HiLD 2025: 3rd Workshop on High-dimensional Learning Dynamics

When Can You Get Away with Low Memory Adam?

Dayal Singh Kalra DAYAL@UMD.EDU
University of Maryland, College Park

John Kirchenbauer JKIRCHEN@UMD.EDU
University of Maryland, College Park

Maissam Barkeshli MAISSAM@UMD.EDU
University of Maryland, College Park

Tom Goldstein TOMG@UMD.EDU
University of Maryland, College Park

Abstract

Adam is the go-to optimizer for training modern machine learning models, but it requires additional memory to maintain the moving averages of the gradients and their squares. While various low-memory optimizers have been proposed that sometimes match Adam's performance, their lack of reliability has left Adam as the default choice. In this work, we apply a simple layer-wise Signal-to-Noise Ratio (SNR) analysis to quantify when second-moment tensors can be effectively replaced by their means across different dimensions. Our SNR analysis reveals how architecture, training hyperparameters, and dataset properties impact compressibility along Adam's trajectory, naturally leading to *SlimAdam*, a memory-efficient Adam variant. *SlimAdam* compresses second moments along dimensions with high SNR when feasible, and leaves when compression would be detrimental.

1. Introduction

Adam with weight decay [15] has become the standard optimizer choice in modern machine learning, consistently outperforming non-adaptive optimizers such as Stochastic Gradient Descent with momentum (SGD-M). Its success is typically attributed to adapting to the geometry of the landscape by estimating the "effective learning rate" for each parameter using a moving average of the squared gradients. An additional benefit of this adaptive mechanism is that the optimal learning rate is less sensitive to training recipe changes. While these factors make Adam the go-to optimizer for training language models, it stores a moving average of squared gradients beyond SGD-M, doubling the optimizer's memory footprint. This memory cost becomes particularly crucial in resource-limited settings, where the memory allocated to the optimizer states could be used for the model parameters or activations.

To avoid the extra memory footprint of Adam, various low-memory optimizers have been proposed [2, 8, 18, 23]. These optimizers are a free lunch in some settings – slashing memory usage with no detectable loss in performance [29, 30] – but they compromise performance in others [16] (for a detailed discussion on related works, see Appendix A). While the potential benefits of low-memory optimizers are clear, a lack of understanding as to when they will perform well is a major barrier to widespread adoption. In this work, we examine when Adam's second moments can be replaced by their mean during training. Our goal here is to develop a principled framework to help users understand and quantify

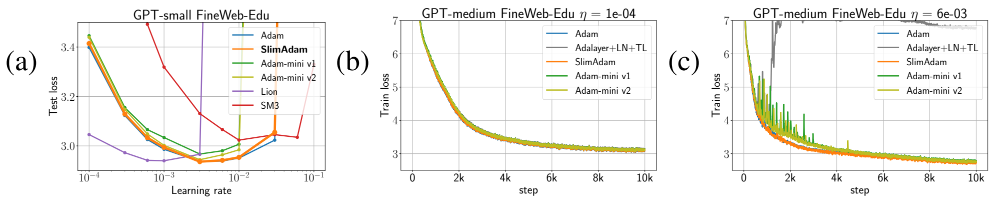


Figure 1: Comparison of low-memory optimizers on GPT pre-training with Fineweb-Edu. (a) *SlimAdam* matches Adam's U-shaped loss curve. (b) All Adam variants show identical curves at small learning rates. (c) At large learning rates, *SlimAdam* maintains Adam's stability while other variants become unstable.

when these low-memory variants of Adam are appropriate for their problem, thereby improving the reliability of low-memory optimizers and providing deeper insights into Adam's dynamics.

We propose and study a simple measure of the compressibility of Adam's second-moment memory. By examining the Signal-to-Noise Ratio (SNR) of the second moment tensor in each layer, we quantify when individual second moments can be effectively replaced by their means across specific dimensions (such as fan$_{\text{in}}$, fan$_{\text{out}}$, or both dimensions). Our SNR-based metrics reveal that layers exhibit varying degrees of compressibility across different dimensions, and this compressibility can depend strongly on the architecture, training hyperparameters, and dataset properties. While several layer types show consistent compressibility patterns across training configurations, we also observe that some layer types show varying compression trends.

Our SNR analysis naturally leads to *SlimAdam*, a memory-efficient Adam variant that compresses the second moments along the most efficient dimensions, or selectively leaves layers uncompressed when needed to maintain stability. By taking an adaptive approach to compression, *SlimAdam* matches Adam's performance while maintaining stability at large learning ratesas shown in Figure 1. For instance, *SlimAdam* saves 98% of second moments in a $\sim 124M$ parameter GPT-style Transformer trained on language tasks. Our investigation also reveals a surprising property of Adam: it uses significantly more second moments at large learning rates than required for optimal performance. For instance, in GPT-style Transformers trained on language modeling, while the SNR analysis suggests that $\sim 35\%$ of second moments could be compressed at Adam's optimal learning rate, *SlimAdam* achieves Adam's performance while compressing 98% of them. This intriguing finding suggests that the majority of Adam's per-parameter adaptivity isn't necessarily required for optimal training.

2. Notations and Preliminaries

For a weight matrix $W \in \mathbb{R}^{\text{fan}_{\text{out}} \times \text{fan}_{\text{in}}}$, let G_t denote its gradient at step t. Adam updates these weights using learning rate η_t and the moving averages of the first two moments of gradients, M_t and V_t, with coefficients β_1 and β_2. The equations governing the updates are: $M_{t+1} = \beta_1 M_t + (1-\beta_1) G_t$, $V_{t+1} = \beta_2 V_t + (1-\beta_2) G_t^2$, and $W_{t+1} = W_t - \eta_t {}^{\hat{M}_{t+1}}/_{\sqrt{\hat{V}_{t+1}}+\epsilon}$. Here, $\hat{M}_t = {}^{M_t}/_{1-\beta_1^t}$ and $\hat{V}_t = {}^{V_t}/_{1-\beta_2^t}$ are the bias-corrected moments and ϵ is a small scalar used for numerical stability.

For our analysis, we generalize Adam to a family of low-memory variants parameterized by layer-specific sharing dimensions. For each layer, we compute an estimate of the second moments by averaging squared gradients across specified dimensions K (fan$_{\text{in}}$, fan$_{\text{out}}$, or both). The difference compared to Adam lies in the second moment update:

$$V_{t+1} = \beta_2 V_t + (1-\beta_2) \mathbb{E}_K \left[G_t^2 \right], \tag{1}$$

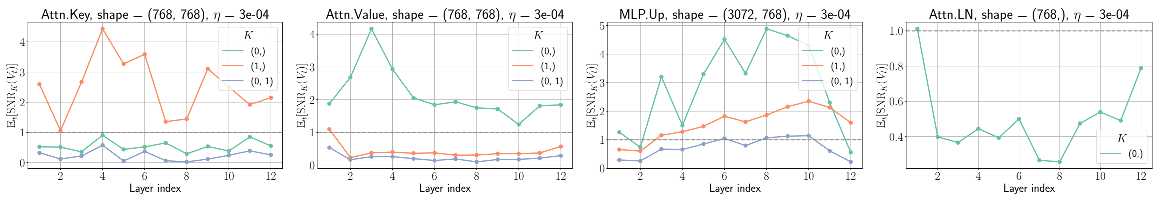


Figure 2: Depth dependence of average SNR values of selected second-moment blocks of the GPT-small model trained on OpenWebText.

where $\mathbb{E}_K[\cdot]$ denotes an average over dimensions K. Since Adam's second moment acts as a per-parameter "effective" learning rate, averaging these moments along dimensions K is equivalent to sharing a common learning rate. The above optimizer coincides with Adam when $K = \varnothing$. Throughout this work, we partition second moments using the default parameter partitioning scheme that groups parameters at the granularity of layer components (weights, biases, and attention components). We use $K = 0$ for fan$_{\text{out}}$, $K = 1$ for fan$_{\text{in}}$ and $K = (0, 1)$ to denote sharing along both dimensions.

3. SNR Analysis of Adam's Second Moments

This section analyzes how effectively Adam's per-parameter second moments can be replaced by their mean along different dimensions (such as fan$_{\text{in}}$, fan$_{\text{out}}$, or both) during training. The feasibility of such a compression depends on how tightly the entries are clustered around their mean value. If entries along a dimension exhibit low variance relative to their mean, they can be effectively represented by a single value. To quantify this concentration of values, we analyze the Signal-to-Noise Ratio (SNR) of the second moments during training. For a second moment matrix $V \in \mathbb{R}^{\text{fan}_{\text{out}} \times \text{fan}_{\text{in}}}$ and specified compression dimensions K, SNR$_K$ is defined as: $\text{SNR}_K(V_t) = \mathbb{E}_{K'}\left[\frac{(\mathbb{E}_K[V_t])^2}{\text{Var}_K[V_t]}\right]$, where $\mathbb{E}_K[\cdot]$ and $\text{Var}_K[\cdot]$ compute the mean and variance along the specified dimensions K, while the outer expectation $\mathbb{E}_{K'}[\cdot]$ averages the ratio over the remaining dimensions to obtain a scalar. SNR$_K$ quantifies compression feasibility along dimensions K throughout the trajectory. When $\text{SNR}_K \gtrsim 1$ [1], the signal dominates the noise, indicating entries can be effectively described by their mean, whereas $\text{SNR}_K \lesssim 1$ suggests that individual entries carry significant information that would be lost when the entries are replaced by their mean. This analysis not only suggests layers that can be compressed but also quantifies their relative compression feasibility. As Adam adapts to the local geometry of the loss landscape, SNR values also serve as a proxy for learning complexity during training, with lower SNR suggesting higher complexity and a need for per-parameter effective learning rates. We are interested in cases where it is feasible to replace the second moments by their mean throughout training. To this end, we define average SNR as: $\mathbb{E}_\tau\left[\text{SNR}_K(V_\tau)\right] = \frac{1}{T}\sum_\tau^T \text{SNR}_K(V_\tau)$, where τ indexes the training steps at which SNR is measured and T is the total number of measurements. The averaged SNR quantifies compression feasibility along dimensions K throughout training.

We analyze the evolution of SNR across diverse training configurations (pre-training, fine-tuning, image classification) to uncover fundamental SNR trends. Table 1 summarizes the preferred compressibility dimension by layer type based on these trends. Below, we discuss language pre-training results and defer additional results to Appendix D.

1. For random Gaussian gradients, $\text{SNR}_K > 1/2$ indicates compression feasibility (see Appendix C for a derivation). However, we find that a more stringent value of 1 is more reliable in practice.

