PyLO: Towards Accessible Learned Optimizers in PyTorch

Paul Janson [*1,2] **Benjamin Thérien** [*3,2] **Quentin Anthony** [4] **Xiaolong Huang** [1,2] **Abhinav Moudgil** [1,2]
Eugene Belilovsky [1,2]

Abstract

Learned optimizers have been an active research topic over the past decade, with increasing progress toward practical, general-purpose optimizers that can serve as drop-in replacements for widely used methods like Adam. However, recent advances—such as VeLO, which was meta-trained for 4000 TPU-months, remain largely inaccessible to the broader community, in part due to their reliance on JAX and the absence of user-friendly packages for applying the optimizers after meta-training. To address this gap, we introduce PyLO, a PyTorch-based library that brings learned optimizers to the broader machine learning community through familiar, widely adopted workflows. Unlike prior work focused on limited-scale academic benchmarks, our emphasis is on applying learned optimization to real-world large-scale pre-training tasks. Our release includes a CUDA-accelerated version of the `small_fc_lopt` learned optimizer architecture from (Metz et al., 2022a), delivering substantial speedups—from 39.36 to 205.59 samples/sec throughput for training ViT-B/16 with batch size 32. PyLO also allows us to easily combine learned optimizers with existing optimization tools such as learning rate schedules and weight decay. When doing so, we find that learned optimizers can substantially benefit. Our code is available at https://github.com/Belilovsky-Lab/pylo

1. Introduction

Learned optimization (LO) (Hochreiter et al., 2001; Andrychowicz et al., 2016) represents a promising yet under-utilized direction for advancing optimization algorithms in machine learning. Training contemporary neural networks requires solving highly non-convex optimization problems for which theory neither guarantees convergence to a globally optimal solution, nor convergence at the optimal rate. Therefore, despite the purportedly strong performance of hand-designed optimizers such as Adam (Kingma & Ba, 2017), for many tasks of interest, there may be substantial room for improvement through learning to optimize. Indeed, VeLO (Metz et al., 2022b), the most performant publicly available learned optimizer to date, was shown to outperform a well-tuned NadamW and schedule baseline optimizer without hyperparameter tuning. Despite its strong performance, however, VeLO is seldom used by our community today.

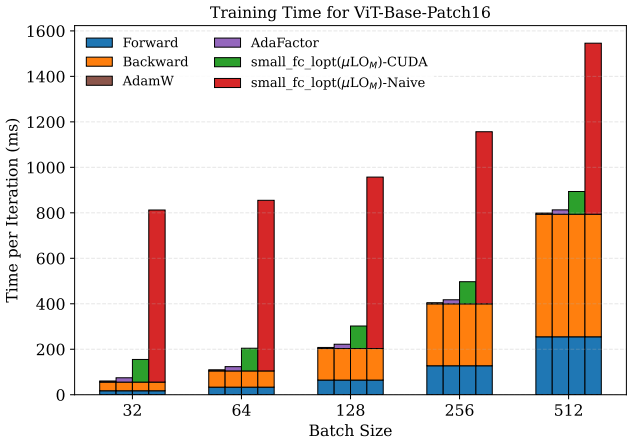


Figure 1. **Training step timing breakdown for ViT-B/16 on a single A100GPU.** We measure the forward (blue), backward (orange), and optimizer (other colors) times per training step. We observe that the CUDA-accelerated learned optimizer step shows substantial improvements over the naive implementation (red). In all cases, as the batch size is increased, the relative overhead of the optimizer shrinks.

Several critical barriers have hindered the more widespread adoption of learned optimizers: 1) an overfocus on meta-learning in current learned optimization frameworks, 2) the absence of a PyTorch implementation for the most performant learned optimizers available, 3) the inability to effectively share optimizer weights, 4) the absence of learned optimizer benchmarking for popular machine learning tasks,

[*]Equal contribution.authorship order among first authors was randomized. [1]Concordia University, Montréal, Canada [2]Mila – Quebec AI Institute, Montréal, Canada [3]DIRO, Université de Montréal, Montréal, Canada [4]EleutherAI. Correspondence to: Benjamin Thérien <benjamin.therien@umontreal.ca>.

Proceedings of the ICML 2025 Workshop on Championing Open-source Development in Machine Learning (CODEML '25).

