (if the scaling law holds).



(a) Scaling laws of Adam-mini (b) Sensitivity analysis

Figure 12: (a) Scaling laws of Adam-mini. We pre-train Llama 2 architectures following Chinchilla's law. For all models sized from 39M to 1B, we consistently find Adam-mini's performance to be similar to AdamW. (b) We find that Adam-mini seems not overly sensitive to hyperparameters.

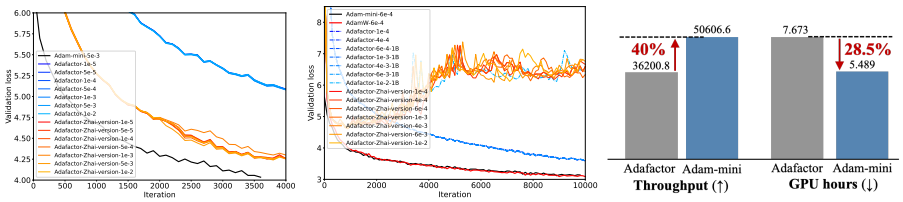
3.4 Detailed Comparison with Adafactor

We now carefully compare Adam-mini and the popular memory-efficient optimizer Adafactor. Besides the original Adafactor, we also consider a modified version in (Zhai et al., 2022), which we call "Adafactor-Zhai-version". For both versions, we use momentum with $\beta_1 = 0.9$.

We first conduct learning rate grid-search on Llama 2-20M and train it following Chinchilla's law. As shown in Figure 13 (a), we find that Adafactor-Zhai-version improves over the original version, but both versions of Adafactor are still consistently worse than Adam-mini. We further sweep over other hyperparameters including (1) $\beta_2 = 0.95$; (2) $\epsilon = \{10^{-30}, 10^{-16}, 10^{-8}, 10^{-6}\}$; (3) warm-up steps = $\{1\%, 2\%, 3\%, 4\%, 5\%, 10\%\}$ total steps. The results are shown in Appendix C.6. We find that the change of hyperparameters does not significantly boost the performance of Adafactor, and both versions still underperform Adam-mini.

We further sweep hyperparameters on Llama 2-1B. In contrast to the case of Llama 2-20M, we find that the Adafactor-Zhai-version now suffers from training instability and the original version performs better. Nevertheless, they still underperform Adam-mini. In Appendix C.7, we conduct a similar hyperparameter search for Lion (Chen et al., 2024) and we find it also underperforms Adam-mini.

About hyperparameter tuning. We acknowledge that it might be possible to improve these methods if we spend more resources on grid search (as claimed by a recent work (Zhao et al., 2024b)). However, based on our experience so far, it is not easy to tune these methods, and to our knowledge, there is no much open-source guidance. Recall that there are 9 tunable hyperparameters in Adafactor, so it is rather non-trivial to find the correct combination. In contrast, Adam-mini is much easier to use. In all our experiments, Adam-mini performs well using **the same hyperparameters as AdamW.**



(a) Llama 2-20M (b) Llama 2-1B (c) Throughput comparison

Figure 13: (a, b): Training curves of Adafactor & Adafactor-Zhai-version on Llama 2-20M & Llama 2-1B pre-training. We find these two methods consistently underperform Adam-mini. (c): On Llama 2-1B, Adam-mini achieves 40% higher throughput than Adafactor (tested on 2× A800-80GB GPUs).

Throughput comparison. Besides the performance comparison, we further find that Adafactor has higher latency than Adam-mini (Figure 13 (c)). This is primarily due to two reasons. First, Adam-mini only requires computing the mean by rows of the weight matrix, whereas Adafactor needs to sum across both the rows and the columns. Second, the dimension of v in Adam-mini equals the output dimension or the number of heads, which is significantly smaller than the dimension of v in Adafactor, which equals the product of the input and output dimension. Consequently, Adam-mini saves computation when taking the square root of v. As such, Adam-mini reaches higher throughput.

4 Concluding Remarks

We proposed Adam-mini, an optimizer that saves 50% memory of Adam. We remark that there is great room to improve the design of Adam-mini: currently Adam-mini uses a simple and cost-effective way to design a learning rate for each dense Hessian sub-block, but it might not be an optimal way. We leave the development of even stronger designs as a future direction.

REFERENCES

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

Kwangjun Ahn, Zhiyu Zhang, Yunbum Kook, and Yan Dai. Understanding adam optimizer via online learning of updates: Adam is ftrl in disguise. In *Forty-first International Conference on Machine Learning*.

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

Anonymous authors. Deconstructing what makes a good optimizer for language models. *https://openreview.net/pdf?id=zfeso8ceqr*, 2024.

Pratik Chaudhari, Anna Choromanska, Stefano Soatto, Yann LeCun, Carlo Baldassi, Christian Borgs, Jennifer Chayes, Levent Sagun, and Riccardo Zecchina. Entropy-sgd: Biasing gradient descent into wide valleys. *Journal of Statistical Mechanics: Theory and Experiment*, 2019(12):124018, 2019.

Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

Xiangning Chen, Chen Liang, Da Huang, Esteban Real, Kaiyuan Wang, Hieu Pham, Xuanyi Dong, Thang Luong, Cho-Jui Hsieh, Yifeng Lu, et al. Symbolic discovery of optimization algorithms. *Advances in neural information processing systems*, 36, 2024.

Ronan Collobert. Large scale machine learning. Technical report, Université de Paris VI, 2004.

Ganqu Cui, Lifan Yuan, Ning Ding, Guanming Yao, Wei Zhu, Yuan Ni, Guotong Xie, Zhiyuan Liu, and Maosong Sun. Ultrafeedback: Boosting language models with high-quality feedback, 2023.

André Belotto Da Silva and Maxime Gazeau. A general system of differential equations to model first-order adaptive algorithms. *The Journal of Machine Learning Research*, 21(1):5072–5113, 2020.

Rudrajit Das, Naman Agarwal, Sujay Sanghavi, and Inderjit S Dhillon. Towards quantifying the preconditioning effect of adam. *arXiv preprint arXiv:2402.07114*, 2024.

Yann N Dauphin, Atish Agarwala, and Hossein Mobahi. Neglected hessian component explains mysteries in sharpness regularization. *arXiv preprint arXiv:2401.10809*, 2024.

Aaron Defazio, Harsh Mehta, Konstantin Mishchenko, Ahmed Khaled, Ashok Cutkosky, et al. The road less scheduled. *arXiv preprint arXiv:2405.15682*, 2024.

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

Tim Dettmers, Mike Lewis, Sam Shleifer, and Luke Zettlemoyer. 8-bit optimizers via block-wise quantization. In *International Conference on Learning Representations*, 2021.

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

George E Forsythe and Ernst G Straus. On best conditioned matrices. *Proceedings of the American Mathematical Society*, 6(3):340–345, 1955.

Behrooz Ghorbani, Shankar Krishnan, and Ying Xiao. An investigation into neural net optimization via hessian eigenvalue density. In *International Conference on Machine Learning*, pp. 2232–2241. PMLR, 2019.

Boris Ginsburg, Patrice Castonguay, Oleksii Hrinchuk, Oleksii Kuchaiev, Vitaly Lavrukhin, Ryan Leary, Jason Li, Huyen Nguyen, Yang Zhang, and Jonathan M Cohen. Training deep networks with stochastic gradient normalized by layerwise adaptive second moments. 2019.

Aaron Gokaslan, Vanya Cohen, Ellie Pavlick, and Stefanie Tellex. Openwebtext corpus, 2019.

Guy Gur-Ari, Daniel A Roberts, and Ethan Dyer. Gradient descent happens in a tiny subspace. *arXiv preprint arXiv:1812.04754*, 2018.

Alexander Hägele, Elie Bakouch, Atli Kosson, Loubna Ben Allal, Leandro Von Werra, and Martin Jaggi. Scaling laws and compute-optimal training beyond fixed training durations. *arXiv preprint arXiv:2405.18392*, 2024.

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

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

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.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.

Adam Ibrahim, Benjamin Thérien, Kshitij Gupta, Mats L Richter, Quentin Anthony, Timothée Lesort, Eugene Belilovsky, and Irina Rish. Simple and scalable strategies to continually pre-train large language models. *arXiv preprint arXiv:2403.08763*, 2024.