Attention key and query second moments consistently show aversion to compression along the fan_{out} dimension, where multiple heads are stacked, suggesting that each attention head requires its own effective learning rate. Ref. [29] reached similar conclusions through an independent Hessian-based analysis, corroborating our findings. On the other hand, attention value and projection second show an opposite trend, with higher compressibility along the fan_{out} dimension as compared to the fan_{in} dimension. For projection layers, aversion to compression along the fan_{in} dimension (where heads are stacked) is intuitive, as the parameters corresponding to each attention head are intended to evolve independently during training. However, for the same reason, the higher compressibility of second moments in the value layer along the head-stacked dimension is unexpected. Intuitions aside, from an absolute magnitude perspective, values and projection layers show higher averaged SNR values along the preferred dimension than keys and queries, indicating greater overall compressibility.

Interestingly, by a similar magnitude argument, MLP second moments exhibit greater compressibility than attention keys and queries. While in general MLP second moments prefer compression along the output dimension (fan_{out}), for some layer indices the second moment can also be compressed along the input dimension (fan_{in}) or even both dimensions simultaneously. LayerNorm and bias components typically exhibit lower compressibility. Due to their minimal contribution to the overall memory, we advise against compressing them.

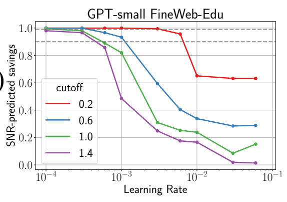
4. Factors Influencing Compressibility

Incompressibility under Heavy-Tailed Distributions: In Appendix G, we investigate how token frequency distribution influences compressibility. We train a simplified two-layer model, solely consisting of a token embedding matrix and a linear head with varying vocabulary sizes. Figure 19 shows that SNR values along the token dimension of both layers decrease substantially as vocabulary size increases, suggesting lower compressibility. We then test the effect of this low compressibility by comparing Adam with shared second moments along the token dimension with standard Adam. Figure 20 shows that the loss gap between these two optimizers increases with vocabulary size.

Large Learning Rates reduce Compressibility: Figure 17 in Appendix E shows that increasing the learning rate consistently reduces SNR values across layers. This decline in averaged SNR values suggests that higher learning rates cause training to explore regions of parameter space where the gradient distribution contains more outliers, thereby reducing compression feasibility. Layers such as Token Embedding/LM Head, LayerNorm, Attention keys, queries, first MLP layer (MLP.Up), averse

Table 1: Summary of preferred compression dimensions by layer type. Compression dimensions marked with * show inconsistent trends across training regimes.

Layer Type	$\mathbf{K}^*$	**Layer Type**	$\mathbf{K}^*$
Attention		*Special Layers*	
Key & Query	fan_{in}	Token Embedding	fan_{out}
Value & Projection	fan_{out}	Language Modeling Head	fan_{in}
MLP Layers		Vision First Layer	fan_{in}
First layer (Up)	$\text{fan}^{\star}_{\text{out}}$	Vision Classification Head	$\text{fan}^{\star}_{\text{in}}$
Middle layer (Gate)	$\text{fan}^{\star}_{\text{out}}$	*Normalization Layers*	–
Last layer (Down)	fan_{out}		

(a)


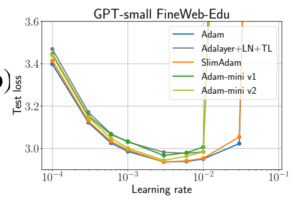
(b)


Figure 3: (a) Fraction of reducible second moments (relative to Adam) across learning rate and SNR cutoff, as predicted by SNR analysis. (b) Performance comparison across learning rates between SlimAdam and baselines: Adam, AdaLayer [30], and Adam-mini [29] (for details, see Appendix A).

compression (SNR $\lesssim 1$) at the optimal learning rate, whereas, attention values and projections and the second MLP layer (MLP.Down) are amenable to compression.

Effect of Initialization on Compressibility: We compare Mitchell initialization [10] used in Section 3 against PyTorch's default initialization scheme. A key feature of Mitchell initialization is that it scales the variance of layers that add to the residual stream (Attn.Proj and MLP.Down) with a factor of $1/\text{depth}$. Figure 18 in Appendix F show that Mitchell initialization leads to higher SNR values compared to the default PyTorch initialization across layers of the GPT-small model. In particular, Attn.Proj and MLP.Down layers show significantly higher SNR values. These exceptionally high SNR values provide empirical support for the $1/\text{depth}$ scaling in Mitchell initialization.

5. DIY: Build Your Own Low-Memory Adam

Building on our comprehensive SNR analysis, we now introduce SlimAdam — a memory-efficient Adam variant that preserves Adam's performance and stability through SNR-guided compression. *SlimAdam* (1) compresses matrix-like second moments along the dimension with the highest SNR when above a threshold and (2) leaves vector-like second moments uncompressed due to their high variability and minimal effect on the overall memory. Our implementation consists of three steps.

Step 1: First, we collect layer-wise SNR statistics using a small proxy model (discussed later).
Step 2:: Next, we identify the compression dimension K^* for each layer type with the highest SNR: $K^* = \arg\max_K \mathbb{E}_\tau[\text{SNR}_K(V^{(l)})]$ and compress a layer only if its SNR is above a given cutoff.
Step 3: Finally, we train the target model using Adam with shared second moments (Equation (1)) along these compression dimensions K^*. The full algorithm is detailed in Appendix H.1.

SNR-predicted compressibility primarily depends on the learning rate used to train the proxy model and the SNR cutoff (Figure 3(a)). GPT models exhibit high compressibility ($\sim 98\%$) at small learning rates, though these savings reduce to $\sim 30\%$ at large learning rates. In theory, we would perform the SNR analysis at the optimal learning rate to determine compression rules, but this approach will only save about 30% of second moments with a cutoff of 1.0. Surprisingly, we find that a more aggressive compression is possible using smaller learning rates. As shown in Figure 3 (b), *SlimAdam* achieves Adam-level performance and stability using compression rules derived at learning rates $10\times$ smaller than optimal, saving $\sim 98\%$ second moments. *SlimAdam*'s success in this setting suggests a previously unreported implicit bias in Adam —it maintains significantly more second moments at large learning rates than required for optimal performance. This also suggests that SNR analysis at small learning rates captures fundamental compression rules while avoiding artifacts that emerge when training Adam at large learning rates.

Depth-averaged SNR $\frac{1}{\text{depth}} \sum_{l=1}^{\text{depth}} \mathbb{E}_\tau[\text{SNR}_K(V^{(l)})]$ yields consistent compression dimensions across model sizes — a 4-layer GPT model with width $n_{\text{embd}} = 256$ yields the same compression dimensions as a 24-layer model with $n_{\text{embd}} = 1024$ (same as in Table 1). This allows using a smaller proxy model to identify compression dimensions for larger target models. Figure 25 in Appendix I verifies that *SlimAdam* with depth-averaged SNR derived rules yield the same performance in GPT pre-training.

6. Discussion

We present a principled SNR framework to analyze when second moments can be effectively replaced with their means, naturally leading to *SlimAdam*, a practical low-memory Adam variant which maintains its performance and stability while saving up to 98% second moments. We hope our work furthers understanding of when low memory optimizers are safe to use in practice while deepening our fundamental understanding of how architecture, training regime, and optimizer design interact.

References

[1] Kwangjun Ahn, Xiang Cheng, Minhak Song, Chulhee Yun, Ali Jadbabaie, and Suvrit Sra. Linear attention is (maybe) all you need (to understand transformer optimization). In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=0uI5415ry7.

[2] Rohan Anil, Vineet Gupta, Tomer Koren, and Yoram Singer. Memory-efficient adaptive optimization. In *NeurIPS 2019*, 2019. URL https://papers.nips.cc/paper_files/paper/2019/hash/8f1fa0193ca2b5d2fa0695827d8270e9-Abstract.html.

[3] Xiangning Chen, Chen Liang, Da Huang, Esteban Real, Kaiyuan Wang, Hieu Pham, Xuanyi Dong, Thang Luong, Cho-Jui Hsieh, Yifeng Lu, and Quoc V Le. Symbolic discovery of optimization algorithms. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=ne6zeqLFCZ.

[4] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=YicbFdNTTy.

[5] Enealor. Pytorch-sm3, 7 2020. URL https://github.com/Enealor/PyTorch-SM3.

[6] Facebook Research. Adafactor optimizer implementation in fairseq. https://github.com/facebookresearch/fairseq/blob/main/fairseq/optim/adafactor.py, 2023. Accessed: February 2025.

[7] Philip Gage. A new algorithm for data compression. *C Users J.*, 12(2):23–38, February 1994. ISSN 0898-9788.

[8] 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, 2020. URL https://openreview.net/forum?id=BJepq2VtDB.

[9] Aaron Gokaslan, Vanya Cohen, Ellie Pavlick, and Stefanie Tellex. Openwebtext corpus. http://Skylion007.github.io/OpenWebTextCorpus, 2019.

[10] Dirk Groeneveld, Iz Beltagy, Evan Walsh, Akshita Bhagia, Rodney Kinney, Oyvind Tafjord, Ananya Jha, Hamish Ivison, Ian Magnusson, Yizhong Wang, Shane Arora, David Atkinson, Russell Authur, Khyathi Chandu, Arman Cohan, Jennifer Dumas, Yanai Elazar, Yuling Gu, Jack Hessel, Tushar Khot, William Merrill, Jacob Morrison, Niklas Muennighoff, Aakanksha Naik, Crystal Nam, Matthew Peters, Valentina Pyatkin, Abhilasha Ravichander, Dustin Schwenk, Saurabh Shah, William Smith, Emma Strubell, Nishant Subramani, Mitchell Wortsman, Pradeep Dasigi, Nathan Lambert, Kyle Richardson, Luke Zettlemoyer, Jesse Dodge, Kyle Lo, Luca Soldaini, Noah Smith, and Hannaneh Hajishirzi. OLMo: Accelerating the science of language models. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar, editors, *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15789–15809, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.841. URL https://aclanthology.org/2024.acl-long.841/.

[11] Kaiming He, X. Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2015.

[12] Andrej Karpathy. nanoGPT: The simplest, fastest repository for training/finetuning medium-sized gpts. https://github.com/karpathy/nanoGPT, 2022.

[13] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[14] Frederik Kunstner, Robin Yadav, Alan Milligan, Mark Schmidt, and Alberto Bietti. Heavy-tailed class imbalance and why adam outperforms gradient descent on language models, 2024. URL https://arxiv.org/abs/2402.19449.

[15] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=Bkg6RiCqY7.

[16] Yang Luo, Xiaozhe Ren, Zangwei Zheng, Zhuo Jiang, Xin Jiang, and Yang You. CAME: Confidence-guided adaptive memory efficient optimization. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki, editors, *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 4442–4453, Toronto, Canada, July 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.acl-long.243. URL https://aclanthology.org/2023.acl-long.243.