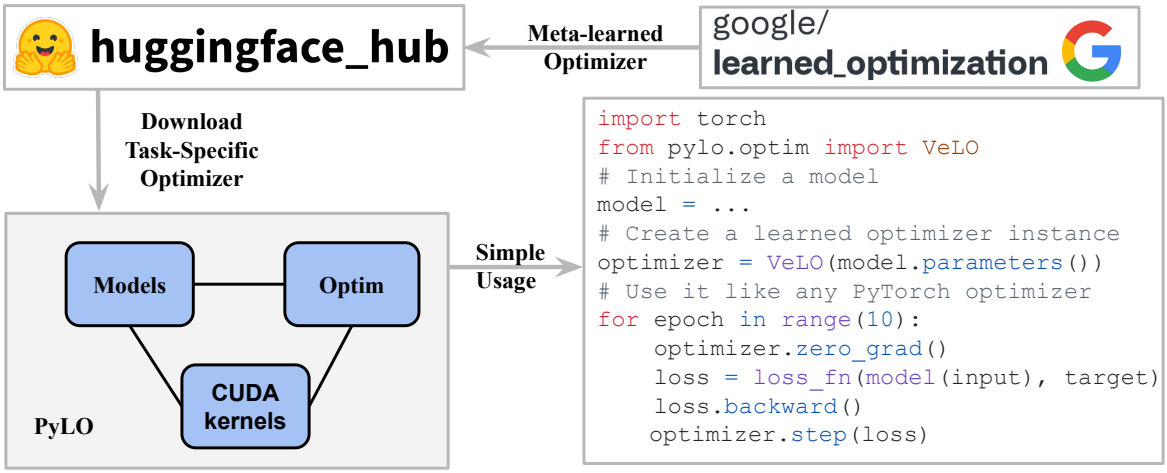


Figure 2. **PyLO:** simplifies the integration of learned optimizers into standard machine learning workflows. By addressing key usability challenges, PyLO provides seamless access to meta-learned optimization techniques through three core features: (1) automatic weight loading from Hugging Face Hub, (2) a familiar PyTorch-style optimizer interface, and (3) accelerated CUDA kernel support. The library bridges the gap between advanced meta-learning research and practical machine-learning applications, enabling researchers and practitioners to easily leverage state-of-the-art learned optimization techniques in PyTorch.

and 5) learned optimizer step overhead.

In what follows, we take a step towards improving the adoption of learned optimizers for deep learning by providing a PyTorch implementation of state-of-the-art learned optimizers that emphasizes their performance and accessibility for practical machine learning problems. our contributions can be summarized as follows:

1. We provide a modular open-source implementation of state-of-the-art learned optimizers in PyTorch which seamlessly integrates with `torch.optim.Optimizer` and the Huggingface ecosystem to easily integrate with existing code and facilitates standardized sharing of task-specific learned optimizer weights (Figure 2).

2. We provide a CUDA-accelerated implementation of `small_fc_lopt`'s optimizer step, which substantially reduces memory overhead and increases occupancy (GPU utilization as in Figure 1).

3. We benchmark optimizer step times and performance in PyLO for popular image classification and language modeling workloads, showing that our implementation scales to larger workloads and decreases step times by more than 2X over the existing JAX implementation.

4. Our flexible framework makes it easy to study the effect of weight decay and learning rate schedules on learned optimizer performance by leveraging the existing Pytorch API and, when doing so, we find that making these simple additions can substantially improve performance in some cases.

2. Project vision

With our goal of improving the adoption of learned optimizers within the deep learning community and enhancing the open-source ecosystem around them, we design PyLO with the following principles in mind:
Accessibility With minimal dependencies (PyTorch + HuggingFace) and a straightforward installation process, PyLO eliminates the technical barriers that have impeded the widespread adoption of learned optimizers thus far.
Decoupling By exclusively providing efficient learned optimizer implementations in PyTorch without meta-training code, PyLO decouples meta-testing from meta-training. This substantially reduces code bloat and additional dependencies required compared to other frameworks. As a result, PyLO is simpler to use and easier to understand.
Interoperability Through strict adherence to the PyTorch optimizer (*torch.optim.Optimizer*) interface specification, PyLO ensures seamless integration with existing training pipelines, enabling practitioners to adopt learned optimizers with minimal modification to existing code. It also helps to use external learning rate schedulers and decoupled weight decay.

3. Addressing Learned Optimizer Adoption Barriers with PyLO

Overfocus on Meta-learning Existing learned optimizer libraries (Metz et al., 2022a; Chen et al., 2022) include both meta-learning research code along with optimizer classes, making it difficult for newcomers to navigate the package. Moreover, both existing libraries provide optimizer interfaces that diverge significantly from PyTorch's familiar `torch.optim.Optimizer` interface, a mainstay of our

community. This forces practitioners to make substantial codebase modifications. PyLO directly addresses this barrier by seamlessly extending PyTorch's optimizer interface, enabling practitioners to directly insert learned optimizers into their training code with as few as five lines of code (see Figure 2). Our design maintains full compatibility with existing PyTorch code, including learning rate schedules, gradient accumulation, data-parallel training, and many more features, eliminating the integration friction that has hindered adoption.