Kaiqi Jiang, Dhruv Malik, and Yuanzhi Li. How does adaptive optimization impact local neural network geometry? *Advances in Neural Information Processing Systems*, 36, 2023.

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

Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations*, 2016.

Fuad Kittaneh. Spectral radius inequalities for hilbert space operators. *Proceedings of the American Mathematical Society*, pp. 385–390, 2006.

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.

Frederik Kunstner, Robin Yadav, Alan Milligan, Mark Schmidt, and Alberto Bietti. Heavy-tailed class imbalance and why adam outperforms gradient descent on language models. *arXiv preprint arXiv:2402.19449*, 2024.

Bingrui Li, Jianfei Chen, and Jun Zhu. Memory efficient optimizers with 4-bit states. *Advances in Neural Information Processing Systems*, 36, 2024.

Ziniu Li, Tian Xu, Yushun Zhang, Yang Yu, Ruoyu Sun, and Zhi-Quan Luo. Remax: A simple, effective, and efficient method for aligning large language models. *arXiv preprint arXiv:2310.10505*, 2023.

Zhenyu Liao and Michael W Mahoney. Hessian eigenspectra of more realistic nonlinear models. *Advances in Neural Information Processing Systems*, 34:20104–20117, 2021.

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

Yang Liu, Jeremy Bernstein, Markus Meister, and Yisong Yue. Learning by turning: Neural architecture aware optimisation. In *International Conference on Machine Learning*, pp. 6748–6758. PMLR, 2021.

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

Qijun Luo, Hengxu Yu, and Xiao Li. Badam: A memory efficient full parameter training method for large language models. *arXiv preprint arXiv:2404.02827*, 2024.

Yang Luo, Xiaozhe Ren, Zangwei Zheng, Zhuo Jiang, Xin Jiang, and Yang You. Came: Confidence-guided adaptive memory efficient optimization. *arXiv preprint arXiv:2307.02047*, 2023.

Kai Lv, Hang Yan, Qipeng Guo, Haijun Lv, and Xipeng Qiu. Adalomo: Low-memory optimization with adaptive learning rate. *arXiv preprint arXiv:2310.10195*, 2023a.

Kai Lv, Yuqing Yang, Tengxiao Liu, Qinghui Gao, Qipeng Guo, and Xipeng Qiu. Full parameter fine-tuning for large language models with limited resources. *arXiv preprint arXiv:2306.09782*, 2023b.

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

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

Francesco Orabona. Neural networks (maybe) evolved to make adam the best optimizer. 2020. URL `https://parameterfree.com/2020/12/06/neural-network-maybe-evolved-to-make-adam-the-best-optimizer/`.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744, 2022.

Yan Pan and Yuanzhi Li. Toward understanding why adam converges faster than sgd for transformers. *arXiv preprint arXiv:2306.00204*, 2023.

Vardan Papyan. The full spectrum of deepnet hessians at scale: Dynamics with sgd training and sample size. *arXiv preprint arXiv:1811.07062*, 2018.

Vardan Papyan. Measurements of three-level hierarchical structure in the outliers in the spectrum of deepnet hessians. *arXiv preprint arXiv:1901.08244*, 2019.

Vardan Papyan. Traces of class/cross-class structure pervade deep learning spectra. *The Journal of Machine Learning Research*, 21(1):10197–10260, 2020.

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

Zhaonan Qu, Wenzhi Gao, Oliver Hinder, Yinyu Ye, and Zhengyuan Zhou. Optimal diagonal preconditioning: Theory and practice. *arXiv preprint arXiv:2209.00809*, 2022.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

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. URL `http://jmlr.org/papers/v21/20-074.html`.

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–16. IEEE, 2020.

Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*, pp. 1–14, 2021.

Nicolas Roux, Pierre-Antoine Manzagol, and Yoshua Bengio. Topmoumoute online natural gradient algorithm. *Advances in neural information processing systems*, 20, 2007.

Levent Sagun, Leon Bottou, and Yann LeCun. Eigenvalues of the hessian in deep learning: Singularity and beyond. *arXiv preprint arXiv:1611.07476*, 2016.

Levent Sagun, Utku Evci, V Ugur Guney, Yann Dauphin, and Leon Bottou. Empirical analysis of the hessian of over-parametrized neural networks. *arXiv preprint arXiv:1706.04454*, 2017.

Adepu Ravi Sankar, Yash Khasbage, Rahul Vigneswaran, and Vineeth N Balasubramanian. A deeper look at the hessian eigenspectrum of deep neural networks and its applications to regularization. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pp. 9481–9488, 2021.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

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

Naichen Shi, Dawei Li, Mingyi Hong, and Ruoyu Sun. Rmsprop converges with proper hyper-parameter. In *International Conference on Learning Representations*, 2020.

Ruoyu Sun and Yinyu Ye. Worst-case complexity of cyclic coordinate descent: O (nˆ 2) o (n 2) gap with randomized version. *Mathematical Programming*, 185:487–520, 2021.

Gemini Team, Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, et al. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*, 2023.

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

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, Yoshua Bengio, et al. Graph attention networks. *stat*, 1050(20):10–48550, 2017.

Bohan Wang, Yushun Zhang, Huishuai Zhang, Qi Meng, Zhi-Ming Ma, Tie-Yan Liu, and Wei Chen. Provable adaptivity in adam. *arXiv preprint arXiv:2208.09900*, 2022.

Yikai Wu, Xingyu Zhu, Chenwei Wu, Annie Wang, and Rong Ge. Dissecting hessian: Understanding common structure of hessian in neural networks. *arXiv preprint arXiv:2010.04261*, 2020.

Xingyu Xie, Pan Zhou, Huan Li, Zhouchen Lin, and Shuicheng Yan. Adan: Adaptive nesterov momentum algorithm for faster optimizing deep models. *arXiv preprint arXiv:2208.06677*, 2022.

Zhewei Yao, Amir Gholami, Qi Lei, Kurt Keutzer, and Michael W Mahoney. Hessian-based analysis of large batch training and robustness to adversaries. *Advances in Neural Information Processing Systems*, 31, 2018.

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

Yang You, Jing Li, Sashank Reddi, Jonathan Hseu, Sanjiv Kumar, Srinadh Bhojanapalli, Xiaodan Song, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. Large batch optimization for deep learning: Training bert in 76 minutes. *arXiv preprint arXiv:1904.00962*, 2019.

David Young. Iterative methods for solving partial difference equations of elliptic type. *Transactions of the American Mathematical Society*, 76(1):92–111, 1954.

Xiaohua Zhai, Alexander Kolesnikov, Neil Houlsby, and Lucas Beyer. Scaling vision transformers. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 12104–12113, 2022.

Guodong Zhang, Lala Li, Zachary Nado, James Martens, Sushant Sachdeva, George Dahl, Chris Shallue, and Roger B Grosse. Which algorithmic choices matter at which batch sizes? insights from a noisy quadratic model. *Advances in neural information processing systems*, 32, 2019a.

Jingzhao Zhang, Tianxing He, Suvrit Sra, and Ali Jadbabaie. Why gradient clipping accelerates training: A theoretical justification for adaptivity. *arXiv preprint arXiv:1905.11881*, 2019b.

Jingzhao Zhang, Sai Praneeth Karimireddy, Andreas Veit, Seungyeon Kim, Sashank Reddi, Sanjiv Kumar, and Suvrit Sra. Why are adaptive methods good for attention models? *Advances in Neural Information Processing Systems*, 33:15383–15393, 2020.

Yushun Zhang, Congliang Chen, Naichen Shi, Ruoyu Sun, and Zhi-Quan Luo. Adam can converge without any modification on update rules. *Advances in Neural Information Processing Systems*, 35: 28386–28399, 2022.

Yushun Zhang, Congliang Chen, Tian Ding, Ziniu Li, Ruoyu Sun, and Zhi-Quan Luo. Why transformers need adam: A hessian perspective. *arXiv preprint arXiv:2402.16788*, 2024.

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

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*, 2024b.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in Neural Information Processing Systems*, 36, 2024.

Shuai Zheng and James T Kwok. Blockwise adaptivity: Faster training and better generalization in deep learning. *arXiv preprint arXiv:1905.09899*, 2019.