[17] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. In *International Conference on Learning Representations*, 2017. URL https://openreview.net/forum?id=Byj72udxe.

[18] Ionut-Vlad Modoranu, Mher Safaryan, Grigory Malinovsky, Eldar Kurtic, Thomas Robert, Peter Richtárik, and Dan Alistarh. Microadam: Accurate adaptive optimization with low space overhead and provable convergence. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://openreview.net/forum?id=Tck41RANGK.

[19] Yan Pan and Yuanzhi Li. Toward understanding why adam converges faster than SGD for transformers. In *OPT 2022: Optimization for Machine Learning (NeurIPS 2022 Workshop)*, 2022. URL https://openreview.net/forum?id=Sf1NlV2r6PO.

[20] Guilherme Penedo, Hynek Kydlíček, Loubna Ben allal, Anton Lozhkov, Margaret Mitchell, Colin Raffel, Leandro Von Werra, and Thomas Wolf. The fineweb datasets: Decanting the web for the finest text data at scale. In *The Thirty-eight Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2024. URL https://openreview.net/forum?id=n6SCkn2QaG.

[21] 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.

[22] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In Katrin Erk and Noah A. Smith, editors, *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1715–1725, Berlin, Germany, August 2016. Association for Computational Linguistics. doi: 10.18653/v1/P16-1162. URL https://aclanthology.org/P16-1162/.

[23] Noam Shazeer and Mitchell Stern. Adafactor: Adaptive learning rates with sublinear memory cost. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 4596–4604. PMLR, 10–15 Jul 2018. URL https://proceedings.mlr.press/v80/shazeer18a.html.

[24] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. Stanford alpaca: An instruction-following llama model. https://github.com/tatsu-lab/stanford_alpaca, 2023.

[25] Llama Team. The llama 3 herd of models, 2024. URL https://arxiv.org/abs/2407.21783.

[26] torchtune maintainers and contributors. torchtune: Pytorch's finetuning library, April 2024. URL https//github.com/pytorch/torchtune.

[27] Jingzhao Zhang, Sai Praneeth Karimireddy, Andreas Veit, Seungyeon Kim, Sashank Reddi, Sanjiv Kumar, and Suvrit Sra. Why are adaptive methods good for attention models? In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 15383–15393. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper_files/paper/2020/file/b05b57f6add810d3b7490866d74c0053-Paper.pdf.

[28] Yushun Zhang, Congliang Chen, Tian Ding, Ziniu Li, Ruoyu Sun, and Zhi-Quan Luo. Why transformers need adam: A hessian perspective, 2024. URL https://arxiv.org/abs/2402.16788.

[29] Yushun Zhang, Congliang Chen, Ziniu Li, Tian Ding, Chenwei Wu, Yinyu Ye, Zhi-Quan Luo, and Ruoyu Sun. Adam-mini: Use fewer learning rates to gain more, 2024. URL https://arxiv.org/abs/2406.16793.

[30] Rosie Zhao, Depen Morwani, David Brandfonbrener, Nikhil Vyas, and Sham Kakade. Deconstructing what makes a good optimizer for language models, 2024. URL https://arxiv.org/abs/2407.07972.

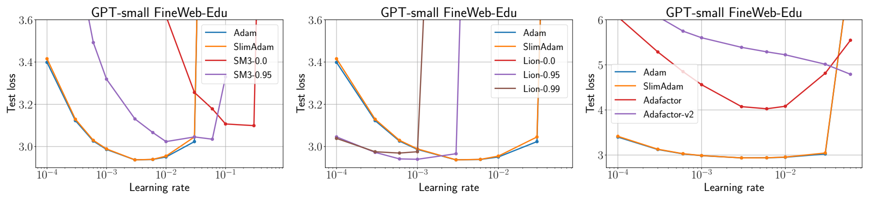


Figure 4: Comparison of *SlimAdam* with different optimizers on GPT pre-training using Fineweb-Edu dataset.

Appendix A. Related Works

The superiority of Adam is primarily observed in language modeling, with SGD performing comparably to Adam in image classification settings [27]. This disparity has motivated several investigations into the unique challenges of language modeling landscapes, with studies identifying several explanations. Ref. [1, 27] demonstrated that the heavy-tailed distribution of the stochastic gradient noise in language modeling cases causes SGD to perform worse than Adam. Ref. [19] attributed Adam's faster convergence to "directional sharpness," which is the curvature along the update direction. Adding to these findings, Ref. [28] illustrated that the Hessian spectrum varies heavily across parameter blocks, attributing SGD's worse performance to using a single learning rate for all blocks. Further insights come from Ref. [14], who found that, in settings with heavy-tailed class imbalance, SGD struggles to decrease loss in infrequent classes, while adaptive optimizers are less sensitive to this imbalance. Ref. [30] argued that Adam's advantage over SGD in language modeling primarily stems from using per-parameter adaptive learning rates in two specific components—LayerNorm and the final layer—positing that for all other layers, a single shared second moment is sufficient.

Several approaches have been proposed to reduce Adam's memory footprint in the past few years. Adafactor [23] approximates the second-moment matrix of a layer using a moving average of the row and column sums of the squared gradients. SM3 [2] groups parameters by tensor dimensions and estimates second moments as the minimum value across relevant group averages. Lion [3] is an algorithmically discovered optimizer that only tracks momentum and uses a sign operation to estimate the update. MicroAdam [18] combines gradient sparsification, quantization, and error feedback to compress optimizer states. Adam-mini [29] assigns adaptive learning rates to block partitions based on the Hessian spectrum at initialization. Specifically, Adam-mini assigns a second moment to the default PyTorch block partition with two exceptions: (1) each parameter in the first and last layer is assigned a second moment, and (2) each key and query head gets a second moment. Below, we further discuss closely related works in detail.

Adam-mini: Ref. [29] introduced Adam-mini, which assigns adaptive learning rates to block partitions based on the Hessian spectrum at initialization. The initial release, Adam-mini v1.0.4 (referred to as Adam-mini v1), uses PyTorch's default block partitioning with two key modifications: (1) individual second moments are assigned to each parameter in the Token Embedding and LM Head, and (2) individual second moments are assigned to each key and query attention head. In a recent update, Adam-mini v1.1.1 (referred to as Adam-mini v2) revises this approach by assigning one second moment per output neuron in each layer, with two exceptions: (1) each key and query attention head receives its own second moment, and (2) each token dimension in the Token Embedding and LM Head receives its own second moment. LayerNorms are always compressed.

Our SNR analysis identifies similar compression rules to Adam-mini but with two key differences. First, Adam-mini assigns one second moment to every output neuron of attention values, projection, and MLPs. In our convention, it amounts to fan_{in} compression. In comparison, our SNR analysis suggests that fan_{out} compression is more appropriate for these layers. The second difference relates to LayerNorm parameters. While Adam-mini compresses these by default, our SNR analysis indicates that LayerNorm second moments show aversion to compression. We attribute *SlimAdam*'s superior learning rate stability to its identification of these more appropriate compression dimensions.

AdaLayer: Ref. [30] found that Adam's superior performance over SGD in language modeling primarily comes from using per-parameter adaptive learning rates in just two components: LayerNorm and the LM Head. All other layers can be trained with SGD. Following their naming convention, we use AdaLayer to refer to Adam with one second moment per weight/bias, and 'AdaLayer+LN+TN' to denote AdaLayer with per-parameter second moments for LayerNorm and final layer parameters.

While our SNR analysis supports their findings about Token Embedding/LM Head and LayerNorm second moments, we find that AdaLayer+LN+TN underperforms Adam and *SlimAdam* using 2% of Adam's second moments closely matches Adam's performance and stability.

SM3: SM3 [2] groups parameters into sets based on similarity, such that each parameter can belong to multiple sets. Then, it maintains a moving average of the maximum of squared moments for each set and approximates a second-moment entry using the minimum value across different sets it belongs to. We use the implementation from [5] with momentum $= 0.9$ and $\beta \in \{0.0, 0.95\}$. Figure 4(a) compares SM3 performance with different β values on the GPT pre-training task. We observe that $\beta = 0.95$ performs better for GPT pre-training. We use this optimal β value in the comparisons shown in Figure 1.

Lion: Lion [3] is an algorithmically discovered optimizer that only tracks momentum and uses the sign operation to determine update directions. For the GPT-small experiment, we found that $\beta_2 = 0.95$ performs best when keeping $\beta_1 = 0.9$ fixed, as shown in Figure 4(b). Similar to other optimizers, we use a weight decay strength of $\lambda = 0.1$ and a gradient clipping threshold of 1.0.

Adafactor: [23] approximates the second-moment matrix of a layer using a moving average of the row and column sums of the squared gradients. We evaluate two implementations: (1) the PyTorch implementation, which does not use a moving average of updates (referred to as Adafactor) and (2) the implementation by Ref. [6], which incorporates the moving average of updates (referred to as Adafactor v2). For both variants, we maintain the same learning rate schedule used in our default experiments. For Adafactor v2, this requires setting `relative_step=False`. As shown in Figure 4(c), both Adafactor variants perform significantly worse than Adam. Due to this performance gap, we exclude these results from Figure 1.

Appendix B. Experimental Details

SNR measurement: We measured SNR values at regular intervals throughout training: every 100 step for the first 1000 steps, then every 1000 step thereafter. For determining the *SlimAdam* rules, we deliberately exclude frequent early-training measurements to prevent biasing the averaged SNR towards initial SNR values.

B.1. Language Pre-training

Model and Datasets: We train GPT-style models [21] using a codebase based on NanoGPT [12] on two language modeling datasets: OpenWebText [9] and 10B token subset of FineWeb-Edu [20]. The datasets are tokenized using the GPT tokenizer with a vocabulary size $n_{\text{vocab}} = 50,304$. The models are trained with a context length of $T_n = 1024$. We use n_{layers} to denote the number of layers, n_{heads} to denote the number of heads, and d_{model} to denote the embedding dimension.

We consider two model configurations:

1. GPT-small ($n_{\text{layers}} = 12$, $n_{\text{heads}} = 12$, $d_{\text{model}} = 768$)
2. GPT-medium ($n_{\text{layers}} = 24$, $n_{\text{heads}} = 16$, $d_{\text{model}} = 1024$)

Both models have an MLP upscaling factor of 4, learnable positional embedding, and weight tying, without biases.

Initialization: Unless specified, we consider the Mitchell initialization [10]: For standard layers, the weights are initialized using a normal distribution $\mathcal{N}(0, 0.02^2)$, while residual projection layers (attention and MLP projections) use a scaled normal distribution $\mathcal{N}(0, {}^{0.02^2}/_{2n_{\text{layers}}})$. In Section 4, we use PyTorch's default uniform initialization: $\mathcal{U}(-\frac{1}{\sqrt{\text{fanin}}}, \frac{1}{\sqrt{\text{fanin}}})$.