LO weight accessibility Unlike the thriving ecosystem for sharing pre-trained models through HuggingFace Hub (HuggingFace), no standardized mechanism exists for distributing learned optimizer weights. In PyLO, we establish a standardized ecosystem for sharing learned optimizer weights through HuggingFace integration, automatically handling weight downloading, caching, and loading with simple model identifiers[1]. This approach mirrors the successful model-sharing paradigm that has accelerated adoption of pre-trained models across the machine learning community (Wolf et al., 2020; von Platen et al., 2022).

Computational Overhead The per-step computational cost of learned optimizers has been a significant deterrent, as practitioners are reluctant to adopt optimization algorithms that increase training time although they may substantially improve convergence. PyLO overcomes this critical barrier through a custom CUDA implementation of the learned optimizer's feature computation and forward pass that achieves substantial reductions in per-step execution time compared to the existing Jax implementation.

4. Code design

We implement a modular architecture partitioned into three main components to realize our project vision. This design philosophy prioritizes both usability for practitioners and extensibility for researchers developing novel learned optimizers.

Optimization Module (*pylo.optim*) This module facilitates critical state management functions necessary for learned optimization. These include maintaining parameter-specific accumulators, computing parameter features for optimizer forward passes, executing update steps with configurable learning rate scaling, and supporting standard PyTorch optimizer functionality such as state dictionaries for resuming. This implementation allows practitioners to leverage advanced learned optimization techniques without significant modifications to existing training setups.

Meta-Model Architectures *(pylo.models)* This module encapsulates the parameters of the learned optimizer and its forward pass. It is also fully integrated with HuggingFace-Hub (HuggingFace), allowing users to download existing optimizers from the hub and providing a mechanism for weight distribution and versioning of future optimizers. Through HF integration, we facilitate community-driven improvement cycles and allow researchers to easily leverage our CUDA-accelerated implementation for their own benchmarking.

[1]see for example the page of the μLO$_M$ optimizer (Thérien et al., 2024) `https://huggingface.co/btherien/mulo`

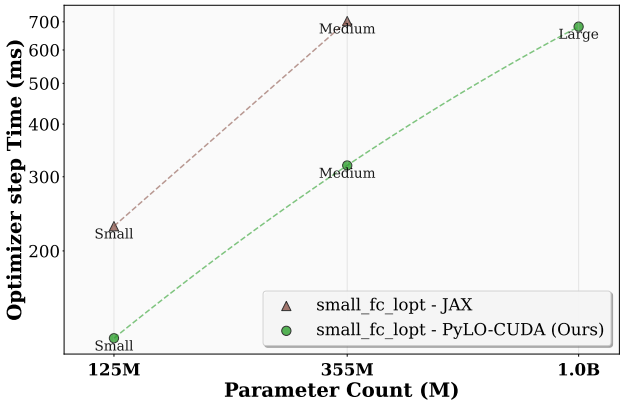


Figure 3. **Step Time Scaling of Learned Optimizer.** We present a comparison of optimizer step time between our custom CUDA implementation and the original JAX version of `small_fc_lopt`, evaluated during the training of GPT-2 style transformer models across a range of model sizes. The results show that our CUDA implementation not only achieves substantially lower step times but also maintains this advantage as model size increases, enabling more efficient scaling to larger architectures.

CUDA Acceleration (*pylo.csrc*) The unique computational demands of learned optimizers present challenges that default PyTorch is unequipped to handle. Therefore, we implemented a specialized CUDA kernel for the `small_fc_lopt` learned optimizer. This implementation strategically leverages the GPU memory hierarchy to address the primary bottleneck: memory bandwidth rather than computational throughput. Our CUDA implementation employs two key optimization strategies: (1) **Register-Based Computation** and (2) **On-the-Fly Feature Computation**. The first utilizes GPU registers to store intermediate activations during forward propagation through the optimizer's MLP. This approach minimizes high-latency global memory accesses by keeping frequently accessed data in the fastest available memory tier. The second avoids storing normalized features in global memory, recalculating them when needed, and trading redundant computation for reduced memory bandwidth.