A RELATED WORKS

Understanding of Adam. There is an active line of works trying to understand why Adam works well (Zhang et al., 2019b; Wu et al., 2020; Shi et al., 2020; Zhang et al., 2022; Wang et al., 2022; Pan & Li, 2023; Jiang et al., 2023; Kunstner et al., 2023; Zhang et al., 2024; Ahn et al.; Kunstner et al., 2024). In contrast to these works, we point out that Adam's v might not function at its full potential as effectively as we expected: sometimes fewer learning rates can reach the same or better results (due to the dense Hessian sub-blocks). Our findings might motivate stronger optimizers that better fit the neural-net Hessian structure.

Similarly to in our Section 2.1, a recent work (Das et al., 2024) also explores the effectiveness of Adam's preconditioner D_{Adam} from a linear algebra perspective. They focus on (a variant of) Adam and prove the following result: First, for diagonal dominant (DD) matrix H_b when the dimension d is less than $\kappa^{1/3}$, their modified version of Adam exhibits a faster convergence rate compared to gradient descent (GD); Second, for non-DD matrix, the constant terms in Adam's upper bound can be much larger than that of GD. Their results take a valuable and important step towards understanding D_{Adam}. However, they remain insufficient to fully support our numerical findings in Figure 5. This is because they only provide an upper bound for the non-DD case, while we need a lower bound. We note that it is rather difficult to derive the desired lower bound, and we leave it as a future direction.

On the Hessian of Neural Nets. Hessian matrix is crucial for the behaviors of gradient methods. There are several important attempts to study the Hessian of MLPs and CNNs (Collobert, 2004; Roux et al., 2007; Martens & Grosse, 2015; Sagun et al., 2016; 2017; Chaudhari et al., 2019; Papyan, 2020; Wu et al., 2020; Liao & Mahoney, 2021; Papyan, 2018; 2019; Sankar et al., 2021; Gur-Ari et al., 2018; Yao et al., 2018; Zhang et al., 2019a; Ghorbani et al., 2019; Yao et al., 2020; Dauphin et al., 2024). Inspired by these works, we explore the Hessian structure of Transformers and connect it to the behaviors of Adam. We then find room to improve and propose to slim down Adam into Adam-mini.

Lightweight optimizers for general tasks. There are several attempts to reduce the memory cost of Adam. Adafactor (Shazeer & Stern, 2018) and its variant CAME (Luo et al., 2023) conduct nonnegative low-rank factorization over Adam's v. SM3 (Anil et al., 2019) is a lightweight version of AdaGrad (Duchi et al., 2011). SM3 chooses the learning rate of the i-th parameter by taking the minimal value in a certain candidate set, and each element in the candidate set is related to the maximal squared gradient under a predetermined cover. All these aforementioned methods could release almost all memory for v and save about 48% of Adam's memory. However, we find that their performance degenerate in various experiments, while Adam-mini maintains as effective as AdamW (as shown in Section 3).

After completing this work, we noticed two methods that share some of the ideas of Adam-mini: BAGM (Zheng & Kwok, 2019) and NovoGrad (Ginsburg et al., 2019). Both of them use block-wise or layer-wise adaptive learning rates to achieve robust performance and better generalization. We summarize their key differences with Adam-mini. BAGM partitions parameters to reach minimal-norm solutions and achieve provable robustness. In particular, their theory in Proposition 1 states that layer-by-layer parameter partition can lead to minimum ℓ-2 norm solutions. Aligning with the theory, they find that the PyTorch default partition (BAGM-B.1) indeed brings overall the best performance on both CIFAR-10 and ImageNet. Although the PyTorch default partition may have benefits on robustness, we find that it overlooks the Hessian structure and oversimplifies the training problem for Transformers (as we discussed in Section 2.3). As a result, the PyTorch default partition will lead to training instability in large-scale LLMs, and this is evident in our failed preliminary versions of Adam-mini in Figure 8 and 9. We then propose a new partition strategy **Algorithm 3** which partition parameters by the smallest dense Hessian sub-blocks. For Transformers, **Algorithm 3** uses different strategies for different building blocks (e.g., partition the embedding layer by tokens, and partition Query by heads) and we find that **Algorithm 3** is necessary to stabilize the training.

In summary, these two methods have different designs and their partition strategies oversimplify the training problems. Consequently, they would cause training instability on large-scale experiments as evident in Figure 8 and 9. In contrast, Adam-mini carefully assigns learning rates following our proposed principle on Hessian structures. Such design principle is crucial for training stability and it works well on various LLMs including 7B models.

Besides algorithmic design, our work also provides new understandings of Adam, and particularly, how Adam behaves on generic optimization problems with near-block-diagonal Hessian. We also provide new findings on the Hessian structure of Transformers and provide new principles for designing better algorithms.

Other orthogonal methods. The idea of Adam-mini can be orthogonally combined with various existing methods. We list two most relevant examples here.

1. GaLore (Zhao et al., 2024a) is a new memory-efficient optimizer for LLMs. Given a gradient matrix g, GaLore calculates a low-rank gradient estimator $\hat{g}$ and then calculates m and v based on this $\hat{g}$. Adam-mini can potentially be combined with GaLore to reach further memory reduction on v. The combined method, e.g., "GaLore-mini", can further reduce about 40% memory on GaLore and about 81% on AdamW in total.[4] Additionally, GaLore-mini can ease the offload burden and enhance the throughput of GaLore, especially when training on customer-level GPUs with limited memory.
2. Sophia (Liu et al., 2023) is another recent diagonal preconditioned optimizer. Just as Adam, Sophia requires memory for m and v. It is possible to combine Adam-mini and Sophia to get "Sophia-mini", which saves up to 50% of memory in Sophia. Sophia-mini can also enhance throughput and further speed up Sophia on wall-clock time as in Table 2.

We list more potential combinations here. LoRA (Hu et al., 2021) is a memory-efficient method for SFT tasks. This method fine-tunes the model via additive low-rank adaptors and uses Adam to update these adaptors. Note that the Adam steps in LoRA can be replaced by Adam-mini. As a result, Adam-mini brings better performance (Figure 11). In parallel to our work, BAdam (Luo et al., 2024) conducts SFT in a block-coordinate-descent (BCD) fashion. This method requires repeated Adam steps to solve the sub-problem in BCD. Similarly as in LoRA, the Adam steps in BAdam can be replaced by Adam-mini to further reduce memory. Nero optimizer (Liu et al., 2021) also cuts down the memory of Adam. It removes the 1st-order momentum and uses a neuron-specific projected gradient-style update. According to (Liu et al., 2021), their design imposes constraints on weight matrices and has the advantage of "balanced excitation and inhibition". Such design can potentially be combined with Adam-mini to further boost performance. To save the memory cost for fine-tuning LLMs, MeZO (Malladi et al., 2023) uses zeroth-order methods to approximate the gradient information. It is possible to combine this idea with Adam-mini to further save memory for SFT. Adam-mini can also potentially be combined with other diagonal preconditioned methods (such as AdaGrad (Duchi et al., 2011) and Adan (Xie et al., 2022)) as well as recent schedule-free optimizers such as SchedulefreeAdamW (Defazio et al., 2024).

There are several other tricks that ease GPU memory burden but are orthogonal to optimizer design. These tricks include gradient checkpointing (Chen et al., 2016), model offloading and sharding (Rajbhandari et al., 2020; 2021), quantization (Dettmers et al., 2021; Li et al., 2024), and fused update (Lv et al., 2023a;b). Adam-mini can be implemented upon these tricks.

Finally, we discuss another popular adaptive optimizer called LAMB (You et al., 2019) (see **Algorithm 7** in Appendix D.1). LAMB might be misunderstood as a similar optimizer to Adam-mini, but actually, it is not. We emphasize that Adam-mini is *significantly different* from LAMB. First, LAMB still keeps the same coordinate-wise learning-rate design $1/\sqrt{v}$ as in Adam. Second, in addition to this $1/\sqrt{v}$, LAMB re-scales the parameters in a layer-by-layer fashion. This re-scaling design is often known as the "layer-wise learning rates", but to be precise, it is actually an additional "layer-wise scaling" besides the "coordinate-wise learning rates $1/\sqrt{v}$". As a result, LAMB does not save memory over Adam and its overall design is quite different from Adam-mini. This is understandable because LAMB is designed for large-batch training, not for memory saving. Numerically, we find that LAMB performs worse than Adam-mini on GPT2 pre-training (Figure 10 (b)).

