Towards Large-scale Training on Apple Silicon

Tycho F. A. van der Ouderaa [1] **Mohamed Baioumy** [1] **Matt Beton** [1] **Seth Howes** [1] **Gelu Vrabie** [1] **Alex Cheema** [1]

Abstract

Training large deep learning models is predominantly done in data centers with NVIDIA GPUs, which are unavailable to most researchers. In this paper, we explore the feasibility of training large language models (LLMs) on clusters of consumer hardware, particularly Apple devices. Compared to NVIDIA GPUs, a cluster of Apple devices has substantially more VRAM, fewer FLOPS, unified memory, and poor bandwidth between nodes. To address these unique hardware constraints, we introduce three key innovations: (1) KPOP, an optimizer that employs Adam in the Kronecker-factored eigenbasis (KFE), enabling efficient training on each node. While this requires more VRAM than AdamW, it outperforms it; (2) a distributed implementation fully leveraging parallel usage of CPU and GPU; and (3) an extension of the optimizer for low-bandwidth environments using top eigenvalues. We provide an extensive evaluation of the proposed methodological advancements, in some cases even outperforming state-of-the-art optimizers such as SGD and Adam even in standard non-Apple training settings. Finally, by combining the proposed techniques, we demonstrate effective training of LLMs on clusters ranging from 2 to 16 Macs.

1. Introduction

There is a clear trend in machine learning towards training increasingly large models on massive compute clusters. This trajectory has been shaped by the hardware and software platforms that dominate the field, often favoring approaches that align well with the prevailing ecosystem.

Figure 1. A cluster of 16 Mac Minis, connected via Thunderbolt 5, training a large language model.

Hooker (2021) characterizes the *hardware lottery* as the phenomenon where 'an idea wins because it is suited to the available software and hardware and not because the idea is superior to alternative research directions', highlighting the often invisible influence of infrastructure on the trajectory of machine learning research.

In turn, this raises the question of how much recent algorithmic and methodological progress reflects genuine insight, and how much is merely an artifact of a development path optimized for specific hardware, particularly massive clusters predominantly composed of NVIDIA GPUs.

Meanwhile, advancements in consumer-grade hardware, particularly in modern Apple Mac systems, have led to significant improvements in computational capabilities (Apple Inc., 2024). These devices now integrate energy-efficient CPUs, high-performance integrated GPUs, and a unified memory architecture, offering a viable alternative to traditional high-performance computing infrastructure for certain machine learning workloads. In this paper, we ask ourselves the question: *How far can we get training a large language model on a cluster of consumer-grade Apple Macs?*

Similarly, (Geiping and Goldstein, 2023) assesses how far one can get when training modern models from scratch on a single GPU. In contrast, we do not restrict ourselves to small models that fit on a single device, but instead aim to train large-scale models across clusters of consumer hardware. This requires hardware differences and practical trade-offs to be taken into account in the design of the training algorithm. In particular, we propose optimizer modifications that trade compute and memory for reduced communication cost, enabling training in low-bandwidth environments.

[1]Exo Technologies Ltd, United Kingdom. Correspondence to: Tycho F. A. van der Ouderaa <tychovdo@gmail.com>.

Accepted for ES-FoMo@ICML2025 workshop.

2. Analysis of hardware constraints

In this section, we provide a high-level overview of how Apple Silicon, the backbone of Apple Mac hardware, differs from NVIDIA hardware. In particular, we analyze differences in the total number of floating point operations per second (FLOPS), the amount of video random access memory (VRAM) available in specialized graphics processing units (GPUs) typically used for training machine learning models, memory bandwidth, and the concept of unified memory. We also discuss inter-node bandwidth in distributed training setups.

Comparison of total FLOPS NVIDIA's data center GPUs, such as the A100 and H100, offer exceptionally high compute throughput, often exceeding 300 TFLOPS for FP16 and Tensor Core operations. These accelerators are purpose-built for large-scale machine learning training. In contrast, Apple Silicon chips like the M3 Ultra provide around 10–30 TFLOPS across the GPU and Neural Engine. Although this is significantly lower per device, Macs are also much cheaper. As a result, a cluster of Apple devices can be cost-effective, provided that distributed training is efficient and the workload scales well across nodes.

Comparison of total VRAM High-end NVIDIA GPUs feature 40–80 GB of dedicated HBM2e or HBM3 memory, optimized for bandwidth and large model capacity. Apple Silicon uses a unified memory architecture, where CPU, GPU, and other components share a common memory pool. The M3 Ultra supports up to 512 GB of unified memory, making it feasible to fit large models directly into memory. However, this memory is shared across subsystems and lacks the dedicated high-bandwidth characteristics of NVIDIA's GPU VRAM.

Comparison of communication bandwidth Distributed training often depends on high-speed communication. NVIDIA GPUs use technologies like NVLink, NVSwitch, and Infiniband, enabling bandwidths of up to 900 GB/s within a node and up to 400 Gbps between nodes. Apple Silicon does not support such interconnects, but Thunderbolt 5 offers up to 120 Gbps of bi-directional bandwidth in boosted mode. While this is far below Infiniband speeds, it may be sufficient for small-scale clusters if communication is carefully optimized.