Training: The training uses a micro-batch size of 32 with 40 gradient accumulation steps, resulting in an effective batch size of $B = 1,280$. All models are trained for $10,000$ steps using different Adam variants with the following hyperparameters: $\beta_1 = 0.9$, $\beta_2 = 0.95$, $\epsilon = 10^{-8}$, and weight decay strength $\lambda = 0.1$. The learning rate is linearly increased from zero to a target learning rate η in $T_{\text{wrm}} = 2048$ steps, followed by cosine decay to $\eta_{\text{min}} = {}^{\eta}/_{10.0}$. Gradients are clipped at a maximum norm of 1.0.

B.2. Linear Model trained on WikiText

Model Architecture: We consider a two-layer linear model composed of an embedding layer followed by a language model head, trained on WikiText-103 [17]. The dataset is tokenized using BPE tokenization [7, 22] with different vocabulary sizes $V \in \{1024, 2048, 4096, 8192, 16384, 32768, 49152, 65536\}$. The embedding dimension is set to $d_{\text{model}} = 768$ and a context length of $T_n = 128$ is considered.

Initialization: The embedding parameters are initialized using a truncated normal distribution $\mathcal{N}(0, 1)$, while the language model head uses a truncated normal distribution $\mathcal{N}(0, {}^{1}/_{\text{fan}_{\text{in}}})$.

Training: The training consists of one epoch with a batch size $B = 16$. The model is trained using Adam variants with hyperparameters $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$, and weight decay strength $\lambda = 10^{-4}$. The learning rate follows a schedule with linear warmup from zero to η over $T_{\text{wrm}} = 2048$ steps, followed by cosine decay to $\eta_{\text{min}} = {}^{\eta}/_{10.0}$. The optimal target learning rate is found by scanning the set $\{$1e-4, 3e-4, 6e-4, 1e-3, 3e-3$\}$.

B.3. Language Fine-tuning

Model and Datasets: We consider pre-trained Llama-3.2 models [25] and fine-tune them on the Alpaca dataset [24] using the torchtune library [26].

Fine-tuning: We finetune the models for 3 epochs using a batch size $B = 16$, optimizer hyperparameters $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$ and weight decay strength $\lambda = 0.1$.

B.4. Image Classification

Model and Datasets: We train ResNet [11] and ViT [4] models on CIFAR-10 and CIFAR-100 datasets [13] with random crop and horizontal flip augmentations.

ResNet: We consider the standard ResNet-18 architecture with batch normalization.

ViT: We consider Vision Transformers [4], with GPT-like architecture adapted for image classification using patch embeddings and a special class token. We consider two model configurations: ViT-mini ($n_{\text{layers}} = 6$ layers, $n_{\text{heads}} = 12$ heads, embedding dimension $d_{\text{model}} = 768$) and ViT-small ($n_{\text{layers}} = 12$ layers, $n_{\text{heads}} = 12$ heads, embedding dimension $d_{\text{model}} = 768$). Both models are initialized using Mitchell initialization, do not use biases, and use a learnable class token and a patch size of 2.

Training: We train these models with a batch size of $B = 128$ for $100,000$ steps with optimization hyperparamters: $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$ and weight decay strength $\lambda = 0.01$. The learning rate is linearly increased from zero to a target learning rate η in $T_{\text{wrm}} = 2048$ steps, followed by cosine decay to $\eta_{\text{min}} = \eta/10.0$.

B.5. Estimated Computational Resources

Each experiment required approximately 12 H100 GPU hours to complete. Our experimental design included around 8 learning rate variations, 2 distinct datasets for the four training regimes, resulting in 64 total runs. This amounted to 768 H100 GPU hours for the primary experiments. Additional small-scale exploratory experiments consumed approximately 250 H100 GPU hours, bringing the total computational resources used in this study to around 1000 H100 GPU hours.

Appendix C. SNR Analysis for Gaussian Gradients

In this section, we analyze the SNR metric for random, iid Gaussian distributed gradients. Consider a gradient matrix $G \in \mathbb{R}^{n\times n}$ with elements G_{ij} sampled from $\mathcal{N}(0, \sigma^2)$. Let $V = G^2$ denote the element-wise squared gradient matrix. Then, the expectation of the mean and variance along column j is:

$$\mathbb{E}\left[V_i\right] = \mathbb{E}\left[\frac{1}{n}\sum_{j=1}^{n} V_{ij}\right] = \frac{1}{n}\sum_{j=1}^{n}\mathbb{E}[G_{ij}^2] = \sigma^2.$$

$$\mathbb{E}\left[\frac{1}{n}\sum_{j=1}^{n}(V_{ij} - \mathbb{E}[V_i])^2\right] = \frac{1}{n}\sum_{j=1}^{n}\left[\mathbb{E}[G_{ij}^4] - \mathbb{E}[G_{ij}]^2\right] = 3\sigma^4 - \sigma^4 = 2\sigma^4.$$

This yields SNR $= 1/2$ for iid Gaussian gradients irrespective of matrix dimension. We numerically verify this result in Figure 5. In real-world scenarios, gradients follow complex distributions, often exhibiting long tails that defy iid Gaussian assumptions. In our experiments, we found that a more stringent cutoff of 1 works better.

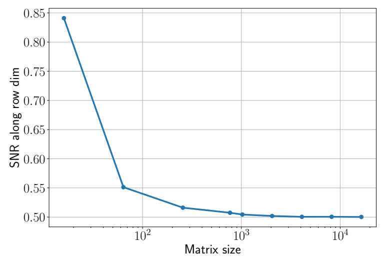


Figure 5: SNR values along the row dimension for iid Gaussian distributed gradients.

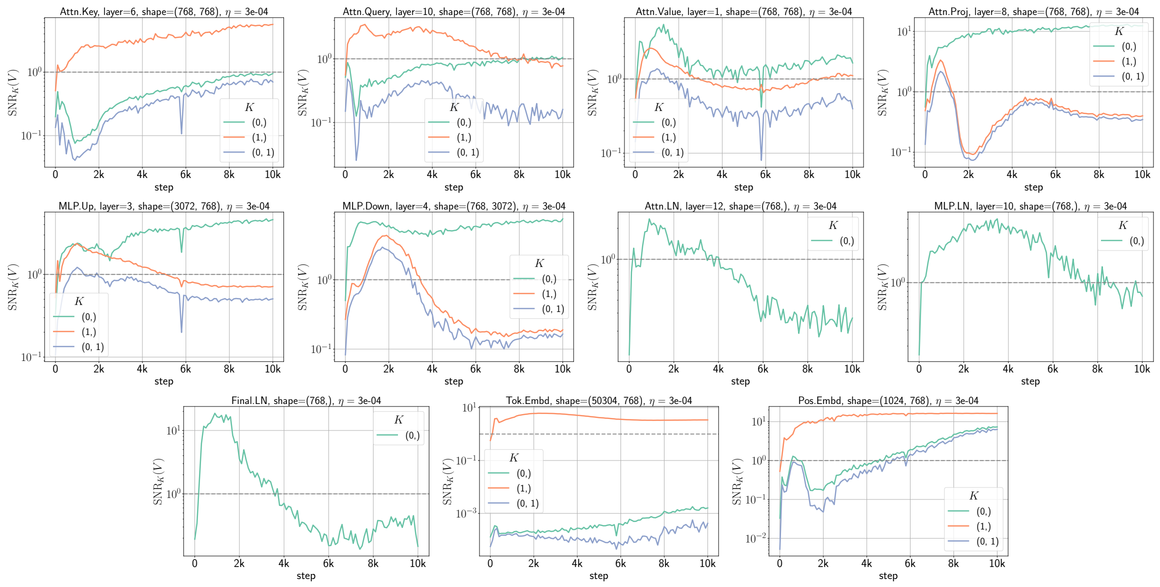


Figure 6: SNR trajectories of GPT-small trained on OpenWebText. For each layer type, the layer number is selected at random.

Appendix D. SNR Analysis Across Diverse Training Regimes

D.1. Language Pre-training

We analyze GPT-style Transformers [21] trained on two language datasets: OpenWebText [9] and a 10B token subset of FineWeb-Edu [20]. Figure 6 shows SNR trajectories as a function of the optimization step for selected second-moment blocks of a GPT-small model trained on OpenWebText. Figure 2 presents the depth dependence of the averaged SNR values of different layer types within a standard transformer block. The lack of consistency as to which compression dimension K exhibits higher SNR across different layer types suggests that optimal compression strategies must be customized for each parameter category rather than applying a uniform approach throughout the model. Below, we describe these trends in detail and discuss their implications.

Token Embedding and Language Modeling Head (LM Head[2]) second moments show a strong aversion to compressing along the token dimension (vocabulary dimension) while favoring compression along the embedding dimension. This pattern suggests that the subset of the parameter matrix corresponding to each token in the vocabulary evolves at its own pace during training, thereby requiring its own learning rate. These findings align with recent studies [29, 30] that advise against compressing the token embedding and LM Head matrices in language modeling. Our SNR analysis extends this understanding by revealing that this aversion to compression is specific only to the token dimension.

Figures 6 and 8 show that similar SNR trajectories are observed across different web text datasets. The layerwise trends shown in Figures 7 and 9 further support this claim. Furthermore, Figure 10 shows that similar SNR trends for a GPT-medium model.

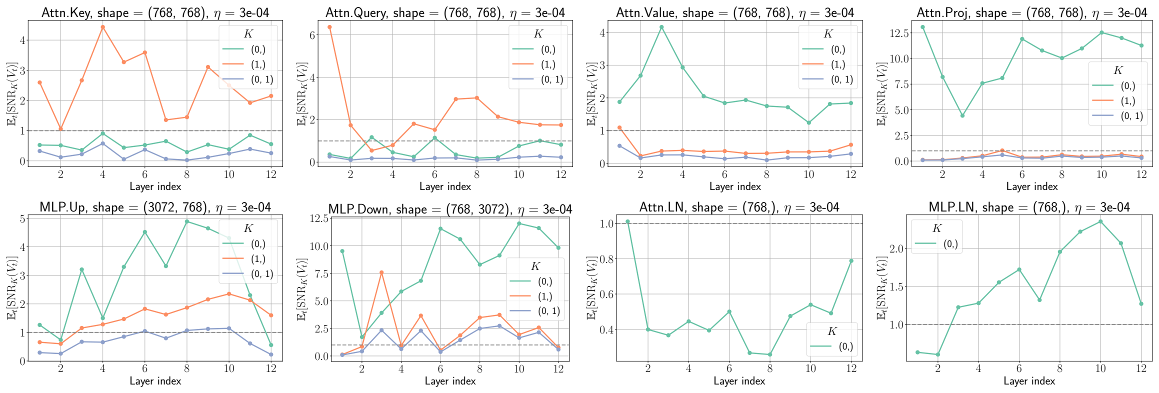


Figure 7: Layer dependence of averaged SNR values of GPT-small trained on OpenWebText.