[4]These results are calculated based on (Zhao et al., 2024a, Table 1). We consider Llama 2-7B and $r = 1024$ in GaLore.

B THE COMPLETE FORM OF ADAM-MINI

We now present the specific realization of Adam-mini on Transformers and other architectures. To be precise, **Algorithm 3** should be renamed as "Partition for CNNs, Diffusion models, and Graph Neural Networks", since we have only tested **Algorithm 3** on these models. In the future, it is possible that we will have more complicated non-Transformer architectures on which **Algorithm 3** fails. In those cases, we need to investigate the Hessian structure of these new architectures (like what we did for Transformers) and then develop the concrete partition algorithms following our **Principle 1** in Section 2.2.

Algorithm 2 Adam-mini in Pytorch style

Input weight-decay coefficient λ and current step t
Choose `param_blocks` from **Algorithm 3 or 3**
for `param` in `param_blocks` **do**
  `g` = `param.grad`
  `param` = `param` - $\eta_t * \lambda *$ `param`
  $\mathtt{m} = (1-\beta_1) * \mathtt{g} + \beta_1 * \mathtt{m}$
  $\hat{\mathtt{m}} = \frac{\mathtt{m}}{1-\beta_1^t}$
  $\mathtt{v} = (1-\beta_2) * \mathtt{mean}(\mathtt{g} \odot \mathtt{g}) + \beta_2 * \mathtt{v}$
  $\hat{\mathtt{v}} = \frac{\mathtt{v}}{1-\beta_2^t}$
  `param` = `param` - $\eta_t * \frac{\hat{\mathtt{m}}}{\sqrt{\hat{\mathtt{v}}}+\epsilon}$
end for

Algorithm 3 Partition for non-Transformers

```
param_blocks = {}
for name, param in parameters do
  param_blocks[name]=param
end for
return param_blocks
```

Algorithm 3 Partition for Transformers

```
param_blocks = {}
for name, param in parameters do
  if 'embed' or 'output' in name then
    Partition param by tokens
    for i = 0...tokens-1 do
      param_blocks[name+i]=param[i]
    end for
  else if 'query' or 'key' in name then
    Partition param by heads
    for i = 0...heads-1 do
      param_blocks[name+i]=param[i]
    end for
  else if 'value', 'attn.proj', or 'mlp' in name then
    Partition param by output neurons
    for i = 0...output_neurons-1 do
      param_blocks[name+i]=param[i]
    end for
  else
    param_blocks[name]=param
  end if
end for
return param_blocks
```

C MORE EXPERIMENTAL RESULTS

C.1 MORE RESULTS FOR MOTIVATION

In Section 2, we showed that "Adam's v is redundant on dense Hessian subblock". We provide experiments on random quadratic functions to support the claim. Here, we conduct the following new experiments on a 1-layer Transformer. We will show that "using single learning rate per block" is also sufficient for Transformers.

The following **Exp 1 and 2** extend the random quadratic experiments in **Figure 4 and 5** to Transformers.

Exp 1: Adam's learning rate is redundant on the dense Hessian subblock. We take some small dense blocks in the Hessian of 1-layer Transformer and denoted as H. We compare $\kappa(H)$ and $\kappa(D_{Adam}H)$ as in the paper. We find Adam is not effective in reducing the kappa of these blocks, and many lrs in Adam can be redundant.

Exp 2: Single learning rate per block is sufficient. We conduct the "block-wise GD" and we grid-search the learning rate for each block. The result is shown in the following figure. We find that block-wise GD outperforms AdamW. This extends the setting from the random quadratic problem in Figure 4.

Table 4: Comparison of $\kappa(H)$ and $\kappa(D_{Adam}H)$ for the dense blocks in the Hessian of 1-layer Transformer.

Hessian Block	$\kappa(H)$	$\kappa(D_{Adam}H)$
1st head in Query	103.80	176.88
1st head in Key	103.46	213.82
1st head in Value	165.66	332.76
1st neuron in attn.proj	39.92	94.56
1st neuron in MLP_fc1	22.04	70.92
1st neuron in MLP_c_proj	63.85	236.71

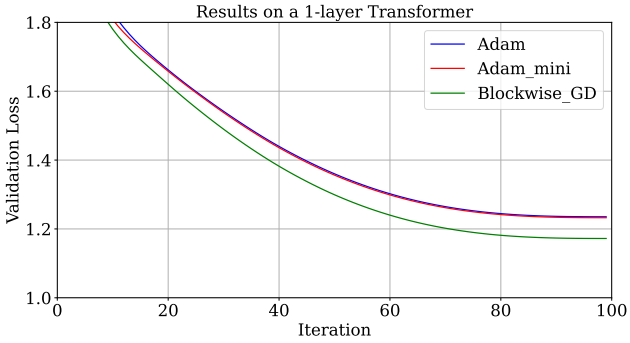


Figure 14: On a 1-layer Transformer, we conduct the "blockwise GD" and we grid-search the learning rate for each block. We find that blockwise GD outperforms AdamW.

Combining **Exp 1 and 2**, we can see that Adam is redundant on the dense Hessian subblocks (**Exp 1**), and a single lr for each block can work well (**Exp 2**). These experiments show that our conclusions on random quadratic problems can be extended to Transformers.

C.2 Ablation Studies on the Design of Adam-mini

We here provide more reasons why we choose $\texttt{mean}(v)$ as the blockwise learning rates. We conduct ablation studies on different choices of quantities that we can borrow from Adam, including $\texttt{2-norm}(v)$, $\texttt{1-norm}(v)$, $\texttt{max}(v)$, and $\texttt{min}(v)$. we found that all these candidates perform worse than $\texttt{mean}(v)$ in Adam-mini.

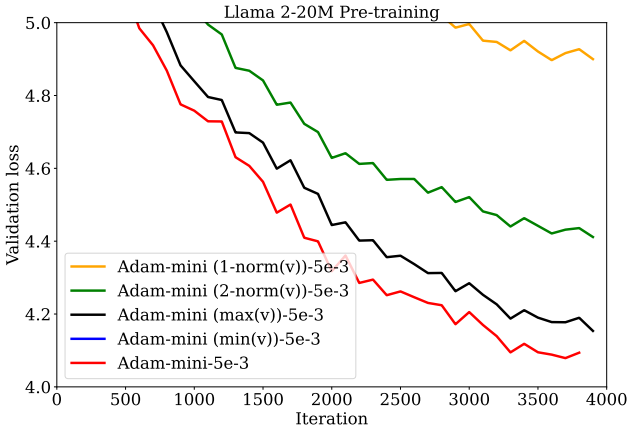


Figure 15: Ablation studies on the design of Adam-mini. We find that $\texttt{mean}(v)$ performs better than other candidates. The blue curve does not show because the algorithm diverges and the curve is out of range.

C.3 More Results on the Scaling Law Experiments

The complete loss curves of Llama 2-1B. We here present the complete validation loss curve of Llama 2-1B, training on 20B tokens, which corresponds to the rightmost curve in the scaling law

experiments in Figure 12 (a). We note that this is a complete pre-training run under the definition of Chinchila's law (Hoffmann et al., 2022). We find that Adam-mini's loss curves closely resemble those of AdamW.

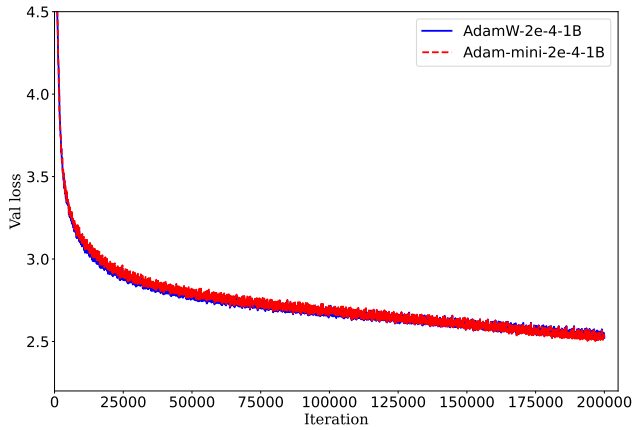


Figure 16: Loss curves of pre-training Llama 2-1B on 26B tokens. This is a complete pre-training run under the definition Chinchila's law. We find that Adam-mini performs similarly to AdamW throughout the training, but with 50% less memory.