3. Method

In this section, we describe a novel optimizer named KPOP, which builds upon recent insights in optimization literature. The motivation behind the optimizer is to utilize high memory per node while allowing low communication bandwidth. Interestingly, even in our standard training setting experiments (see Section 4), in which we use NVIDIA GPUs, we find that the optimizer outperforms state-of-the-art optimizers, including Adam (Kingma and Ba, 2015; Loshchilov and Hutter, 2019), both in terms of the number of iterations and wall-clock time. We find this a notable result, highlighting that the optimizer may be generally useful and applicable beyond the scope of this work, which focuses on demonstrating training large models with consumer hardware, in particular clusters of Apple Macs.

Our optimizer is built upon the KFAC optimizer (Martens and Grosse, 2015), which provides a practically effective and scalable approximation of the natural gradient by factorizing the Fisher information matrix. In particular, we build on the Kronecker-Factored Eigenbasis (KFE) formulation introduced by George et al. (2018), which diagonalizes the KFAC approximation to enable more efficient computation and better conditioning. The idea of applying component-wise adaptive optimizers, such as Adam, in the KFE was suggested as a promising future direction in their work but, as far as we know, was never actualized. In this paper, we develop this idea into a practical optimizer, KPOP, which applies Adam in the KFE. Our approach operationalizes this suggestion by combining curvature-aware preconditioning with adaptive learning rates. We demonstrate that KPOP retains the benefits of natural-gradient methods while improving robustness and ease of use.

3.1. KPOP: Adam in the KFE

KFAC approximation We start by considering the Fisher information matrix for the model parameters $\boldsymbol{\theta} = (\mathbf{W}_1, \ldots, \mathbf{W}_L)$, where $\{\mathbf{W}_i\}_{i=1}^{L}$ denote the weight matrices of the linear layers that make up the architecture. In KFAC (Martens and Grosse, 2015), it was shown that by assuming independence between layers and input and outputs, the Fisher matrix can be approximated as a block-diagonal matrix where each block consists of a Kronecker product:

$$\mathbf{F} \approx \text{diag}\left(\mathbf{R}_1 \otimes \mathbf{C}_1, \mathbf{R}_2 \otimes \mathbf{C}_2, ..., \mathbf{R}_L \otimes \mathbf{C}_L\right) \quad (1)$$

between $\mathbf{R}_i \in \mathbb{R}^{R \times R}$ and $\mathbf{C}_i \times \mathbb{R}^{C \times C}$ represent the row and column Kronecker factors for each weight matrix $\mathbf{W}_i \in \mathbb{R}^{R \times C}$, respectively. Although the true Fisher information matrix $\mathbf{F}$ scales quadratically in the number of parameters $\mathcal{O}(P^2)$, the KFAC approximation can be represented using only $\mathcal{O}(LR^2 + LC^2)$ memory, where L is the number of weight matrices and R and C are the respective maximum number of rows and columns used in the architecture, drastically reducing the number of parameters. An example of this approximation being scaled to large language models with over a billion parameters can be found in van der Ouderaa et al. (2023).

EMA Updates of Kronecker Factors To adapt the Fisher information matrix during training, we employ Exponential

Moving Averages (EMA) for the row and column Fisher factors. At each iteration, we compute the batch-level estimates for the row and column Fisher factors, $\mathbf{R}_i$ and $\mathbf{C}_i$, as:

$$\mathbf{C}_i \leftarrow \frac{1}{B}\mathbf{X}_i\mathbf{X}_i^\top \in \mathbb{R}^{C\times C} \quad \text{and} \tag{2}$$

$$\mathbf{R}_i \leftarrow \frac{1}{B}\Delta\mathbf{Y}_i\Delta\mathbf{Y}_i^\top \times \mathbb{R}^{R\times R} \tag{3}$$

where $\mathbf{X}_i \in \mathbb{R}^{C\times B}$ is the activation matrix for the i-th layer, $\Delta\mathbf{Y}_i = \nabla_{\mathbf{Y}_i} L \in \mathbb{R}^{R\times B}$ is the gradient of the final cross-entropy training loss L with respect to the output of linear layer associated to weight $\mathbf{W}_i$, and B is the batch size (both batch and sequence length in LLM setting). We then update the EMA estimates of the row and column Fisher factors using decay parameters α:

$$\widehat{\mathbf{C}}_i \leftarrow \alpha\widehat{\mathbf{C}}_i + (1-\alpha)\mathbf{C}_i \quad \text{and} \tag{4}$$

$$\widehat{\mathbf{R}}_i \leftarrow \alpha\widehat{\mathbf{R}}_i + (1-\alpha)\mathbf{R}_i \tag{5}$$

The matrices $\widehat{\mathbf{C}}_i$ and $\widehat{\mathbf{R}}_i$ provide running estimates of the Kronecker-factored Fisher information matrix approximation throughout the training process, while maintaining a constant memory footprint.