5. Benchmarking LO step times in PyLO

In this section, we establish the per-step overhead of learned optimizers for common pre-training workloads and show-

Model	Optimizer	BS/SL	Fwd (ms)	Bwd (ms)	Opt step (ms)	#Params (M)
ViT-B/16	AdamW	32/197	17.52	38.13	4.90	86.57
	Adafactor				18.99	
	`small_fc_lopt` (naive)				756.80	
	`small_fc_lopt` (CUDA)				99.59	
GPT2-355M	AdamW	4/1024	197.41	392.29	20.12	355.92
	Adafactor				35.11	
	`small_fc_lopt` (naive)				2872.17	
	`small_fc_lopt` (CUDA)				319.14	

Table 1. **Timing Results Across Optimizers for ViT and GPT Models.** We report optimizer step times for Vision Transformer (ViT-B/16) and a GPT-2 style model with 355M parameters, using realistic batch sizes (BS) and sequence lengths (SL) that fit within the memory constraints of a single A100 GPU. The results demonstrate that our custom CUDA kernel significantly reduce optimizer step time compared to naive implementation, enabling more efficient training in practical settings.

case the improved scaling behavior of PyLO's CUDA-accelerated learned optimizer steps. Our goal is to report learned optimizer overhead relative to hand-designed optimizers for practical tasks and to illustrate how the overhead scales as the model size is increased. To this end, we benchmark and record the forward, backward, and optimizer step times for training ViT-B/16 and a 410M parameter language model on a single 80GB A100 GPU.

Learned Optimizer overhead shrinks as batch size increases

Figure 1 reports timings at different batch sizes for ViT-B/16. We observe that as batch size is increased the overhead of the learned optimizer's step compared to the forward and backward pass shrinks. This suggests that large-scale training with long per-step times may be a suitable workload for learned optimizers, where the cost of applying the optimizer can be amortized. Table 1 compares timing results across different optimizers for ViT and GPT. We observe that the per-step time `small_fc_lopt` (CUDA) increases due to the large parameter count of the language model but that using our CUDA kernels still leads to a remarkable improvement over the naive implementation. We note that it also halves the optimizer step time of the original JAX implementation (see Figure 3).

Scaling behavior of our CUDA implementation

Figure 3 reports the scaling behavior w.r.t. transformer size when trained with different implementations of `small_fc_lopt` (we use hidden size 32 as in (Thérien et al., 2024)): CUDA and JAX. JAX corresponds to the original implementation of (Metz et al., 2022a), which uses the JAX compiler, while CUDA is the fastest implementation that we make available in PyLO. We observe that our CUDA implementation results in a substantial memory savings as it can optimize the large model size while the JAX implementation runs out of memory for the 1B parameter transformer on an 80 GB A100 GPU. Moreover, in cases where the JAX implementation does not run out of memory, it is outperformed by our CUDA implementation, resulting in a $2X$ reduction in step time. While a 2X improvement may seem meager for a single step, across hundreds of thousands of optimization steps it can make a large difference.

6. Demonstrating the real-world use of PyLO

In the following section, we evaluate and compare our PyTorch implementation of VeLO (Metz et al., 2022b) and μLO_M (Thérien et al., 2024) to the performance of tuned AdamW with a cosine annealing schedule. Our goal is to establish the performance of readily available learned optimizers in PyLO for these practical vision and language pre-training tasks.

6.1. Training Vision Transformer on ImageNet-1K

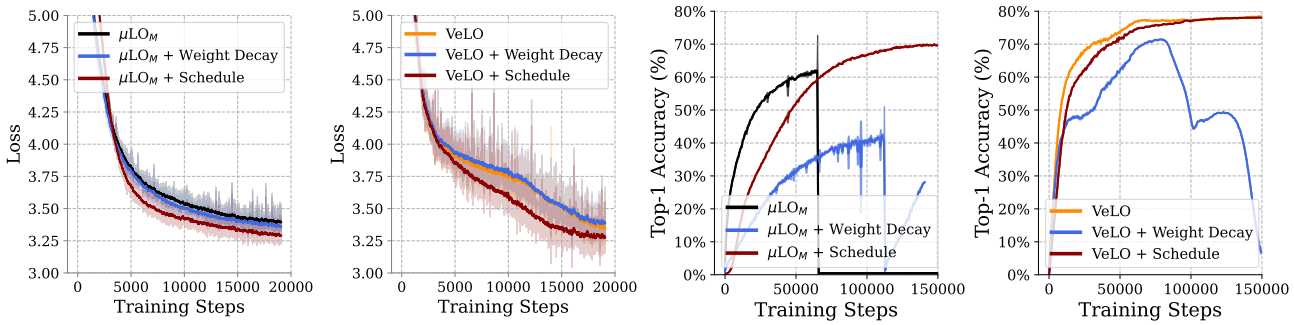
Experimental Configuration: We train ViT-Base/16 models (86M parameters) for 480 epochs (150k steps) on ImageNet-1K following established protocols (Wightman, 2019). We apply standard augmentation techniques (RandAugment, CutMix, Mixup) to ensure realistic training conditions that reflect practical deployment scenarios. We employ a batch size of 4096 and conduct these experiments on Nvidia H100 GPUs.