The final validation perplexity. In Table 5, we present the final validation perplexity for all models after training on the token amount suggested by Chinchilla's law (Hoffmann et al., 2022). For all models from 39M to 1B, we find that Adam-mini reaches a lower validation perplexity than AdamW.

Table 5: Final validation perplexity for all models pre-trained by Adam and Adam-mini. The token amount follows Chinchila's law. After pre-training, Adam-mini reaches a lower validation perplexity than AdamW.

Model size	39M	67M	102M	162M	271M	1B
Total tokens	1.02B	1.76B	2.67B	4.25B	7.10B	26.21B
AdamW	40.795	29.319	24.670	20.360	17.178	12.452
Adam-mini	**40.407**	**29.014**	**24.192**	**20.172**	**17.035**	**12.372**

C.4 Non-LLM Tasks

We now evaluate Adam-mini on non-LLM tasks. Table 6 shows the results for training ResNet18, Swin-Transformer, DiT-XL-2, DC-AE-Diffusion on ImageNet, DDPM diffusion model on CelebA, a Graph Convolution Net (GCN), and a Graph Attention Net (GAT) on OGB-arxiv. The training curves are shown in Figure 17 and 18. We find the performance of Adam-mini to be comparable or better than AdamW, but with less memory.

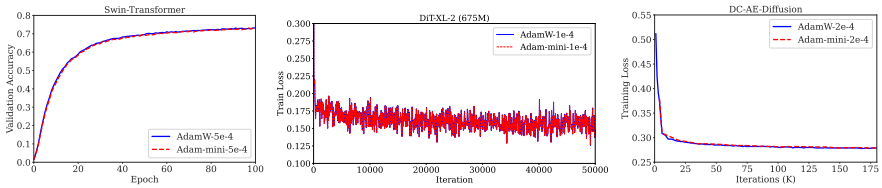
Table 6: On popular non-LLM tasks, Adam-mini performs on par or better than AdamW.

Domain	Model	Optimizer	Metric	25% steps	50% steps	75% steps	100% steps
Vision	DDPM	AdamW	Train loss (↓)	0.0529	0.0497	0.0420	0.0394
Vision	DDPM	Adam-mini	Train loss (↓)	**0.0525**	**0.0495**	**0.0416**	**0.0388**
Vision	ResNet18	AdamW	Val acc (↑)	0.6149	0.6478	0.6613	0.6669
Vision	ResNet18	Adam-mini	Val acc (↑)	0.6140	**0.6501**	**0.6629**	0.6667
Vision	Swin-Transformer	AdamW	Val acc (↑)	**0.6290**	0.6940	**0.7180**	**0.7310**
Vision	Swin-Transformer	Adam-mini	Val acc (↑)	0.6230	**0.6960**	0.7160	0.7300
Vision	DiT-XL-2	AdamW	Train loss (↓)	0.1605	0.1696	0.1607	0.1431
Vision	DiT-XL-2	Adam-mini	Train loss (↓)	**0.1601**	**0.1693**	**0.1605**	**0.1430**
Vision	DC-AE-Diffusion	AdamW	Train loss (↓)	0.2860	**0.2820**	0.2800	0.2780
Vision	DC-AE-Diffusion	Adam-mini	Train loss (↓)	**0.2860**	0.2830	**0.2800**	**0.2780**
Graph	GAT	AdamW	Val acc(↑)	0.7277	0.7367	0.7399	0.7421
Graph	GAT	Adam-mini	Val acc (↑)	**0.7378**	**0.7394**	**0.7403**	**0.7429**
Graph	GCN	AdamW	Val acc (↑)	0.7347	0.7428	0.7379	0.7374
Graph	GCN	Adam-mini	Val acc (↑)	**0.7406**	0.7427	**0.7380**	**0.7423**

In Table 7, we further evaluate the image quality from the model trained by Adam-mini. We find that the Adam-mini performs on par with AdamW.

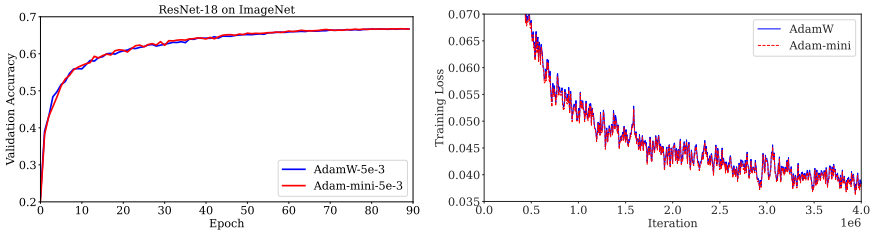
Table 7: Evaluation scores: Adam-mini performs on par with AdamW.

Domain	Model	Optimizer	FID (↓)	Inception Score (↑)
Vision	DiT-XL-2	AdamW	91.83	12.38
Vision	DiT-XL-2	Adam-mini	**88.20**	**13.90**
Vision	DC-AE-Diffusion	AdamW	34.72	41.79
Vision	DC-AE-Diffusion	Adam-mini	**33.15**	**44.38**



(a) Swin-Transformer (b) DiT-XL-2 (c) DC-AE-Diffusion

Figure 17: The training curves of Swin-Transformer, DiT-XL-2, and DC-AE-Diffusion. We find that Adam-mini performs on par with AdamW



(a) ResNet-18 (b) DDPM

Figure 18: The training curves of ResNet-18 and DDPM diffusion model. We find that Adam-mini performs on par with AdamW.

C.5 More Discussions on the Partition Strategies of Value

As shown in Figure 7, the Hessian structure of `value` is less clear compared to other blocks: it shows the hint of 16 diagonal blocks (where 16 is the number of output neurons), but the pattern is not that clear. This gives rise to two potential partition strategies: (I) partition by output neuron; (II) treat as a whole. Numerically, we find that strategy (I) works well when the number of total training steps is large. This includes most of our experiments such as GPT-2 in Figure 9 (with more than 50k total steps) and the scaling law experiments of Llama models in Figure 12 (e.g., Llama 2-1B is trained with more than 200k total steps). On the other hand, we find that strategy (II) works better when the number of total training steps is small. This includes our Llama experiments with 10k total steps in Figure 10.

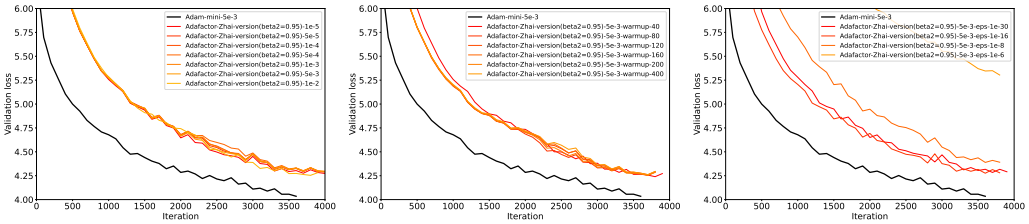
Based on these findings, we recommend using strategy (I) when the total number of training steps is large, and using strategy (II) if otherwise. Note that strategy (II) can be used simply by adding one line of code after creating the optimizer: `optimizer.wv_names = {}`.

C.6 Detailed Comparison with Adafactor

In this section, we conduct a more hyperparameter search for Adafactor on Llama 2-20M pre-training. We will focus on tuning Adafactor-Zhai-version since it performs better than the original Adafactor (see Figure 13). We consider the following three setups.

- **Setup 1:** We change the default $\beta_2 = 0.999$ to $\beta_2 = 0.95$ and sweep over learning rates.

- **Setup 2:** We use learning rate = 5e-3, $\beta_2 = 0.95$ and sweep over warm-up step = $\{1\%, 2\%, 3\%, 4\%, 5\%, 10\%\}$ total steps.
- **Setup 3:** We use learning rate = 5e-3, $\beta_2 = 0.95$ and warm-up step = 1% total steps and sweep over $\epsilon = \{10^{-30}, 10^{-16}, 10^{-8}, 10^{-6}\}$.



(a) Llama 2-20M (b) Llama 2-20M (c) Llama 2-20M

Figure 19: The training curves of Adafactor-Zhai-version on Llama 2-20M pre-training. (a,b,c) corresponds to the aforementioned **Setup 1, 2, 3**, respectively. We find that Adafactor consistently underperforms Adam-mini.

The results are shown in Figure 19. In all these cases, Adafactor-Zhai-version consistently underperforms Adam-mini and the change of hyperparameters does not help much.