Projection to the KFE Eigenbasis In the next step, we transform the Fisher approximation into the Kronecker-Factored Eigenbasis (KFE) (George et al., 2018; Bae et al., 2018). The KFE representation diagonalizes the Fisher factors $\mathbf{C}_i$ and $\mathbf{R}_i$ by projecting them onto their respective eigenbases. Specifically, we compute the eigenbasis matrices $\widehat{\mathbf{Q}}_i^C$ and $\widehat{\mathbf{Q}}_i^R$ for the column and row factors:

$$\begin{aligned}\widehat{\mathbf{Q}}_i^C &= \text{Eig}(\widehat{\mathbf{C}}_i)\\ \widehat{\mathbf{Q}}_i^R &= \text{Eig}(\widehat{\mathbf{R}}_i)\end{aligned} \tag{6}$$

where $\text{Eig}(\cdot)$ denotes the eigen-decomposition. Although direct computation of this decomposition can be expensive, it can be made efficient through various strategies, such as reusing eigenvector estimates from previous iterations. We discuss these strategies in more depth in Section 3.2.

The eigenvectors in $\widehat{\mathbf{Q}}_i^C$ and $\widehat{\mathbf{Q}}_i^R$ represent rotations of the weight space. We then transform the gradient $\mathbf{G}_i$ from the original parameter space into the KFE:

$$\mathbf{G}_i' = \widehat{\mathbf{Q}}_i^R\mathbf{G}_i(\widehat{\mathbf{Q}}_i^C)^\top \tag{7}$$

This transformation rotates the weight space to the KFE, where the Fisher approximation is closer to diagonal. This aligns more closely with the assumptions of adaptive optimizers like Adam, which implicitly assume diagonal curvature.

Adaptive Optimization in the KFE Finally, we apply the Adam optimizer (Kingma and Ba, 2015) within the KFE. Adam adapts the learning rate for each parameter based on the first and second moment estimates. In the KFE space, we compute the first and second moment estimates for the transformed gradient $\mathbf{G}_i'$. By writing the projected gradients as a vector $\text{vec}(\mathbf{G}')$, we can write the Adam update in its most canonical form: starting with an EMA of the mean and squared (in our case rotated) gradients:

$$\mathbf{m}_i \leftarrow \beta_1\mathbf{m}_i + (1-\beta_1)\text{vec}(\mathbf{G}_i') \quad \text{and} \tag{8}$$

$$\mathbf{v}_i \leftarrow \beta_2\mathbf{v}_i + (1-\beta_2)\text{vec}(\mathbf{G}_i')^2 \tag{9}$$

where β_1 and β_2 are the decay rates for the moment estimates, and $\cdot^2$ denotes the element-wise square. The bias-corrected moment estimates are given by:

$$\widehat{\mathbf{m}}_i \leftarrow \frac{\mathbf{m}_i}{1-\beta_1^t} \quad \text{and} \quad \widehat{\mathbf{v}}_i \leftarrow \frac{\mathbf{v}_i}{1-\beta_2^t} \tag{10}$$

The parameter update is then computed as:

$$\Delta\mathbf{W}_i \leftarrow -\text{mat}\left(\alpha\frac{\widehat{\mathbf{m}}_i}{\sqrt{\widehat{\mathbf{v}}_i}+\epsilon}\right) \tag{11}$$

where $\text{mat}(\cdot)$ reshapes the vector into a matrix and is defined to be the inverse of $\text{vec}(\cdot)$ and a small constant ϵ is added for numerical stability. Finally, we rotate the update back to the original weight space using the inverse of the eigenbasis matrices:

$$\mathbf{W}_i \leftarrow \mathbf{W}_i + (\widehat{\mathbf{Q}}_i^R)^\top\Delta\mathbf{W}_i\widehat{\mathbf{Q}}_i^C \tag{12}$$

This completes the update for the weight matrix $\mathbf{W}_i$.

Final algorithm and related work Pseudocode for the full KPOP algorithm is presented in 1. The algorithm is very closely related to SOAP (Vyas et al., 2024b), a recently proposed optimizer that performs Adam in the eigenbasis of Shampoo (Gupta et al., 2018). The eigenbasis of Shampoo and KFAC require the same amount of memory and are closely related, as discussed in Appendix B of (Anil et al., 2020). Although using the top eigenvalues of the KFE has been explored in the context of post-training pruning (Wang et al., 2019), we are not aware of applying it to compress gradients during training. Similar compressions for the momentum term have been proposed in GaLoRe (Zhao et al., 2024) and AdaMeM (Vyas et al., 2024a). Notably, Vyas et al. also hypothesizes top eigenvalues of the KFE can be useful in optimization, but did not attempt this in any of the experiments.

3.1.1. Top eigenvalue gradients: TopKPOP

To reduce communication overhead, we propose TopKPOP, which extends KPOP by retaining only the top eigenvalues

of the Kronecker-factored eigenbasis (KFE). Concretely, instead of using all eigenvectors, we introduce a sparsity ratio $0 < s < 1$ and compute only the top eigenvectors:

$$\begin{aligned} \hat{\mathbf{Q}}_i^R &= \text{TopEig}(\mathbf{R}, R') \in \mathbb{R}^{R \times R'} \\ \hat{\mathbf{Q}}_i^C &= \text{TopEig}(\mathbf{R}, C') \in \mathbb{R}^{C \times C'} \end{aligned} \tag{13}$$

where $\text{TopEig}(\mathbf{R}, R')$ denotes the matrix containing the top R' eigenvectors of $\mathbf{R}$. We set $R' = \sqrt{s}R$ and $C' = \sqrt{s}C$. In practice, R' and C' may not be integers, in which case we round to the nearest integer, typically using ceiling. The projected gradients then have shape