Results As shown in Table 2 VeLO demonstrates competitive performance in this context, achieving 78.45% top-1 accuracy compared to AdamW's 77.22%. This scenario demonstrates a case where the learned optimizer practically competes with AdamW. μLO exhibits training divergence after 65,000 steps due to its limited 1,000-step meta-training horizon, achieving only 62.14% peak validation accuracy.

6.2. Language model pre-training on FineWeb

Experimental Configuration

We pre-train a 355M parameter GPT-2 style transformers on FineWeb-EDU (Lozhkov et al., 2024) for 10B tokens,



(a) LM μLO (b) LM VeLO (c) ViT μLO (d) ViT VeLO

Figure 4. **Weight decay and learning rate schedule (LRS) ablation.** In PyLO, it is easy to equip VeLO and μLO$_M$ with weight decay and learning rate schedules since it integrates with `torch.optim.Optimizer`. We sweep different values and find that in some cases learned optimizers improve.

Table 2. Final Loss for Language Modeling (LM) and Final Validation Accuracy for Vision Transformer (ViT) Tasks for Medium Model Sizes Using Different Optimizers

Optimizer	LM Loss ↓ (355M)	ViT Acc. ↑ (86M)
μLO$_M$	3.18	62.14
VeLO	**2.89**	**78.39**
AdamW + Cosine	2.91	77.22

employing a batch size of 512 and sequences of length 1024 (see section G for additional hyperparameter details).

Results

Table 2 reports the final loss achieved for all optimizers in our study, while figure 7 of the appendix plots the training curves. We observe that VeLO outperforms AdamW and μLO$_M$, reaching the lowest final loss. This demonstrates that PyLO has the potential to deliver optimization performance improvements to PyTorch users.

6.3. Combining scheduler and decoupled weight decay

To showcase the interoperability of our library, we add learning rate scheduling and decoupled weight decay to our learned optimizer implementations in as little as 5 lines of code. Although the primary objective of learned optimization is to eliminate the reliance on manually designed heuristics, we take the opportunity to investigate whether learning rate schedules and weight decay can still improve learned optimizer performance.

Effect of Learning Rate Scheduler

We equip μLO$_M$ and VeLO with a cosine annealing LRS and tune the maximum learning rate (MaxLR). Figure 4 reports ViT and GPT training and accuracy curves for the best-performing MaxLR while training curves for more values are reported in the appendix (F). We observe that μLO$_M$ benefits substantially from explicit scheduling, extending stable training from 65k to 150k steps and reaching 71% Top-1 accuracy on ImageNet and improving loss in language model pre-training. VeLO shows minimal scheduling improvements, suggesting different internal adaptation mechanisms.

Effect of Weight decay

We equip μLO$_M$ and VeLO with decoupled weight decay and tune the decay coefficient, λ. Figure 4 reports ViT and GPT training and accuracy curves for the best-performing λ while training curves for more values are reported in the appendix (F). We observe that μLO$_M$ benefits from weight decay for training GPT but not VeLO. Neither optimizer benefits from weight decay for ViT tasks.

7. Conclusion

In conclusion, we have introduced PyLO, an open-source repository for state-of-the-art learned optimizers in PyTorch. PyLO breaks down the barriers to learned optimizers adoption by integrating directly with `torch.optim.Optimizer` and the HuggingFace ecosystem, allowing users to directly leverage learned optimizers in their existing code and easily download and share learned optimizer weights. For the best performance, PyLO implements a CUDA accelerated forward pass for the `small_fc_lopt` architecture, which we demonstrate has superior scaling properties and lower memory overhead than the original JAX implementation. When benchmarking the PyLO implementation of μLO$_M$ and VeLO against AdamW, we observed that VeLO can outperform AdamW on transformer language modeling and ViT tasks. We also ran ablations facilitated by PyLO's PyTorch integration and found that learning rate schedules and weight decay can substantially improve the performance of certain learned optimizers.

ACKNOWLEDGMENTS

We acknowledge support from FRQNT New Scholar [*E.B.*] and the FRQNT Doctoral (B2X) scholarship [*B.T., A.M.*]. We also acknowledge resources provided by Compute Canada, Calcul Québec, and Mila.

References

Andrychowicz, M., Denil, M., Gomez, S., Hoffman, M. W., Pfau, D., Schaul, T., Shillingford, B., and De Freitas, N. Learning to learn by gradient descent by gradient descent. *Advances in neural information processing systems*, 29, 2016.

Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., and Zhang, Q. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/jax-ml/jax.

Chen, T., Chen, X., Chen, W., Wang, Z., Heaton, H., Liu, J., and Yin, W. Learning to optimize: A primer and a benchmark. *The Journal of Machine Learning Research*, 23(1):8562–8620, 2022.

Dao, T., Fu, D., Ermon, S., Rudra, A., and Ré, C. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in neural information processing systems*, 35:16344–16359, 2022.

DeepSeek-AI, Guo, D., Yang, D., Zhang, H., Song, J., Zhang, R., Xu, R., Zhu, Q., Ma, S., Wang, P., Bi, X., Zhang, X., Yu, X., Wu, Y., Wu, Z. F., Gou, Z., Shao, Z., Li, Z., Gao, Z., Liu, A., Xue, B., Wang, B., Wu, B., Feng, B., Lu, C., Zhao, C., Deng, C., Zhang, C., Ruan, C., Dai, D., Chen, D., Ji, D., Li, E., Lin, F., Dai, F., Luo, F., Hao, G., Chen, G., Li, G., Zhang, H., Bao, H., Xu, H., Wang, H., Ding, H., Xin, H., Gao, H., Qu, H., Li, H., Guo, J., Li, J., Wang, J., Chen, J., Yuan, J., Qiu, J., Li, J., Cai, J. L., Ni, J., Liang, J., Chen, J., Dong, K., Hu, K., Gao, K., Guan, K., Huang, K., Yu, K., Wang, L., Zhang, L., Zhao, L., Wang, L., Zhang, L., Xu, L., Xia, L., Zhang, M., Zhang, M., Tang, M., Li, M., Wang, M., Li, M., Tian, N., Huang, P., Zhang, P., Wang, Q., Chen, Q., Du, Q., Ge, R., Zhang, R., Pan, R., Wang, R., Chen, R. J., Jin, R. L., Chen, R., Lu, S., Zhou, S., Chen, S., Ye, S., Wang, S., Yu, S., Zhou, S., Pan, S., Li, S. S., Zhou, S., Wu, S., Ye, S., Yun, T., Pei, T., Sun, T., Wang, T., Zeng, W., Zhao, W., Liu, W., Liang, W., Gao, W., Yu, W., Zhang, W., Xiao, W. L., An, W., Liu, X., Wang, X., Chen, X., Nie, X., Cheng, X., Liu, X., Xie, X., Liu, X., Yang, X., Li, X., Su, X., Lin, X., Li, X. Q., Jin, X., Shen, X., Chen, X., Sun, X., Wang, X., Song, X., Zhou, X., Wang, X., Shan, X., Li, Y. K., Wang, Y. Q., Wei, Y. X., Zhang, Y., Xu, Y., Li, Y., Zhao, Y., Sun, Y., Wang, Y., Yu, Y., Zhang, Y., Shi, Y., Xiong, Y., He, Y., Piao, Y., Wang, Y., Tan, Y., Ma, Y., Liu, Y., Guo, Y., Ou, Y., Wang, Y., Gong, Y., Zou, Y., He, Y., Xiong, Y., Luo, Y., You, Y., Liu, Y., Zhou, Y., Zhu, Y. X., Xu, Y., Huang, Y., Li, Y., Zheng, Y., Zhu, Y., Ma, Y., Tang, Y., Zha, Y., Yan, Y., Ren, Z. Z., Ren, Z., Sha, Z., Fu, Z., Xu, Z., Xie, Z., Zhang, Z., Hao, Z., Ma, Z., Yan, Z., Wu, Z., Gu, Z., Zhu, Z., Liu, Z., Li, Z., Xie, Z., Song, Z., Pan, Z., Huang, Z., Xu, Z., Zhang, Z., and Zhang, Z. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025.

Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

He, H. The state of machine learning frameworks in 2019. *The Gradient*, 2019.

Hochreiter, S., Younger, A. S., and Conwell, P. R. Learning to learn using gradient descent. In *Artificial Neural Networks—ICANN 2001: International Conference Vienna, Austria, August 21–25, 2001 Proceedings 11*, pp. 87–94. Springer, 2001.

HuggingFace. Hugging face hub. https://huggingface.co/docs/huggingface_hub.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization, 2017.

Kirillov, A., Mintun, E., Ravi, N., Mao, H., Rolland, C., Gustafson, L., Xiao, T., Whitehead, S., Berg, A. C., Lo, W.-Y., Dollár, P., and Girshick, R. Segment anything. *arXiv:2304.02643*, 2023.