C.7 Detailed Comparison with Lion

We now conduct the hyperparameter grid search over Lion. We find that Lion is not easy to tune and we have *not* managed to make Lion work. We consider the following settings.

Tuning strategies in (authors, 2024). authors (2024) carefully tune Lion on Llama models (150M, 300M, 600M, 1.2B). We will adopt their optimal tuning strategies in (authors, 2024, Table 1) [5]. We here summarize their key messages.

- **Message 1:** The optimal learning rate (lr) of Lion is usually 10 times smaller than AdamW.
- **Message 2:** The magical number lr = 3.16e-4 works the best for most models (Llama 150M, 300M, 600M).
- **Message 3:** $\beta_1 = \{0.95, 0.9\}$ perform similarly and perform significantly better than other β_1 candidates including $\beta_1 = \{0.99, 0.98, 0.8, 0.5, 0\}$.
- **Message 4:** $\beta_2 = \{0.99, 0.98, 0.95\}$ perform similarly and perform significantly better than other β_2 candidates including $\beta_2 = \{0.9999, 0.999, 0.995, 0.9, 0.8\}$.

In the following, we will use the above messages to tune the hyperparameters of Lion.

Architecture. We consider Llama 2-20M, which is the same architecture as the ones investigated in (authors, 2024), but with different model size. We also consider GPT-2-125M, which is a task that Lion is not tested before (neither in (authors, 2024) nor in other literatures to our knowledge).

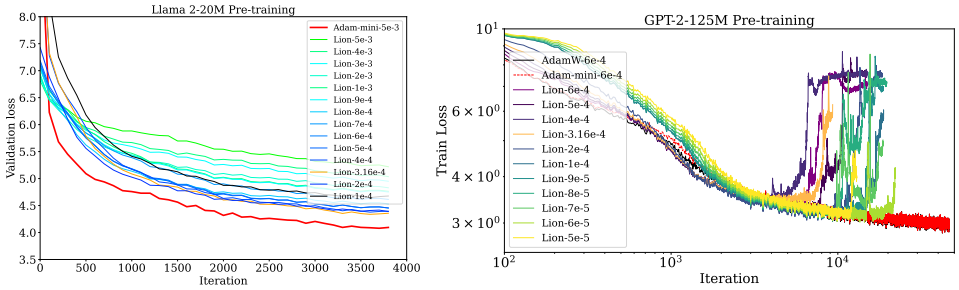
Our tuning strategies. Following the above **Message 1 and 2** from (authors, 2024), we will use the following tuning strategies for Lion on Llama 2-20M and GPT-2-125M.

- **Learning rate for Llama 2-20M:** The standard lr is 5e-3, so we try lr = [5e-4, 6e-4, 7e-4, 8e-4, 9e-4, 1e-3, 2e-3, 3e-3, 4e-3, 5e-3]. For completeness, we also investigate lr = [4e-4, 3.16e-4, 2e-4, 1e-4].
- **Learning rate for GPT-2-125M:** The standard lr is 6e-4, so we try lr = [6e-5, 7e-5, 8e-5, 9e-5, 1e-4, 2e-4, 0.000316, 4e-4, 5e-4, 6e-4]

As for (β_1, β_2), we will use $(\beta_1, \beta_2) = (0.95, 0.98)$. We use these hyperparameters for two reasons. First, they are the optimal choice among other candidates by **Message 3 and 4**. Second, $(\beta_1, \beta_2) =$

[5] We would like to mention that (authors, 2024) is a concurrent work to us, and their tuning strategies in is not public available by the time we submitted this script.

$(0.95, 0.98)$ is recommended by the authors of Lion to be "helpful in mitigating instability during training" [6].



(a) Llama 2-20M (b) GPT-2-125M

Figure 20: The training curves of Lion on Llama 2-20M and GPT-2-125M pre-training. The hyperparameters are chosen under the optimal strategies by (authors, 2024). We find that Lion consistently underperforms Adam-mini on Llama 2-20M, and it encounters loss spikes on GPT-2-125M.

The results are shown in Figure 20. After using the above tuning strategies, we find that **Lion still underperforms Adam-mini and AdamW** on Llama 2-20M and GPT-2-125M. In particular, Lion **encounters loss spikes** on GPT-2-125M for all the learning rate candidates above.

We summarize our findings on Lion below.

- **First: worse performance.** With all the effort above, we haven't managed to make Lion work, and we haven't been able to reproduce (authors, 2024) on Llama 2-20M and GPT-2-125M (different model size and architectures from (authors, 2024)). One possible reason is that Lion might work under their specific setup (dataset, architecture, batch size, etc.), but the effectiveness is not easily transferable.
- **Second: no general tuning guidance.** We find that there are no general tuning strategies for Lion. We emphasize that authors (2024) only focuses on Llama architectures, and their resulting tuning strategy seems not robust and transferable to other architectures. In particular, their optimal strategy on Llama causes loss spikes on GPT-2. To our knowledge, Table 1 in (authors, 2024) is the only public tuning strategy for Lion, so it seems unclear how to tune Lion in general.

  In contrast, Adam-mini is much easier to use. In all our experiments (a wide range of tasks and models), Adam-mini performs on par with Adam **using the same hyperparameters as AdamW** (including learning rate, β_1, β_2, ϵ, etc.). We believe that "easy adaptation of the hyperparameters" can serve as one advantage of Adam-mini over Lion, apart from the performance superiority.
- **Third: Adam-mini is more principled and explainable.** We emphasize that Lion is designed by symbolic search, and its design principle is largely unclear. In contrast, the Design principle of Adam-mini is much more understandable: we remove the redundant lrs in Adam according to the Hessian structure. We believe Adam-mini is more "white-box" than Lion and more explainable to users.

C.8 Sample Responses from LLMs trained by Adam-mini

The sample response in shown in Figure 21.

D Some Preliminary Results

D.1 Preliminaries on Adam, AdamW and LAMB

We introduce Adam, AdamW, and LAMB in **Algorithm 5**, **6**, and **7**. These methods need to track m and v along the training. Both m and v are vectors of the same size as # model parameter.

[6] https://github.com/lucidrains/lion-pytorch

Prompt: Compose an engaging travel blog post about a recent trip to Hawaii, highlighting cultural experiences and must-see attractions.

AdamW: Sure! Here's a travel blog post about a recent trip to Hawaii: ..... Whether you're interested in history, nature, or simply soaking up the sun, Hawaii has something for everyone. Until next time, Aloha!

Adam-mini: Sure! Here's a travel blog post about a recent trip to Hawaii: ..... I highly recommend visiting this beautiful archipelago for a truly unforgettable adventure.

Prompt: Rewrite your previous response. Start every sentence with the letter A.

AdamW: A trip to Hawaii is a must-visit destination for any traveler, offering a variety of cultural experiences and must-see attractions. From the Polynesian Cultural Center to the USS Arizona Memorial, and from Lanikai Beach to the lively night markets, every corner of the islands reveals a new surprise and a new way to connect with the spirit of Hawaii.

Adam-mini: A trip to Hawaii is a must-do for any travel enthusiast. A trip to the beautiful Hawaiian Islands is a paradise on earth. A visit to Oahu is a great way to start your adventure. A helicopter tour of the island is both educational and breathtaking. A visit to Maui is a must-see. A visit to Haleakala National Park is a highlight of your trip. A visit to the Polynesian Cultural Center is a great way to learn about the history and traditions of the islands. A traditional luau is a must-attend event. Overall, a trip to Hawaii is an unforgettable experience.

Figure 21: Response Sample of Llama 2-7B finedtuned by AdamW and Adam-mini to the #81 test question from the MT-Bench (Zheng et al., 2024) dataset. Since there is little information in the first round response, we omit the answers from the models.