$$\mathbf{G}_i' = \widehat{\mathbf{Q}}_i^{R\top} \mathbf{G}_i \widehat{\mathbf{Q}}_i^C \in \mathbb{R}^{R' \times C'} \tag{14}$$

and Adam effectively operates on a lower-dimensional linear subspace of dimension

$$\text{vec}(\mathbf{G}_i') \in \mathbb{R}^{sRC} \tag{15}$$

This reduces the memory overhead of the Adam optimizer, specifically the storage required for the $\mathbf{m}$ and $\mathbf{v}$ vectors, by a factor of s. It also reduces communication overhead by the same factor, since the projected gradients $\text{vec}(\mathbf{G}_i')$ are the quantities exchanged between nodes.

3.1.2. Large batch sizes

A final strategy to reduce communication time relative to computation time is to increase the batch size. While this does not reduce communication per se, it increases the computation per step, thereby lowering the relative cost of communication.

Second-order optimizers, like KPOP, are particularly well-suited to large batch training because they adapt to the curvature of the loss landscape, allowing for more informed updates even when gradients become less noisy at larger batch sizes, as argued in (Grosse and Martens, 2016; Martens and Grosse, 2015; Osawa et al., 2019). In Section 4.3, we demonstrate that indeed KPOP scales favourably with large batch sizes compared to first-order methods like Adam.

3.2. Implementation details and efficiency

What gradients are preconditioned Most of the parameter of a large language model are weight matrices of linear layers (e.g. Q, K, V matrices and fully-connected MLP blocks). Since KFAC only applies to these weight matrices, we only compute a preconditioner for the matrix-valued gradients of these weight matrices $\mathbf{W}_i$ to the KFE when doing KPOP. For the embedding layer, last layer and other (e.g. scalar and vector-valued) parameters, we use standard Adam.

Scheduling A strategy to avoid the costly computation of the preconditioner is to refrain from computing the preconditioner matrix at every iteration. In Vyas et al. (2024b), it was found to be practical to perform this computation every 10 or 100 iterations, which we also adopt in our experiments. In addition, we propose using a warm-up period during which the preconditioner is computed at every iteration, regardless of the chosen frequency. After this period, we switch to infrequent updates.

Reusing previous estimates Following (Wang et al., 2019) and (Vyas et al., 2024b), we can approximate eigenvalue composition by a single power iteration $\text{Eig}(\mathbf{M}) \approx \text{QR}(\mathbf{MQ})$, where $\mathbf{M}$ is the current estimate $\widehat{\mathbf{C}}_i$ or $\widehat{\mathbf{R}}_i$ and $\mathbf{Q}$ corresponds to the last previous eigenvector estimate.

MLX Implementation We implemented our algorithm using the MLX framework (Hannun et al., 2023), which provides a functional programming environment for differentiable graph compilation that supports compilation of computational graphs containing operations both on GPU and CPU. Since MLX is a functional framework, we cannot rely on hooks, as one would in PyTorch, to obtain Kronecker factors. Instead, we drew inspiration from a JAX implementation approach that uses dummy variables to obtain gradients with respect to the outputs of linear layers, as discussed in the JAX community (Johnson, 2024).

4. Results in regular training setting

This paper aims to make progress toward training LLMs on consumer hardware, such as Apple Silicon. Motivated by this, we propose several methodological advances, including the novel KPOP optimizer and extensions for low-bandwidth training. To establish a clear baseline, we first evaluate KPOP on a standard NVIDIA setup, ensuring that observed performance improvements are due to the proposed methods rather than hardware-specific factors. Interestingly, we find that our proposed optimizer may in some cases also outperform existing baselines in this setting. Results on Apple Silicon are presented in section Section 5.

4.1. Quantitative performance

It is well known that papers proposing new deep learning optimizers often report strong results within their own carefully constructed experimental setups, but these results frequently fail to generalize or replicate consistently across broader tasks and architectures. Despite the steady stream of proposed alternatives, Adam has remained the de facto standard in both research and practice due to its robustness and ease of tuning. Whether KPOP will achieve similar longevity remains to be seen, but in this work, we make a concerted effort to compare it against well-tuned baselines of competitive optimizers under fair and realistic conditions. To that end, we use NanoGPT as our testing ground (Karpathy, 2022), a lightweight, modular codebase widely used by

practitioners for speed runs and by researchers for optimizer evaluation. NanoGPT has also served as a benchmark in recent optimization papers (Jordan et al., 2024; Liu et al., 2025; Vyas et al., 2024b), making it a credible and widely accepted baseline for quantitative comparison.

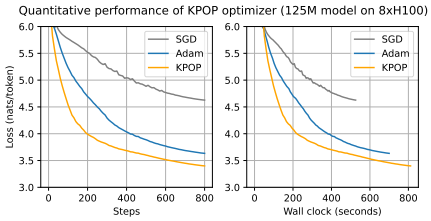

Figure 2. Quantitative evaluation of KPOP optimizer on standard NVIDIA training set-up. Using 125M model on 8xH100 GPU.