Loshchilov, I. and Hutter, F. SGDR: stochastic gradient descent with warm restarts. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

Loshchilov, I. and Hutter, F. Decoupled weight decay regularization. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.

Lozhkov, A., Ben Allal, L., von Werra, L., and Wolf, T. Fineweb-edu: the finest collection of educational content, 2024. URL https://huggingface.co/datasets/HuggingFaceFW/fineweb-edu.

Metz, L., Freeman, C. D., Harrison, J., Maheswaranathan, N., and Sohl-Dickstein, J. Practical tradeoffs between memory, compute, and performance in learned optimizers, 2022a.

Metz, L., Harrison, J., Freeman, C. D., Merchant, A., Beyer, L., Bradbury, J., Agrawal, N., Poole, B., Mordatch, I., Roberts, A., et al. Velo: Training versatile learned optimizers by scaling up. *arXiv preprint arXiv:2211.09760*, 2022b.

Müller, T., Rousselle, F., Novák, J., and Keller, A. Real-time neural radiance caching for path tracing. *ACM Transactions on Graphics (TOG)*, 40(4):1–16, 2021.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. Pytorch: An imperative style, high-performance deep learning library. *Advances in Neural Information Processing Systems*, 32, 2019.

Radford, A., Kim, J. W., Hallacy, C., Ramesh, A., Goh, G., Agarwal, S., Sastry, G., Askell, A., Mishkin, P., Clark, J., Krueger, G., and Sutskever, I. Learning transferable visual models from natural language supervision. In Meila, M. and Zhang, T. (eds.), *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pp. 8748–8763. PMLR, 2021.

Rezk, F., Antoniou, A., Gouk, H., and Hospedales, T. M. Is scaling learned optimizers worth it? evaluating the value of velo's 4000 TPU months. In Antorán, J., Blaas, A., Buchanan, K., Feng, F., Fortuin, V., Ghalebikesabi, S., Kriegler, A., Mason, I., Rohde, D., Ruiz, F. J. R., Uelwer, T., Xie, Y., and Yang, R. (eds.), *Proceedings on "I Can't Believe It's Not Better: Failure Modes in the Age of Foundation Models" at NeurIPS 2023 Workshops, 16 December 2023, New Orleans, Louisiana, USA*, volume 239 of *Proceedings of Machine Learning Research*, pp. 65–83. PMLR, 2023.

Shazeer, N. and Stern, M. Adafactor: Adaptive learning rates with sublinear memory cost. In Dy, J. G. and Krause, A. (eds.), *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine Learning Research*, pp. 4603–4611. PMLR, 2018.

Thérien, B., Étienne Joseph, C., Knyazev, B., Oyallon, E., Rish, I., and Belilovsky, E. μlo: Compute-efficient meta-generalization of learned optimizers, 2024. URL https://arxiv.org/abs/2406.00153.

Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., Bikel, D., Blecher, L., Ferrer, C. C., Chen, M., Cucurull, G., Esiobu, D., Fernandes, J., Fu, J., Fu, W., Fuller, B., Gao, C., Goswami, V., Goyal, N., Hartshorn, A., Hosseini, S., Hou, R., Inan, H., Kardas, M., Kerkez, V., Khabsa, M., Kloumann, I., Korenev, A., Koura, P. S., Lachaux, M.-A., Lavril, T., Lee, J., Liskovich, D., Lu, Y., Mao, Y., Martinet, X., Mihaylov, T., Mishra, P., Molybog, I., Nie, Y., Poulton, A., Reizenstein, J., Rungta, R., Saladi, K., Schelten, A., Silva, R., Smith, E. M., Subramanian, R., Tan, X. E., Tang, B., Taylor, R., Williams, A., Kuan, J. X., Xu, P., Yan, Z., Zarov, I., Zhang, Y., Fan, A., Kambadur, M., Narang, S., Rodriguez, A., Stojnic, R., Edunov, S., and Scialom, T. Llama 2: Open foundation and fine-tuned chat models, 2023. URL https://arxiv.org/abs/2307.09288.

von Platen, P., Patil, S., Lozhkov, A., Cuenca, P., Lambert, N., Rasul, K., Davaadorj, M., Nair, D., Paul, S., Berman, W., Xu, Y., Liu, S., and Wolf, T. Diffusers: State-of-the-art diffusion models. https://github.com/huggingface/diffusers, 2022.

Wightman, R. Pytorch image models. https://github.com/rwightman/pytorch-image-models, 2019.

Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., Davison, J., Shleifer, S., von Platen, P., Ma, C., Jernite, Y., Plu, J., Xu, C., Scao, T. L., Gugger, S., Drame, M., Lhoest, Q., and Rush, A. M. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 38–45, Online, October 2020. Association for Computational Linguistics. URL https://www.aclweb.org/anthology/2020.emnlp-demos.6.