Algorithm 5 Adam in Pytorch style

Let λ be the weight decay coefficient
for `param` in `parameter_blocks` **do**
  $\mathtt{g} = \mathtt{param.grad}$
  if $\lambda > 0$ **then**
    $\mathtt{g} = \mathtt{g} + \lambda * \mathtt{param}$
  end if
  $\mathtt{param} = \mathtt{param} - \eta_t * \lambda * \mathtt{g}$
  $\mathtt{m} = (1-\beta_1) * \mathtt{g} + \beta_1 * \mathtt{m}$
  $\hat{\mathtt{m}} = \frac{\mathtt{m}}{1-\beta_1^t}$
  $\mathtt{v} = (1-\beta_2) * \mathtt{g} \odot \mathtt{g} + \beta_2 * \mathtt{v}$
  $\hat{\mathtt{v}} = \frac{\mathtt{v}}{1-\beta_2^t}$
  $\mathtt{param} = \mathtt{param} - \eta_t * \frac{\hat{\mathtt{m}}}{\sqrt{\hat{\mathtt{v}}}+\epsilon}$
end for

Algorithm 6 AdamW in Pytorch style

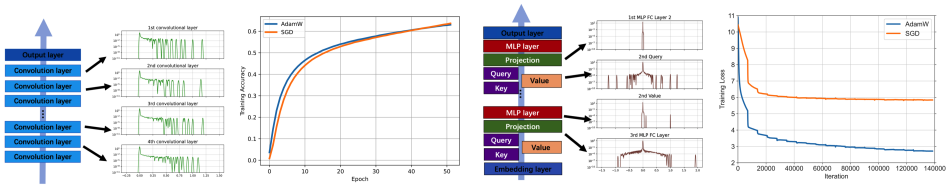
Let λ be the weight decay coefficient
for `param` in `parameter_blocks` **do**
  $\mathtt{g} = \mathtt{param.grad}$
  $\mathtt{param} = \mathtt{param} - \eta_t * \lambda * \mathtt{g}$
  $\mathtt{m} = (1-\beta_1) * \mathtt{g} + \beta_1 * \mathtt{m}$
  $\hat{\mathtt{m}} = \frac{\mathtt{m}}{1-\beta_1^t}$
  $\mathtt{v} = (1-\beta_2) * \mathtt{g} \odot \mathtt{g} + \beta_2 * \mathtt{v}$
  $\hat{\mathtt{v}} = \frac{\mathtt{v}}{1-\beta_2^t}$
  $\mathtt{param} = \mathtt{param} - \eta_t * \frac{\hat{\mathtt{m}}}{\sqrt{\hat{\mathtt{v}}}+\epsilon}$
end for

Algorithm 7 LAMB in Pytorch style

Let λ be the weight decay coefficient, let ϕ be a scaling function.
for `param` in `all_layers` **do**
  `g` = `param.grad`
  `param` = `param` - $\eta_t * \lambda *$ `g`
  $\mathtt{m} = (1-\beta_1) * \mathtt{g} + \beta_1 * \mathtt{m}$
  $\hat{\mathtt{m}} = \frac{\mathtt{m}}{1-\beta_1^t}$
  $\mathtt{v} = (1-\beta_2) * \mathtt{g} \odot \mathtt{g} + \beta_2 * \mathtt{v}$
  $\hat{\mathtt{v}} = \frac{\mathtt{v}}{1-\beta_2^t}$
  $\mathtt{r} = \frac{\hat{\mathtt{m}}}{\sqrt{\hat{\mathtt{v}}}+\epsilon}$
  `param` = `param` - $\eta_t * \frac{\phi(\|\mathtt{param}\|)}{\|r+\lambda * \mathtt{param}\|} * \mathtt{r}$
end for

D.2 Preliminary results in (Zhang et al., 2024)

We here restate (Zhang et al., 2024, Figure 3). This figure shows that: for Transformers, different parameter blocks have different Hessian eigenvalue distributions, while for CNNs, the eigenvalue distributions are similar among blocks. This suggests that Transformers need different learning rates for different blocks to handle the heterogeneity in eigenvalue distributions.



(a) VGG16 (b) VGG16 (c) BERT (d) BERT

(a) VGG16 (b) BERT

Figure 22: Figure 3 in (Zhang et al., 2024). The eigenvalues distribution are similar among blocks for CNNs, while they differ significantly across blocks for Transformers. This indicates Transformers need different learning rates for different blocks to handle the heterogeneity in eigenvalues.

E Experimental Details

E.1 Training configurations for Section 3

Unless mentioned otherwise, we choose the model configurations by their standard protocols. We choose the learning rates by the recommendation from open-source platforms if applicable. For instance, for GPT2 series, we use the recommended learning rates by (Liu et al., 2023), which are reported to be optimal by grid search. Unless mentioned otherwise, Adam-mini, Adafactor, CAME, SM3, and LAMB use the same learning rate as the recommended ones of AdamW. If there is no public recommended learning rate for AdamW, we tune the learning rate for all optimizers within the same computational budget and report the best performance. For other hyperparameters, we follow the recommendation from open-source platforms or by their default setting. For SM3 and Adafactor, we incorporate momentum with $\beta_1 = 0.9$ to offer a fair comparison with other optimizers and the rest of the hyperparameters are set as default. The detailed configurations are explained as follows.

GPT2 pre-training. We use the nanoGPT codebase[7] to train GPT2 sized 125M (small), 330M (medium), and 1.5B (XL) on Openwebtext. For all models, we use `seq_len` $= 1024$, batch size = 480, weight decay coefficient $\lambda = 0.1$, $\epsilon =$ 1e-8, $\beta_1 = 0.9, \beta_2 = 0.95$. We use cosine-decay learning rate schedule with 2000 iterations of warm-up. For GPT2-small and medium, we use the

[7] `https://github.com/karpathy/nanoGPT/tree/master`

recommended peak learning rate by (Liu et al., 2023), which are reported to be the optimal ones found by grid search. For GPT2-XL, we use the recommended peak learning rate by the Levanter[8]. The chosen peak learning rates are 6e-4, 3e-4, 1e-4 for GPT2-small, medium, XL, respectively. The minimal learning rate is chosen as 3e-5, 6e-5, 1e-5 for these models.

Llama pre-training. For all experiments on the Llama series (from 20M to 13B), we use the Torchtitan codebase[9] and C4 dataset (Raffel et al., 2020). For all experiments, we use weight decay coefficient $\lambda = 0.1$, $\epsilon =$ 1e-8, $\beta_1 = 0.9, \beta_2 = 0.95$. For Llama 2-1B, Llama 3-8B, we use learning rate = 3e-4. For Llama 2-13B, we use learning rate = 1e-4. As for the learning rate schedule, we use warm-up step = 1% total step and use linear decay schedule after the warm-up (this is the default setting in the Torchtitan codebase). For Figure 10 (a) and all the experiments of Adafactor and Lion, we use `seq_len` $=$ 512 and batch size $=$ 128. For Figure 10 (b), we use `seq_len` $=$ 2048 and batch size $=$ 8. For Figure 10 (b), we shrink the batch size due to the limited hardware. For all the scaling law experiments, we use `seq_len` $=$ 512 and batch size $=$ 256. We summarize the detailed setups for the scaling law experiments in later paragraphs.

SFT and RLHF. We use the Llama 2-7B pretrained model (Touvron et al., 2023) for our study. We use the `ultrafeedback` dataset [10]. The implementation of SFT and RLHF code is based on the ReMax codebase[11]. Specifically, we train a SFT model with 40% of the chosen data and train a reward model using the remaining 60%. Then, we apply the reinforcement learning algorithm ReMax (Li et al., 2023), a memory-efficient alternative to PPO (Schulman et al., 2017), to optimize the preference reward.

We use DeepSpeed ZeRO-2 in our training. GPT-4 evaluation template in Table 3 is from the codebase[12]. In the reward optimization stage, We use ReMax, a memory-efficient alternative to PPO. We use UltraFeedback dataset Cui et al. (2023) and use 40% data for SFT and 60% data for ReMax.

SFT. We use 80 samples in a batch and train the model for 3 epochs. For the full parameter tuning, we search the learning rate from {1e-6, 2e-6, 3e-6, 4e-6, 5e-6, 1e-5, 2e-5} based on validation loss, and we use 2e-6 with cosine annealing for both AdamW and Adam-mini. For LoRA, We apply LoRA for all layers except the embedding layer. The rank of LoRA is set to 128. After selecting the learning rate from the same set as the full parameter tuning, we use 2e-5 for both AdamW and Adam-mini when LoRA is applied. The weight decay coefficient is set to 0 as recommended by LlamaFactory[13]. The rest of the hyperparameters of AdamW and Adam-mini are $\epsilon =$ 1e-8, $\beta_1 = 0.9,\ \beta_2 = 0.95$.