4.2. Computational and Memory Cost

The KPOP optimizer introduces overhead in per iteration computation and memory. In particular, computing the preconditioner matrices $\mathbf{Q}_i^R$ and $\mathbf{Q}_i^C$ and synchronizing Kronecker factors $\mathbf{R}_i$ and $\mathbf{C}_i$ requires additional time, and storing running estimates of the Kronecker factors $\widehat{\mathbf{R}}_i$ and $\widehat{\mathbf{C}}_i$ requires additional memory. In Figure 3, we measure

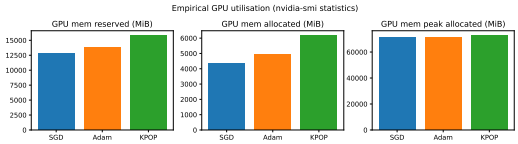

Figure 2. (a) Memory overhead.

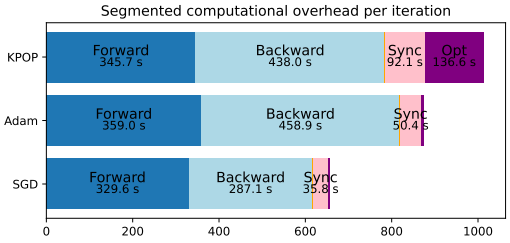

Figure 2. (b) Computational overhead.

Figure 3. Comparing memory and computational overhead of KPOP optimizer to Adam on a standard 8×H100 NVIDIA training setup. KPOP incurs additional cost due to synchronization of Kronecker factors $(\mathbf{R}, \mathbf{C})$ and computing preconditioners $(\mathbf{Q}^R, \mathbf{Q}^C)$.

total memory usage using the same experimental setup as before, reporting statistics from `nvidia-smi` for reserved memory, allocated memory, and peak allocated VRAM. Similarly, we report the total compute time per iteration in Figure 3, segmented across the forward pass, backward pass, communication time, and optimizer time. On average, KPOP iterations take about 10–20% more time than Adam in this measurement. However, as shown in Section 4.1, KPOP is more effective in terms of wall-clock time: individual iterations are slower, but is more efficient with each iteration yielding a larger gain in test performance.

4.3. Large batch size scaling

Larger batch sizes help amortize communication overhead by increasing the amount of computation between synchronization points, thus improving the efficiency of distributed training. In theory, scaling the batch size by a factor of two should reduce the number of optimization steps needed to reach a target performance by half. In practice, however, this linear scaling holds only up to a certain batch size threshold, referred to as the critical batch size (McCandlish et al., 2018). Beyond this point, efficiency gains diminish. As both model and data sizes continue to grow, it becomes increasingly important to design optimization algorithms that are robust to large batch training, enabling higher critical batch sizes and better utilization of compute in parallel environments. As discussed in Section 3.1.2,

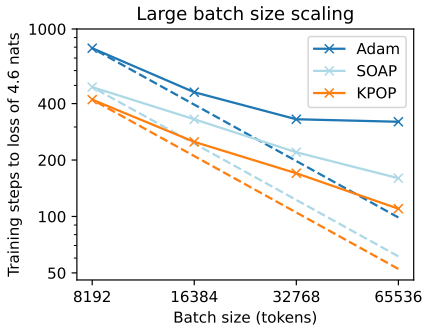

Figure 4. Comparing the number of iterations it takes to reach a validation loss of 4.1 for Adam, SOAP and KPOP.

second-order optimizers, like KPOP, have been found to be more amenable for large batch size training. To verify whether this also holds for KPOP, we compare KPOP to Adam. We follow the set-up described in Section 6.3 of (Vyas et al., 2024b) and evaluate the training steps it takes to reach a certain validation loss, in our case 4.1, each time halving the batch size and doubling the preconditioner frequency and do not use any preconditioner warm-up for this experiment. We also include the SOAP (Vyas et al., 2024b) optimizer in the experiment as a reference, which we find to have similar performance as KPOP both in absolute performance as well as scaling in batch size. We report results in Figure 4, and find that KPOP scales favourably with batch size compared to Adam, in our experiment. This adds evidence to the fact that second-order optimizers can remain effective at larger batch sizes.

Method	Sparsity	Test. Loss	Test. PPL
KPOP	1.0	**3.41**	**30.3**
TopKPOP	0.9	3.41	30.3
	0.7	3.43	30.9
	0.5	3.46	31.8
	0.3	3.51	33.4
	0.2	3.55	34.8
	0.1	3.65	38.5
	0.01	4.04	56.8

Table 1. Performance at different TopKPOP sparsity levels.

4.4. Reducing bandwidth with TopKPOP

In this section, we explore using the use of top eigenvalues as discussed in Section 3.1.1. In particular, we have a percentage $s \in (0, 1]$ where $s = 1$ is standard KPOP and lower values $0 < s < 1$ result in faster preconditioner computation, lower communication overhead, but at the cost of lower performance per iteration. We assess the effect of s by measuring optimization and synchronization time and final test performance after a full training run using the same set-up as in previous chapters.

5. Results on Apple Silicon

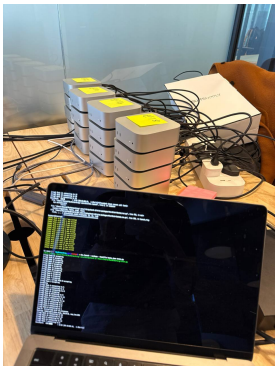

Figure 5. 16×Mac Mini cluster (M4), Thunderbolt 5.