D.2. Language Fine-tuning

Next, we examine second-moment compressibility during fine-tuning with Llama-3.2 [25] on the Alpaca dataset [24]. Figure 11 shows the SNR trends of selected layers, which reveal layer-wise patterns with subtle distinctions from GPT pre-training.

We find lower SNR values across all layers during fine-tuning, suggesting an aversion to compressibility in general in this experimental setting. This is particularly pronounced in the attention layer, where key and query second moments exhibit SNR values well below 1.0. While attention value and projection second moments maintain an SNR value above 1.0 along fan_{out} dimension, these values are notably smaller than those observed during GPT pre-training. MLP layers display variable compressibility patterns. The first two MLP layers (MLP.Up and MLP.Gate) show sporadic compressibility (SNR $\gtrsim 1$) at certain depths, but without consistently favoring either input or output dimension compression. In comparison, the output MLP layer (MLP.Down) consistently maintains a high SNR value (SNR $\gtrsim 1$) across depths, favoring compression along the fan_{out} dimension. Attention and MLP RMSNorms show consistently low SNR values across layers, while the final RMSNorm's SNR gradually increases during training, eventually exceeding 1.0. The token embeddings show reduced SNR values even along the embedding dimension, possibly due to a larger vocabulary relative to the embedding dimension for the Llama model than the GPT-small model.

2. Unless otherwise mentioned, we use weight tying meaning that the Token Embedding and LM Head share the same underlying set of parameters and moments.

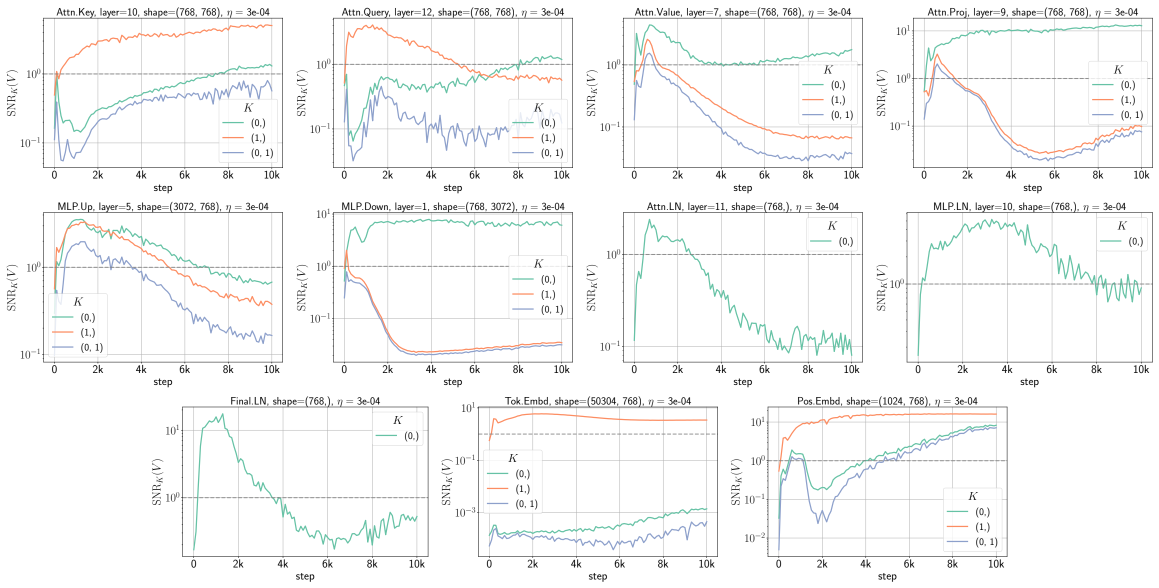


Figure 8: SNR trajectories of GPT-small trained on 10B subset of FineWeb-Edu. For each layer type, the layer number is selected at random.

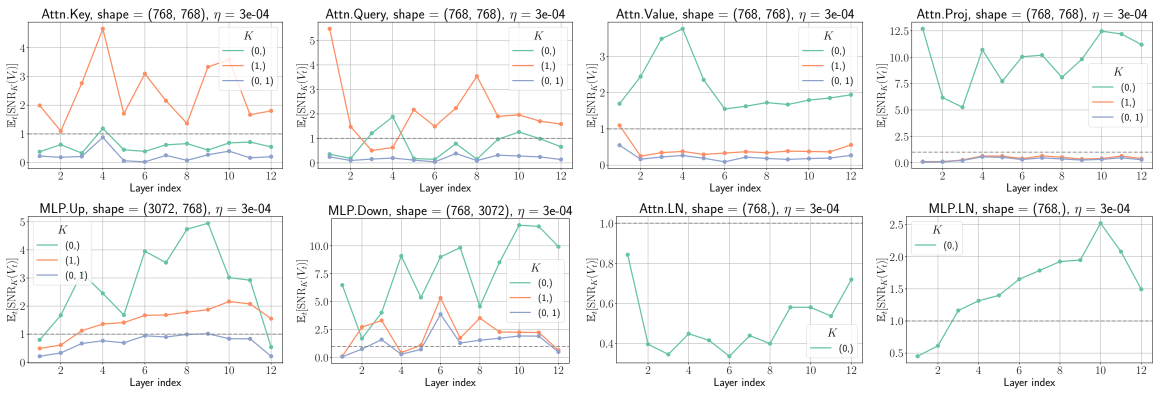


Figure 9: Layer dependence of averaged SNR values of GPT-small trained on 10B token subset of FineWeb-Edu.

D.3. ResNet Image Classification

Compared to language pre-training and fine-tuning settings, the second moments of ResNets trained on CIFAR-100 and CIFAR-10 (Figures 12 and 13) exhibit high SNR values. These SNR values suggest high second-moment compression feasibility across layers. In particular, the intermediate convolutional layers show exceptionally high SNR values across both fan$_{\text{in}}$ and fan$_{\text{out}}$ dimensions, with an increasing trend as a function of depth. By comparison, the first and last layers behave differently. The first convolutional layer resists compression along the fan$_{\text{out}}$ dimension (shown in Figure 13), while the final layer exhibits SNR values close to 1.0 that decreases late in training.

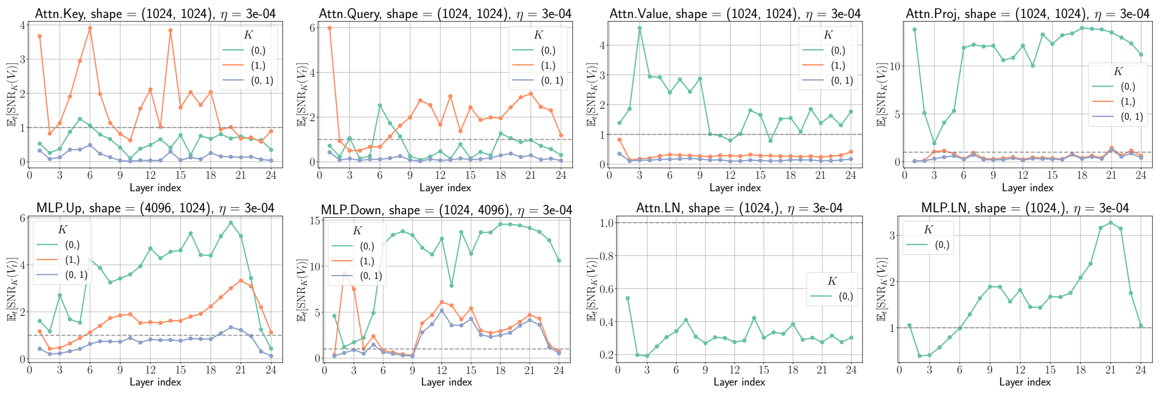


Figure 10: Layer dependence of average SNR values of the GPT-medium trained on FineWeb-Edu.

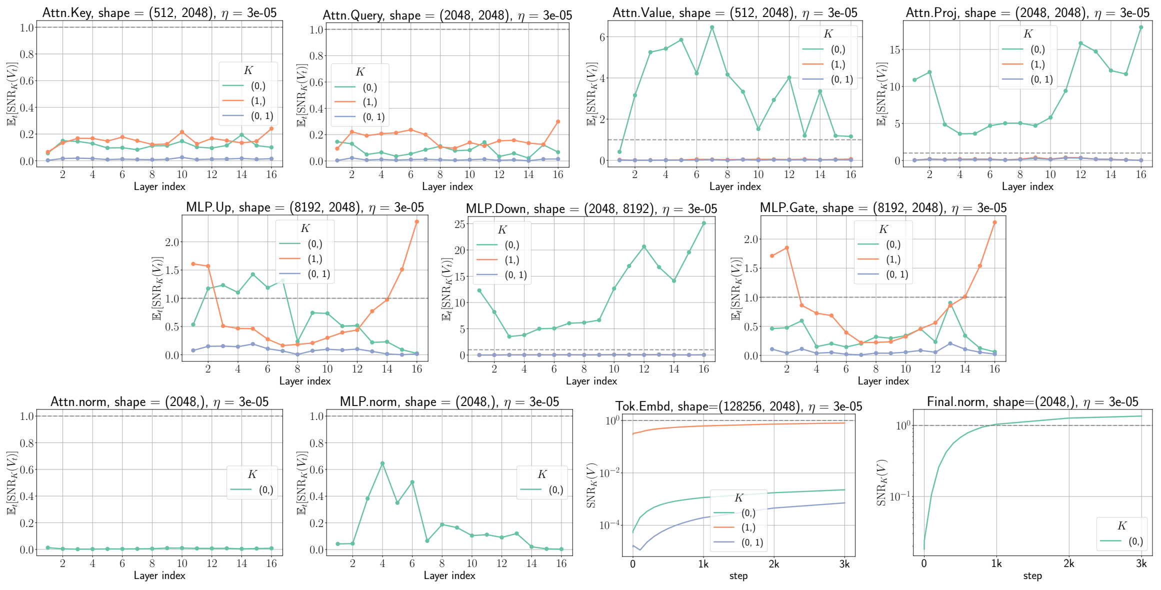


Figure 11: SNR analysis of pre-trained Llama 3.2 1B fine-tuned on Alpaca dataset.