ReMax. We use 48 samples in a batch and train the model for 1 epoch. By searching the peak learning rate from {5e-7, 1e-6, 2e-6} based on validation reward, AdamW uses 1e-6 while Adam-mini selects 5e-7 as the peak learning rate. The weight decay coefficient is set to 0. The rest of the hyperparameters of AdamW and Adam-mini are $\epsilon =$ 1e-8, $\beta_1 = 0.9,\ \beta_2 = 0.95$.

ResNet. We use the PyTorch official implementation codebase[14] to train ResNet18 (He et al., 2016) on ImageNet (Deng et al., 2009). We use cosine-decay learning rate, epoch =90, $\beta_1 = 0.9, \beta_2 = 0.999, \epsilon =$1e-8. For ResNet18, we use batch size = 256, peak learning rate = 0.005. For ViT-base, we use batch size = 128, peak learning rate = 0.0001. These configurations are used for both Adam-mini and AdamW.

Diffusion models. We use the codebase[15] to train diffusion models. The image size is 64 and the training objective is to predict the noise as in (Ho et al., 2020). We use the default U-Net archiecture

[8] https://github.com/stanford-crfm/levanter/blob/e183ec80ec5971b12d4a3fb08a160268de342670/config/gpt2_xl.yaml

[9] https://github.com/pytorch/torchtitan

[10] https://huggingface.co/datasets/argilla/ultrafeedback-binarized-preferences-cleaned

[11] https://github.com/liziniu/ReMax

[12] https://github.com/lm-sys/FastChat/tree/main/fastchat/llm_judge

[13] https://github.com/hiyouga/LLaMA-Factory

[14] https://github.com/pytorch/examples/blob/main/imagenet/main.py

[15] https://github.com/lucidrains/denoising-diffusion-pytorch

hyper-parameters and the dimension multiply in U-Net is (1, 2, 4, 8). We use the CelebA dataset[16] and train the diffusion model with a learning rate 5×10^{-5} with cosine decay. The batch size is 128 and the training epoch is 50.

Graph Neural Networks. We use the DGL implementation[17] of Graph Convolution Networks (GCN) (Kipf & Welling, 2016) and Graph Attention Networks (GAT) (Velickovic et al., 2017) for OGBN-arxiv[18] dataset. All configurations as default. For both Adam-mini and AdamW, we use the default learning rate = 0.005 for GCN and the default learning rate = 0.002 for GAT.

Scaling law experiments. We use the codebase Torchtitan[19] to train Llama models of different sizes. All the model configurations are shown in Table 8 and all the training configurations are shown in Table 9. The experimental setups are inspired by (Hägele et al., 2024). In all experiments, we fix the warm-up steps to be 1% of the total steps, as suggested by (Ibrahim et al., 2024).

Model Size	d_{model}	n_{layers}	n_{heads}	seq_len
39M	384	8	6	512
67M	512	10	8	512
102M	640	12	10	512
162M	768	16	12	512
271M	1024	16	16	512
1B	2048	18	16	512

Table 8: The model configurations in the scaling law experiments.

Model	LR	Batch size (# tokens)	Steps	Tokens	Token/Params Ratio
39M	6e-4	0.13M	7.8K	1.02B	26.15
67M	6e-4	0.13M	13.4K	1.76B	26.27
102M	6e-4	0.13M	20.4K	2.67B	26.17
162M	6e-4	0.13M	32.4K	4.25B	26.23
271M	6e-4	0.13M	54.2K	7.10B	26.21
1B	2e-4	0.13M	200K	26.21B	26.21

Table 9: Training configurations for the scaling law experiments.

Trajectory comparison in Figure 10 (c). We train a 8-layer Transformer sized 11M on Openwebtext and launch AdamW, Adam-mini, and other memory-efficient optimizers under the same random seed and same learning rate 1e-5. We save the model weights for every 250 iterations and compare their Euclidean distance to the weights along AdamW's trajectory.

E.2 Detailed Setup for Other Experiments

Configurations for Figure 3. We use a synthetic binary classification dataset with 100 samples using the data generation process as shown below. We use a 1-hidden-layer network with an input dimension of 64, and a width of 16, and with `Tanh` activation function. We train the model for 500 steps using Adam with learning rate of 1e-4, and the model reaches 100% classification accuracy. With the help of auto-differentiation framework, we calculate the Hessian with two passes of backpropagation (Pearlmutter, 1994) and the calculation is exact.

```
def generate_data(n_samples_per_class, n_classes, input_dim):
    # Generate synthetic data for specified dimensions
    X = []
```

[16] https://cseweb.ucsd.edu/~weijian/static/datasets/celeba/
[17] https://github.com/dmlc/dgl/tree/master/examples/pytorch/ogb/ogbn-arxiv
[18] https://ogb.stanford.edu/docs/nodeprop/
[19] https://github.com/pytorch/torchtitan

```
    y = []
    for i in range(n_classes):
        center = np.random.rand(input_dim) * 10  # Random class center
        class_samples = np.random.randn(n_samples_per_class, input_dim) *
 0.5 + center  # Add some noise
        X.append(class_samples)
        y.extend([i] * n_samples_per_class)

    X = np.vstack(X)  # Combine all class samples
    y = np.array(y)    # Convert labels to a NumPy array
    return X, y
```

Configurations for Figure 4. For each dense sub-block $H_l, l = 1, 2, 3$, we use random positive definite matrices. We fix the choose the eigenvalues of each H_l as follows: for $l = 1$, we independently sample from {1,2,3} for 30 times; for $l = 2$, we repeat this procedure for {99,100,101}; for $l = 3$, we repeat this procedure for {4998 ,4999, 5000}. For the single (blockwise) learning rate method, we use GD with optimal constant learning rate $2/(L + \mu)$, where L, μ are the largest and smallest eigenvalue of the (blockwise) Hessian. We use Adam with $\beta_1 = 0$. This helps us focus on the effect of coordinatewise learning rate in Adam. We also set $\beta_2 = 1$ to the time-varying learning rate. This is necessary because, for any $\beta_2 < 1$, Adam with constant learning rate will oscillate on quadratic functions. This is theoretically proved in (Da Silva & Gazeau, 2020, Proposition 12, Figure 1) and empirically observed in (Zhang et al., 2024, Section 3.3).

Configurations for Figure 5. To generate a positive definite matrix H_b, we first uniformly sample $\frac{d(d-1)}{2}$ independent angles $\theta_{i,j}$ from the interval $[-\frac{\pi}{2}, \frac{\pi}{2}]$, where $i < j$. Starting with the identity matrix, we perform a rotation of the i-th and j-th rows by the angle $\theta_{i,j}$ for each sampled pair. Through $\frac{d(d-1)}{2}$ rotation operations, we obtain the orthogonal matrix Q. We define $\Lambda = \text{diag}(\kappa, 1, \ldots, 1)$, and the matrix H_b is generated using the expression $H_b = Q\Lambda Q^T$. The python code for H_b generation is listed as follows:

```
def generate_Hb(theta, kappa, d):
    Q = np.eye(d)
    for i in range(d):
        for j in range(i+1,d):
            P = np.eye(d)
            P[i,i] = math.cos(theta[i,j])
            P[i,j] = math.sin(theta[i,j])
            P[j,i] = -math.sin(theta[i,j])
            P[j,j] = math.cos(theta[i,j])
            Q = P @ Q
    Lambda = np.eye(d)
    Lambda[0,0] = kappa
    return Q @ Lambda @ Q.transpose()
```

We note that as θ approaches 0, the diagonal-over-off-diagonal ratio of the matrix Q decreases. For the sampled values of θ, we utilize $R\theta$ to produce H_b with varying ratios, where $R \in \{\frac{k}{50} | k = 0, 1, \ldots, 50\}$. For each matrix, we sample 100 initial points from the Xavier initialization distribution to compute the resulting κ of Adam algorithm. For each pair of d and κ, we sample 40 different θ values. By averaging the results obtained, we plot the Figure 5.

Configurations for Figure 7. We use the nanoGPT codebase and Openwebtext dataset. We consider a 1-layer Transfomer with n_emb = 16, n_head = 4, and the width (i.e., the number of output neuron) of `mlp.fc_1` equals 32.

Throughput Comparison 2. The results on throughput are tested on 2× A800-80GB GPUs. We did not turn on CPU offload. We report the throughput from the summary file of the Wandb log.