Figure 6. 2×Mac Studio cluster (M3 Ultra), Thunderbolt 5.

We demonstrate that KPOP and its communication-efficient variant, TopKPOP, enable large language model training on consumer-grade Apple hardware. We evaluate two configurations: (1) a cluster of 16 Mac Minis with M4 chips, and (2) a smaller setup with 2 Mac Studios equipped with M3 Ultra chips. All devices are connected via Thunderbolt 5. Due to differences in implementation, results in this section are not directly comparable to the NVIDIA-based experiments of the previous section, which used PyTorch instead of MLX.

The model employs a small-scale LLaMA architecture (Touvron et al., 2023) with 4 transformer layers, a hidden size of 128, and an intermediate feedforward dimension of 256. We train on FineWeb dataset (Penedo et al., 2024) for 11-12 K iterations with 16384 tokens per batch, 16 batches of 1024 tokens. Training is conducted for a single epoch using a batch size of 16 and an initial learning rate of 0.02 and a cosine annealed (Loshchilov and Hutter, 2016) and no weight decay.

Table 2. Training transformer models on Apple Silicon clusters.

Optimizer	Setup	Nodes	Chip	Model size	Iterations	Train time	Test loss	Test PPL
Adam	Mac Studio Cluster	2	M3 Ultra	35.7 M	11 K	34 min	5.33	205.7
KPOP (**ours**)	Mac Studio Cluster	2	M3 Ultra	35.7 M	7.9 K (early stop)	34 min	4.76	117.0
KPOP (**ours**)	Mac Studio Cluster	2	M3 Ultra	35.7 M	11 K	48 min	**4.70**	**110.5**
Adam	Mac Mini Cluster	16	M4	1.7 M	12 K	5 min	6.48	648.8
KPOP (**ours**)	Mac Mini Cluster	16	M4	1.7 M	2.4 K (early stop)	5 min	5.41	222.5
KPOP (**ours**)	Mac Mini Cluster	16	M4	1.7 M	12 K	25 min	**4.87**	**130.8**
TopKPOP @ 10% (**ours**)	Mac Mini Cluster	16	M4	1.7 M	12 K	25 min	5.27	194.2

In Table 2, we compare the performance of Adam and our proposed optimizer, KPOP, on the two Apple Silicon clusters. For each cluster, we report two KPOP configurations: one early-stopped to match Adam's wall-clock time, and one run for the same number of iterations as Adam. This enables a fair comparison in terms of both time-to-quality and final performance. In both cases, KPOP significantly outperforms Adam in final test loss and perplexity. The early-stopped KPOP results demonstrate faster convergence, while the full KPOP runs achieve stronger final performance. We also evaluate TopKPOP, a sparsified variant, on the 16× Mac Mini cluster. While it does not reduce wall-clock time, suggesting the setup is limited more by computation or memory than communication, it nonetheless demonstrates a real reduction in communication overhead while maintaining strong final performance. These results highlight the feasibility of efficient, large-scale training on consumer-grade hardware, showing that clusters of Apple Silicon devices can be leveraged for meaningful model development at scale.

6. Conclusion

In this paper, we introduced KPOP, a novel deep learning optimizer inspired by recent advancements in second-order optimization, that performs Adam updates in the Kronecker-factored eigenbasis (KFE). Experimentally, we find that KPOP outperforms SGD and Adam in standard training settings. We further show a communication-efficient extension, TopKPOP, by retaining only the top eigenvalues during synchronization, drastically reducing gradient all-reduce overhead in distributed setups. We show that these algorithmic improvements in turn enable large-scale language model training on consumer-grade hardware, specifically Apple hardware. We demonstrate this by training models across clusters of 2 to 16 Apple Silicon machines connected via Thunderbolt 5. These results underscore the potential of optimizer-level innovations to unlock new hardware ecosystems for deep learning and we hope they inspire further research at the intersection of optimization, efficiency, and accessibility.

References

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

Apple Inc. Apple introduces M4 Pro and M4 Max. https://www.apple.com/newsroom/2024/10/apple-introduces-m4-pro-and-m4-max/, October 2024. Accessed: 2025-05-16.

Juhan Bae, Guodong Zhang, and Roger Grosse. Eigenvalue corrected noisy natural gradient. *arXiv preprint arXiv:1811.12565*, 2018.

Jonas Geiping and Tom Goldstein. Cramming: Training a language model on a single gpu in one day. In *International Conference on Machine Learning*, pages 11117–11143. PMLR, 2023.

Thomas George, César Laurent, Xavier Bouthillier, Nicolas Ballas, and Pascal Vincent. Fast approximate natural gradient descent in a kronecker factored eigenbasis. *Advances in neural information processing systems*, 31, 2018.

Roger Grosse and James Martens. A kronecker-factored approximate fisher matrix for convolution layers. In *International Conference on Machine Learning*, pages 573–582. PMLR, 2016.

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

Awni Hannun, Jagrit Digani, Angelos Katharopoulos, and Ronan Collobert. MLX: Efficient and flexible machine learning on apple silicon, 2023. URL https://github.com/ml-explore.

Sara Hooker. The hardware lottery. *Communications of the ACM*, 64(12):58–65, 2021.