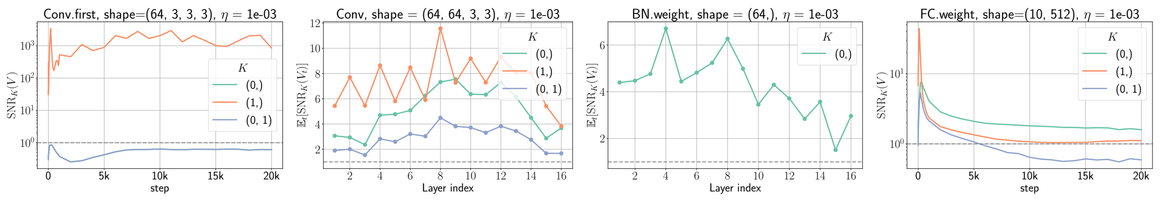


Figure 12: SNR trends of different layers of ResNet-18 trained on CIFAR-10.

D.4. ViT Image Classification

Next, we analyze Vision Transformers (ViTs)[4], with GPT-style Transformer adapted for image classification. Figure 15 shows that ViTs trained on CIFAR-100 exhibit SNR trends combining characteristics from both ResNet and GPT pre-training. The attention moments maintain GPT-like

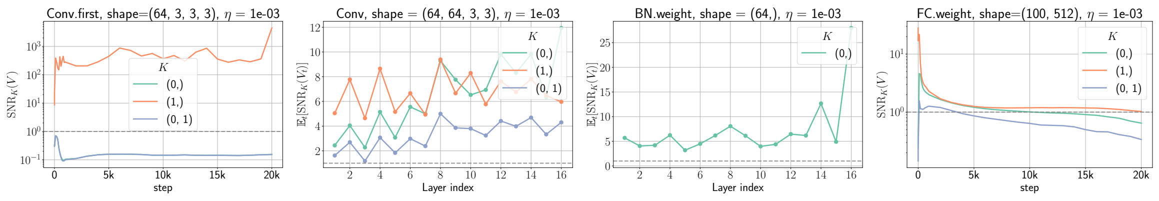


Figure 13: SNR trends of different layers of ResNet-18 trained on CIFAR-100.

SNR trends but with higher SNR values. The keys and query second moments favor fan_{in} compression, while values and projections prefer fan_{out} dimension. These attention components exhibit higher SNR values than GPT pre-training, with the averaged SNR increasing with depth for most layers. Unlike GPT pre-training, the first MLP layer (MLP.Up) favors fan_{in} compression instead of fan_{out}. This suggests that this layer type's compression behavior is training regime-dependent. By comparison, the second layer (MLP.Down) maintains GPT-like fan_{out} preference and exhibits high SNR values along both dimensions. Similar to ResNet's first convolution layer, ViT's patch embedding layer favors fan_{in} compression. Meanwhile, the classification layer maintains SNR values close to 1.0 without consistent preference toward a particular compression dimension. Notably, all LayerNorm components display surprisingly high SNR values, suggesting high compressibility.

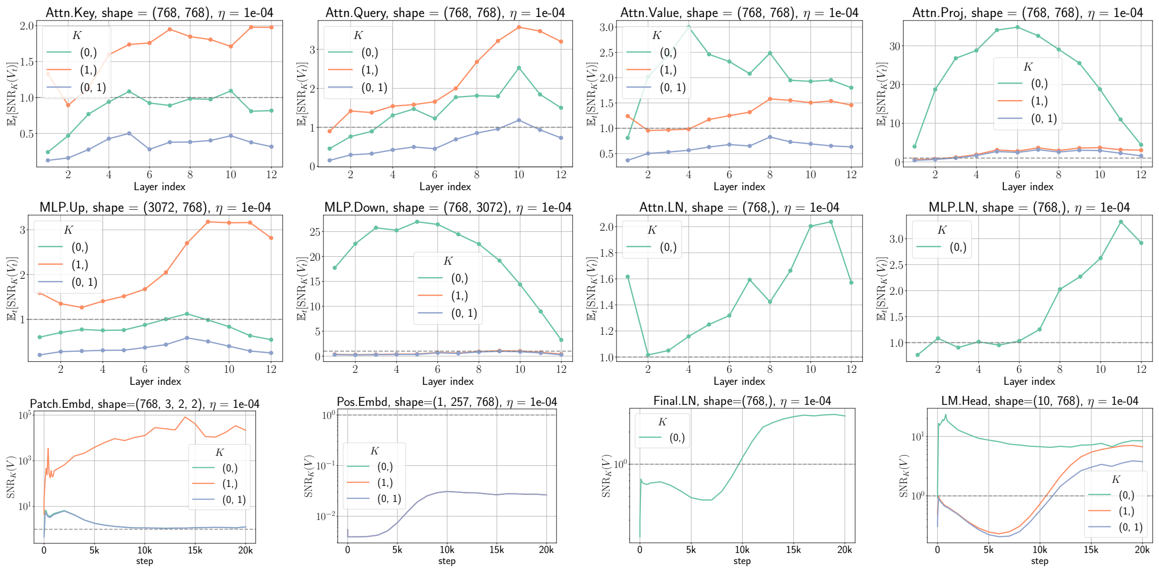


Figure 14: SNR trends of different layers of ViT-small trained on CIFAR-10.

Next, we examine the SNR trends of ResNets and ViTs trained on image classification tasks. As shown in Figures 12 and 13, ResNets trained on both CIFAR-10 and CIFAR-100 exhibit consistently high SNR values, suggesting compressibility. Most layers maintain high SNR values throughout training, with notable exceptions at the network boundaries. The first convolutional layer averses compressibility along the fan_{out} dimension, while the final layer exhibits declining SNR values during later training stages when both dimensions are compressed. Unlike LayerNorm in Transformers, BatchNorm layers demonstrate SNR values around 1.0 throughout training.

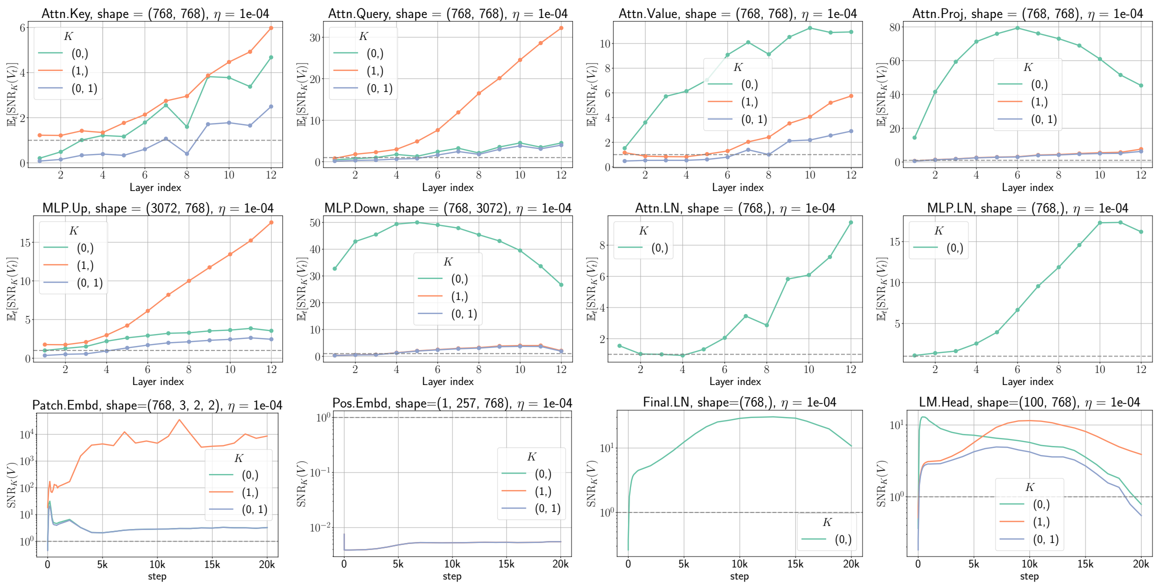


Figure 15: SNR trends of different layers of ViT-small trained on CIFAR-100.

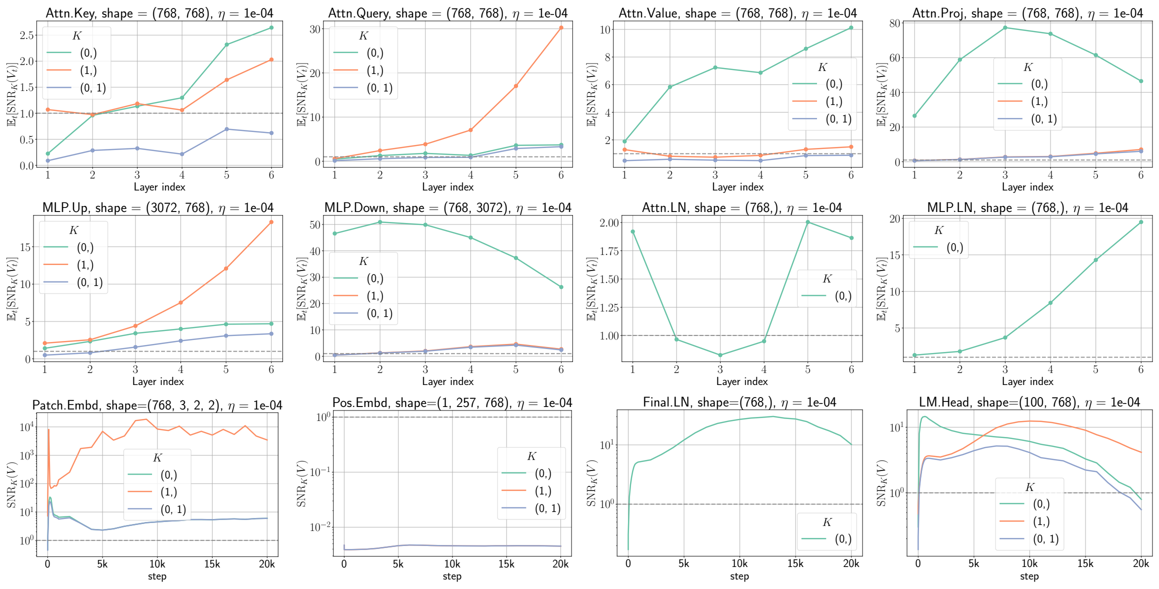


Figure 16: SNR trends of different layers of ViT-mini trained on CIFAR-100.