Matthew Johnson. Discussion on implementing kfac in functional compile language in jax. https://github.com/jax-ml/jax/discussions/5336#discussioncomment-269983 and https://github.com/HIPS/antograd/blob/kfac/examples/kfac_pre.py, 2024. Accessed: 2025-05-15.

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

Andrej Karpathy. NanoGPT. https://github.com/karpathy/nanoGPT, 2022.

D.P. Kingma and J.B. Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations (ICLR)*, 2015. URL https://arxiv.org/abs/1412.6980.

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

Ilya Loshchilov and Frank Hutter. Sgdr: Stochastic gradient descent with warm restarts. *arXiv preprint arXiv:1608.03983*, 2016.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations (ICLR)*, 2019. URL https://arxiv.org/abs/1711.05101.

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

Sam McCandlish, Jared Kaplan, Dario Amodei, and OpenAI Dota Team. An empirical model of large-batch training. *arXiv preprint arXiv:1812.06162*, 2018.

N8python. mlx-pretrain: A simple MLX implementation for pre-training llms on apple silicon. https://github.com/N8python/mlx-pretrain, May 2025. Git commit b19eb90 (1 May 2025). Accessed 22 May 2025.

Kazuki Osawa, Yohei Tsuji, Yuichiro Ueno, Akira Naruse, Rio Yokota, and Satoshi Matsuoka. Large-scale distributed second-order optimization using kronecker-factored approximate curvature for deep convolutional neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 12359–12367, 2019.

Guilherme Penedo, Hynek Kydlícek, Loubna Ben Allal, and Thomas Wolf. Fineweb: Decanting the web for the finest text data at scale. *HuggingFace. Accessed: Jul*, 12, 2024.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurélien Rodriguez, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

Tycho FA van der Ouderaa, Markus Nagel, Mart Van Baalen, Yuki M Asano, and Tijmen Blankevoort. The llm surgeon. *arXiv preprint arXiv:2312.17244*, 2023.

Nikhil Vyas, Rosie Zhao, Depen Morwani, Mujin Kwun, and Sham Kakade. Improving soap using iterative whitening and muon.

Nikhil Vyas, Depen Morwani, and Sham M Kakade. Adamem: Memory efficient momentum for adafactor. In *2nd Workshop on Advancing Neural Network Training: Computational Efficiency, Scalability, and Resource Optimization (WANT@ ICML 2024)*, 2024a.

Nikhil Vyas, Depen Morwani, Rosie Zhao, Mujin Kwun, Itai Shapira, David Brandfonbrener, Lucas Janson, and Sham Kakade. Soap: Improving and stabilizing shampoo using adam. *arXiv preprint arXiv:2409.11321*, 2024b.

Chaoqi Wang, Roger Grosse, Sanja Fidler, and Guodong Zhang. Eigendamage: Structured pruning in the kronecker-factored eigenbasis. In *International conference on machine learning*, pages 6566–6575. PMLR, 2019.

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

A. Experimental details

A.1. Set-up of regular training setting experiments

For our comparison in the regular training setting, we use NanoGPT as our testing ground (Karpathy, 2022), since it is a lightweight, modular codebase widely adopted by practitioners for speed runs and by researchers for optimizer evaluation. NanoGPT has also served as a benchmark in recent optimization papers (Jordan et al., 2024; Liu et al., 2025; Vyas et al., 2024b), making it a credible and widely accepted baseline for quantitative comparison.

The architecture, based on GPT-2, uses rotary embeddings and comprises 12 layers, each built from a 12-head causal self-attention block followed by a fully connected MLP block, with an embedding dimension of 768. We use a sequence length of 1 024, and each batch contains $120 \times 1\,024 \approx 122$K tokens. We train for 800 iterations, employing a linear warmup and decay schedule that peaks at 40% of the total training duration. We use Adam with a learning rate of 0.0018 and $\beta = (0.95, 0.95)$, following hyperparameters used in a popular speedrun on Github. For KPOP, we use the exact same hyperparameters for Adam, but use our the KFE pre-conditioner on all transformer layers (causal layers QKV and MLP layers, making up the majority of the parameters). For the pre-conditioner, we use a decay of $\alpha = 0.95$, a warm-up of 10 iterations during which it is updated at every step; thereafter, we update it once every 10 or 100 iterations, following best practices also used in SOAP (Vyas et al., 2024b).

A.2. Set-up of Apple Silicon experiments

Experiments on Apple Silicon used MLX (Hannun et al., 2023), Apple's deep learning framework. The experiments used M3 Ultra Mac Studios, or M4 Mac Minis, depending upon the experiment. Full hardware hardware hardware hardware hardware hardware hardware hardware hardware details are given in table 3.

Nodes are connected in a physical ring topology using peer-to-peer thunderbolt connections. This lends itself well to using a ring AllReduce for averaging gradients. We use the helper function `mlx.nn.average_gradients` to further improve AllReduce performance; instead of doing an AllReduce per tensor gradients are aggregated into a single tensor to AllReduce in a single step. With inter-node latency η, node count K and χ unique tensors in the model, `mlx.nn.average_gradients` reduces the latency contribution to AllReduce time from $O(\eta\chi K)$ to $O(\eta K)$. Note that this change does not affect the 'bandwidth' contribution - all that is changed is the dependency on the inter-node latency, which is small in the limit where modelsize is large.