Appendix E. Effect of Large Learning Rates on Compressibility

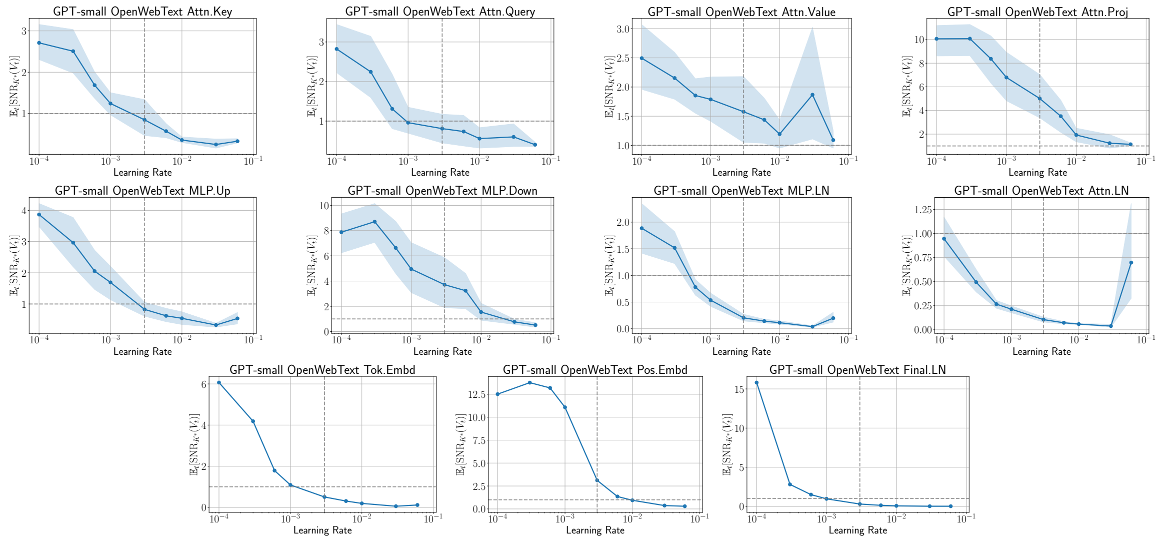


Figure 17: The effect of learning rate on the averaged SNR values of different layers of a GPT-small model trained on the OpenWebText dataset. For each layer, we have selected the dimension K^* with the highest SNR. The shaded region around the mean trend shows the variation across depth. The vertical dashed line at 3e-03 denotes the optimal learning rate.

This section provides supporting results for Section 4 on the effect of learning rates on averaged SNR values $\mathbb{E}_t[\mathrm{SNR}_K(V_t)]$. For each layer, we analyze the effect of the learning rate on the dimension K^* with the highest SNR. Figure 17 shows that the averaged SNR values consistently decrease with the learning rate. This decline suggests that higher learning rates cause training to explore regions of parameter space where gradients contain more outliers, thereby reducing compression feasibility across all layers. Based on the effect of increasing the learning rate on SNR values, we classify layer types into two categories:

1. *Layers that exhibit low SNR values ($\lesssim 1$) at the optimal learning rate:* Token Embedding/LM Head, LayerNorm, attention keys, queries and MLp.Up.
2. *Layers that exhibit high SNR values ($\gtrsim 1$) even at the optimal learning rate:* Attention values, projections and MLP.Down.

Appendix F. Effect of Initialization on Compressibility

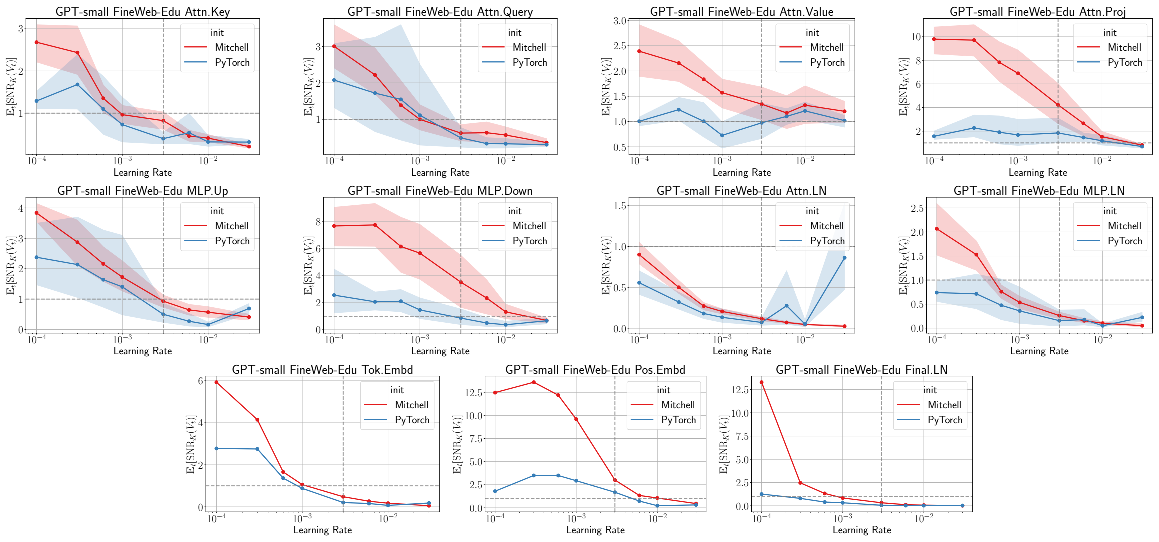


Figure 18: The effect of initialization on the averaged SNR values of different layers of a GPT-small model trained on the OpenWebText dataset. For each layer, we have selected the dimension K^* with the highest SNR. The shaded region around the mean trend shows the variation across depth. The vertical dashed line at 3e-03 denotes the optimal learning rate for Mitchel initialization.

This section provides supporting results for Section 4 on the effect of initialization on averaged SNR values $\mathbb{E}_t[\mathrm{SNR}_K(V_t)]$. We analyze how different initialization schemes affect SNR trends by comparing PyTorch's default initialization with the commonly used Mitchell initialization used in GPT models (recall that Mitchell initialization scales down the variance by $1/\text{depth}$ in layers that add to the residual stream, such as Attn.Proj and MLP.Down). For simplicity, we select the dimension K^* with the highest SNR for each layer.

Figure 18 shows that PyTorch's default initialization exhibits substantially lower SNR values across layers, especially the layers that add to the residual stream (Attn.Proj and MLP.Down) exhibit substantially lower SNR values. These results suggest that the compression feasibility depends on initialization choices and architectural details, suggesting that a single compression strategy is unlikely to work universally.

Appendix G. Incompressibility under Heavy-Tailed Distributions

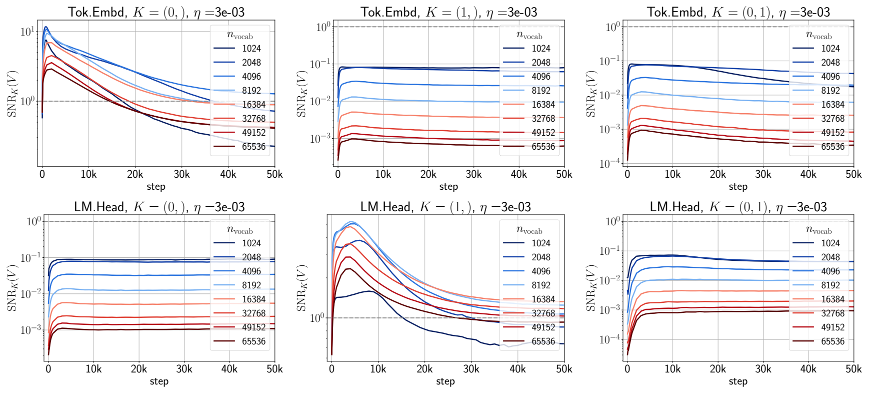


Figure 19: SNR trajectories of the token embedding and linear head of the simplified two-layer model with varying vocabulary sizes.

In Section 3, we observed that language models strongly averse compression along the token dimension in the first and last layers. This resistance suggests that individual tokens require their own learning rates, as their gradients evolve at different paces. To better understand this phenomenon, we investigate how token frequency distribution influences compressibility.

We examine a simplified two-layer model, solely consisting of a token embedding matrix and a linear head. We train the model on the WikiText-103 dataset [17], tokenized using BPE tokenizer [7] with varying vocabulary sizes. By progressively reducing the vocabulary size, we systematically remove rare tokens to control the tail of the token distribution. Figure 19 shows that SNR values along the token dimension and linear head of both layers decrease substantially as vocabulary size increases, suggesting lower compressibility.

We then analyze how large vocabulary sizes affect performance by training the model using Adam with shared second moments (Equation (1)) along dimensions $(K_{\text{embd}}, K_{\text{head}})$. Figure 20 shows the loss gap between the above optimizer and standard Adam, defined as $\Delta L_{\text{Adam}} = L_{(K_{\text{embd}}, K_{\text{head}})} - L_{\text{Adam}}$. For large vocabularies, compression is only effective along embedding dimensions, while token-dimension compression degrades performance. In contrast, small vocabularies permit compression along both dimensions. These findings extend the work of Ref. [14], which showed that Adam outperforms SGD on language tasks by making faster progress on rare tokens. Our analysis suggests that the apparent advantage of Adam in language modeling might stem in large part from allowing individual second moments to each token in the vocabulary.

For both layers, the SNR values along the token dimension ($K=0$ for Tok.Embd and $K=1$ for LM.Head) decrease as the vocabulary size is increased. This suggests that at large vocabulary sizes, each token evolves at its own pace and this requires its own effective learning rate.

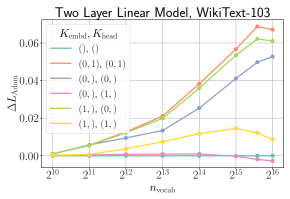


Figure 20: Test loss gap $\Delta L_{\text{Adam}} = L_{(K_{\text{embd}},K_{\text{head}})} - L_{\text{Adam}}$ of the linear model trained with Adam with shared second moments across dimensions $(K_{\text{embd}}, K_{\text{head}})$.

Appendix H. The *SlimAdam* Optimizer

H.1. *SlimAdam* Algorithm

This section describes the *SlimAdam* algorithm in detail. *SlimAdam* implementation consists of three steps. The code is available at `https://github.com/ml-conf-authors/low-memory-adam`.

Step 1: Collect SNR statistics using a small proxy model

First, we collect layer-wise SNR statistics using a small proxy model with a $10\times$ smaller learning rate than optimal. In theory, we would perform the SNR analysis at the optimal learning rate to determine compression rules, but this approach only saves around 30% of seconds moments with a cutoff of 1.0 for Transformer models. Instead, we chose a $10\times$ smaller learning rate, which predicts saving around 98% of second moments for a large range of cutoffs.

```
  Algorithm 1: Collect SNR statistics using a small proxy model
Input: Small model, dataset, optimization hyperparameters (10× smaller learning rate)
Train for $T_{\text{SNR}}$ steps  foreach layer l in model do
    foreach compression dimension $K \in \{(0,),(1,),(0,1)\}$ do
        Compute and record $\text{SNR}_K(V_t^{(l)})$ according to Equation (2)
    end
end
```

Step 2: Extract Compression Rules from SNR Statistics

Next, we identify the compression dimension K^* for each layer type with the highest SNR:

$$K^* = \arg\max_K \mathbb{E}_\tau[\text{SNR}_K(V^{(l)})]. \tag{2}$$

If SNR $\frac{1}{\text{depth}} \sum_{l=1}^{\text{depth}} \mathbb{E}_\tau[\text{SNR}_{K_{\max}}(V^{(l)})]$ exceeds the cutoff, we set the compression dimension $K^{(l)}$ to $K_{\max}$. Otherwise, no compression is performed. This results in consistent compression rules that generalize across depth and width and can be reused.

Step 3: SlimAdam Optimizer

Finally, we train the target model using Adam with shared second moments (Equation (1)) along these compression dimensions K^*. Given either (1) SNR derived compression rules or (2) pre-computed rules from Table 1, SlimAdam applies these rules during training using Equation Equation (1). If $K^{(l)} = \varnothing$, SlimAdam does not compress second moments, and the optimization is identical to Adam.

```
Algorithm 2: Compression Rule Extraction from SNR Statistics
Data: layer-wise SNR statistics and SNR cutoff
foreach layer $l$ in model do
    $K^{(l)} \leftarrow \varnothing$ if $\dim(V^{(l)}) > 1$ then
        $K_{\max} = \arg\max_K \mathbb{E}_\tau[\mathrm{SNR}_K(V_\tau^{(l)})]$ if $\frac{1}{depth}\sum_{l=1}^{depth} \mathbb{E}_\tau[SNR_{K_{\max}}(V^{(l)})] > cutoff$ then
            $K^{(l)} \leftarrow K_{\max}$
        end
    end
end
return $K^*$ for all layers
```

For new training configurations, we suggest deriving compression rules using the SNR statistics of a smaller model. For known training setups, such as GPT pre-training, Table 1 rules can be used out of the box.

```
Algorithm 3: SlimAdam
Data: Learning rate $\eta$, moment decay rates $\beta_1, \beta_2$, layer-wise compression rules $K^{(l)}$
for each training step $t$ do
    $G_t := \nabla_W L(\boldsymbol{\theta}_t)$ for each layer $l$ do
        $M_{t+1}^{(l)} = \beta_1 M_t^{(l)} + (1-\beta_1) G_t^{(l)}$ if $K^{(l)} \neq \varnothing$ then
            $V_{t+1}^{(l)} = \beta_2 V_t^{(l)} + (1-\beta_2)\mathbb{E}_{K^{(l)}}[(G_t^{(l)})^2]$
        else
            $V_{t+1}^{(l)} = \beta_2 V_t^{(l)} + (1-\beta_2)(G_t^{(l)})^2$
        end
        $\hat{M}_{t+1}^{(l)} \leftarrow \frac{M_{t+1}^{(l)}}{1-\beta_1^t}$ $\hat{V}_{t+1}^{(l)} \leftarrow \frac{V_{t+1}^{(l)}}{\sqrt{1-\beta_2^t}}$ $W_{t+1}^{(l)} = W_t^{(l)} - \eta_t \frac{\hat{M}_{t+1}^{(l)}}{\sqrt{\hat{V}_{t+1}^{(l)}+\epsilon}}$
    end
end
```

H.2. Effect of SNR cutoff and Proxy Model Learning Rate on *SlimAdam* Performance

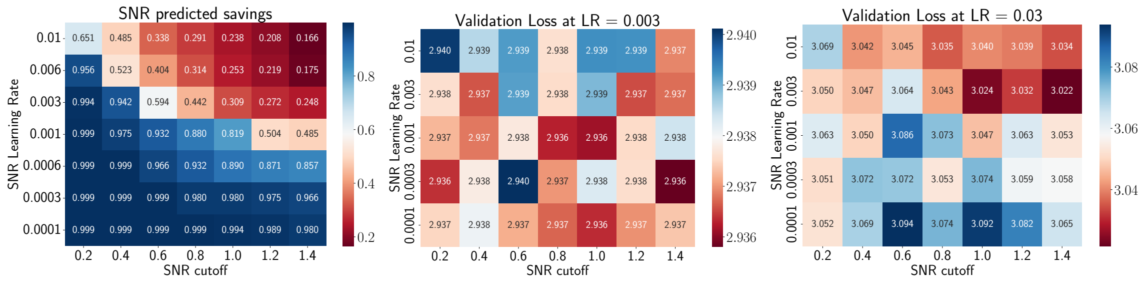


Figure 21: **Effect of SNR cutoff and proxy model learning rate on *SlimAdam* performance:** (left) SNR predicted memory savings, (middle, right) validation loss as a function of SNR learning rate and cutoff for optimal learning rate and a large learning rate.

Figure 21 shows the effect of SNR cutoff and proxy model learning rate (SNR learning rate) on *SlimAdam* performance for GPT-small pre-trained on FineWeb-Edu.

H.3. Additional Results for *SlimAdam*

This section provides additional results for Section 5. Figure 22 compares SNR predicted savings and performance of *SlimAdam* with other baselines on additional tasks. Figures 23 and 24 shows the training loss and downstream performance (HellaSwag and TruthfulQA) of Llama-3.2 1B and Llama 3.2 3B fine-tuned on the Alpaca dataset.

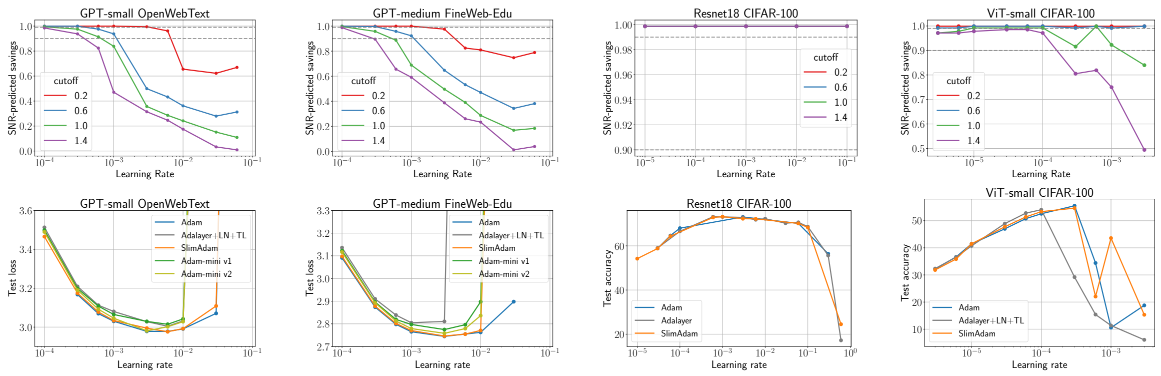


Figure 22: (Top) Fraction of second moments saved (relative to Adam) as a function of learning rate and SNR cutoff across training configuration, as suggested by the SNR analysis. (Bottom) Performance comparison across learning rates between SlimAdam and baselines.

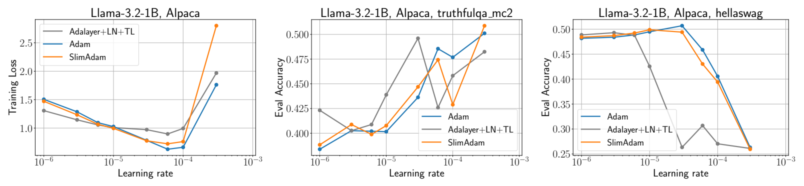


Figure 23: Training loss and Downstream performance of Llama-3.2 1B finetuned on the Alpaca dataset.

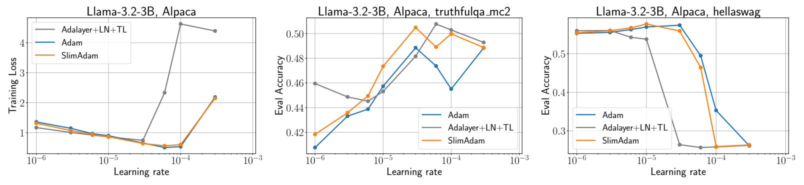


Figure 24: Training loss and Downstream performance of Llama-3.2 3B finetuned on the Alpaca dataset.

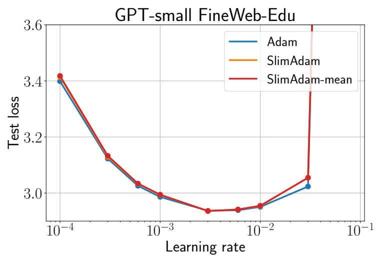


Figure 25: *SlimAdam* with compression rules derived from depth-averaged SNR per layer type (*SlimAdam-mean*) achieves identical performance to SlimAdam with per-layer compression rules.

Appendix I. Robustness of *SlimAdam* Compression Rules

This section analyzes the robustness of *SlimAdam* rules across datasets and model sizes. These variations disappear when using the depth-averaged SNR.

I.1. Dataset Dependency of SlimAdam Rules

This section analyzes how *SlimAdam*'s compression rules vary across different datasets. We compare rules derived from OpenWebText against FineWeb-Edu using GPT-small. The compression rules remain largely consistent, with differences in only five matrices, primarily in early MLP layers, as summarized in Table 2.

Table 2: Compression rule differences between datasets for GPT-small.

Layer	OpenWebText	FineWeb-Edu
Attention		
Attn Query (L3)	None	fan-out
MLP		
MLP Up (L0)	fan-out	None
MLP Up (L1)	None	fan-out
MLP Proj (L1)	fan-out	fan-in
MLP Proj (L2)	fan-in	fan-out

I.2. Width Dependency of SlimAdam Rules

This section analyzes the robustness of *SlimAdam*'s compression rules across model widths (d_{model}). We compare the SNR-derived compression rules for GPT-small with embedding dimension $d_{\text{model}} = 768$ against a narrower model ($d_{\text{model}} = 256$. Out of all layer matrices, we observe differences in compression rules for only 12 matrices, primarily in early to middle layers, as shown in Table 3.

The variations observed in Tables 2 and 3 can be eliminated by deriving compression rules using depth-averaged SNR for each layer type. Figure 25 shows that compression rules derived from depth-averaged SNR result in identical performance to SlimAdam with per-layer compression rules.

Table 3: *SlimAdam* compression rule differences between narrow (width 256) and wide (width 768) models.

Layer	$d_{\text{model}} = 256$	$d_{\text{model}} = 768$
Attention Components		
Attention Value (L0)	fan-in	fan-out
Attention Key (L2)	fan-out	fan-in
Attention Query (L2)	fan-in	fan-out
Attention Query (L3)	fan-in	None
MLP Components		
MLP Up (L0)	fan-in	fan-out
MLP Up (L1)	fan-out	None
MLP Proj (L2)	fan-out	fan-in
MLP Up (L3)	fan-in	fan-out
MLP Up (L4)	fan-in	fan-out
MLP Proj (L4)	fan-in	fan-out
MLP Proj (L5)	fan-in	fan-out
MLP Up (L6)	fan-in	fan-out