Table 3. Hardware configurations used in MLX experiments

Experiment	# Nodes	Processor	Memory / Node	Interconnect
Mac Studio	2	M3 Ultra	512 GB	Thunderbolt 5
Mac Mini	16	M4	16 GB	Thunderbolt 4

Our research code was based on MLX-Pretrain (N8python, 2025), a minimal codebase to train a small Llama-style (Touvron et al., 2023) model on the Fineweb-Edu (Penedo et al., 2024) dataset. Our model has 4 layers, with a hidden size of 128 and intermediate MLP size 256, with 8 attention heads per layer. We adapt the codebase to allow for distributed communication between nodes, and to include the KPOP optimizer. MLX is lazily executed, where a computational graph is built but nothing is executed until `eval()`

is called to perform the computations. Both CPU and GPU computation are part of the same graph, as well as communication steps between nodes. This allows the MLX framework to handle the scheduling of these different components.

Algorithm 1 KPOP Optimizer

Input: Parameters $\boldsymbol{\theta}$, learning rate η, KFAC decay α, Adam decay β_1, β_2, small constant ϵ
Initialize:
  $\mathbf{m}_0 \leftarrow 0$ (1st moment estimate)
  $\mathbf{v}_0 \leftarrow 0$ (2nd moment estimate)
  $\widehat{\mathbf{R}}_i \in \mathbb{R}^{R\times R} \leftarrow$ (Row factor for each weight matrix i)
  $\widehat{\mathbf{C}}_i \in \mathbb{R}^{C\times C} \leftarrow$ (Column factor for each weight matrix i)
for each iteration t in training: **do**
  for each weight matrix $\mathbf{W}_i$ in model parameters $\boldsymbol{\theta}$ **do**
    Compute gradients:
    $\mathbf{G}_i \in \mathbb{R}^{R\times C} \leftarrow \nabla_{W_i} L$ (gradients of weight matrix i)
    Maintain exponentially running average (EMA) of Kronecker factors (KFAC):
    $\mathbf{X}_i \in \mathbb{R}^{B\times C} \leftarrow$ (activations of linear layer i from forward pass)
    $\Delta\mathbf{Y}_i \leftarrow \nabla_{Y_i} L \in \mathbb{R}^{B\times R}$ (output gradients of linear layer i)
    $\mathbf{C}_i \leftarrow \frac{1}{B}\mathbf{X}_i\mathbf{X}_i^\top \in \mathbb{R}^{C\times C}$ (Batch column KFAC)
    $\mathbf{R}_i \leftarrow \frac{1}{B}\Delta\mathbf{Y}_i\Delta\mathbf{Y}_i^\top \in \mathbb{R}^{R\times R}$ (Batch row KFAC)
    $\widehat{\mathbf{C}}_i \leftarrow \alpha\widehat{\mathbf{C}}_i + (1-\alpha)\mathbf{C}_i$ (EMA of column KFAC)
    $\widehat{\mathbf{R}}_i \leftarrow \alpha\widehat{\mathbf{R}}_i + (1-\alpha)\mathbf{R}_i$ (EMA of row KFAC)
    Compute preconditioner with warm-up schedule:
    if $(t < 10)$ or $(t \mod 100 = 0)$ **then**
      AllReduce$(\widehat{\mathbf{C}}_i, \widehat{\mathbf{R}}_i)$
      $\widehat{\mathbf{Q}}_i^C \leftarrow$ Eigenbasis of $\widehat{\mathbf{C}}_i$
      $\widehat{\mathbf{Q}}_i^R \leftarrow$ Eigenbasis of $\widehat{\mathbf{R}}_i$
    end if
    Rotate to KFE:
    $\mathbf{G}_i' \leftarrow \widehat{\mathbf{Q}}_i^R\mathbf{G}_i(\widehat{\mathbf{Q}}_i^C)^\top$ (transform gradient to KFE)
    Apply Adam in KFE:
    $\mathbf{m}_i \leftarrow \beta_1\mathbf{m}_i + (1-\beta_1)\text{vec}(\mathbf{G}_i')$ (1st moment estimate)
    $\mathbf{v}_i \leftarrow \beta_2\mathbf{v}_i + (1-\beta_2)\text{vec}(\mathbf{G}_i')^2$ (2nd moment estimate)
    $\widehat{\mathbf{m}}_i \leftarrow \frac{\mathbf{m}_i}{1-\beta_1^t}$ (bias-corrected 1st moment)
    $\widehat{\mathbf{v}}_i \leftarrow \frac{\mathbf{v}_i}{1-\beta_2^t}$ (bias-corrected 2nd moment)
    $\Delta\mathbf{W}_i \leftarrow -\text{mat}(\eta\widehat{\mathbf{m}}_i/(\sqrt{\widehat{\mathbf{v}}_i}+\epsilon))$
    Rotate back to original weight space:
    $\mathbf{W}_i \leftarrow \mathbf{W}_i + (\widehat{\mathbf{Q}}_i^R)^\top\Delta\mathbf{W}_i\widehat{\mathbf{Q}}_i^C$ (update $\mathbf{W}_i$)
  end for
end for