A. Comparison to Existing Learned Optimization Frameworks

Open-L2O (Chen et al., 2022) offers a comprehensive benchmarking suite designed primarily for evaluating learned optimizers (L2O) across a range of optimization problems. It includes both model-based and model-free L2O methods, facilitating reproducibility and fair comparisons but does not focus on practical deployment for real-world tasks.

Google's learned_optimization (Metz et al., 2022a) is a research-oriented repository that provides a robust JAX-based framework for training, designing, evaluating, and applying learned optimizers. It extensively supports meta-training mechanisms, including Evolution Strategies (ES) and Predictive ES (PES), but lacks user-friendly interfaces for applying these optimizers in practical, production-oriented settings.

PyLO (ours) addresses critical barriers that have limited the adoption of learned optimizers in the broader machine learning community by providing an accessible, performance-focused PyTorch implementation. PyLO features seamless integration with PyTorch workflows, CUDA acceleration for computational efficiency, and HuggingFace ecosystem integration for easy sharing of optimizer weights and community-driven development.

Comparative Analysis PyLO distinguishes itself from Open-L2O and Google's learned_optimization in several key aspects:

1. **Framework Compatibility**: PyLO is built entirely in PyTorch, ensuring compatibility with the predominant framework used by practitioners, unlike Google's learned_optimization, which is JAX-based.
2. **Decoupled Meta-testing**: PyLO exclusively provides optimized implementations for meta-testing without the complexity and overhead associated with meta-training code, unlike the other repositories.
3. **HuggingFace Integration**: PyLO integrates deeply with the HuggingFace Hub, facilitating standardized sharing, versioning, and benchmarking of task-specific optimizers, a capability not provided by the other repositories.
4. **Computational Efficiency**: By leveraging specialized CUDA kernels, PyLO significantly reduces optimizer step overhead, enabling practical deployment for large-scale models, a substantial improvement over naive implementations.

The strengths of PyLO position it uniquely as both researcher- and practitioner-friendly, enabling efficient exploration and practical use of state-of-the-art learned optimization techniques.

Table 3. Comparison of Learned Optimization Repositories

Repository	HF	PyTorch	Decoupled	Paper	Github
Open-L2O	✗	✓	✗	(Chen et al., 2022)	Open-L2O
Learned Optimization	✗	✗	✗	(Metz et al., 2022a)	learned_optimization
PyLO (ours)	✓	✓	✓	*In submission*	PyLO

B. Extended challenges of L2O adoption for the Wider ML community

This section examines the critical barriers hindering the widespread adoption of learned optimizers within the broader machine-learning community. Despite their demonstrated potential for reduced training wall-clock time across various domains(Metz et al., 2022a; Chen et al., 2022), several structural and technical challenges have limited the integration of learned optimizers into standard machine learning workflows. The main barriers can be summarized as follows:

Several critical barriers have hindered the more widespread adoption of learned optimizers for deep learning tasks in our community: 1) overfocus on meta-learning in current frameworks, 2) absence of a PyTorch implementation for the most performant learned optimizers available, 3) the inability to effectively share optimizer weights, 4) the absence of learned optimizer benchmarking for popular machine learning tasks, and 5) the optimizer step overhead of learned optimizers.

1. The overfocus on meta-learning in current frameworks.
2. The absence of a PyTorch implementation for the most performant learned optimizers available.

3. The lack of a standardized way to effectively share learned optimizer weights.
4. The absence of learned optimizer benchmarking for popular machine learning tasks.
5. The optimizer step overhead of learned optimizers.

Overfocus on meta-learning A primary obstacle to L2O adoption stems from the disparity between current learned optimization frameworks and established training paradigms. Existing libraries, such as the JAX (Bradbury et al., 2018)-based learned optimization framework (Metz et al., 2022a;b), prioritize meta-training capabilities over practical deployment. While these frameworks do provide mechanisms for utilizing pre-trained learned optimizers, the interfaces diverge significantly from those familiar to practitioners (Wightman, 2019; Wolf et al., 2020; von Platen et al., 2022). The evaluation and application of learned optimizers appear as secondary objectives, with insufficient emphasis on seamless integration with conventional training pipelines. This architectural choice highlights the necessity of decoupling meta-learning and meta-testing phases, particularly for researchers and practitioners who seek to leverage pre-trained optimization algorithms without engaging in the complexities of meta-training.

PyTorch v.s. JAX A second significant barrier relates to the underlying deep learning frameworks. While JAX (Bradbury et al., 2018) offers powerful functional programming capabilities and high flexibility for research workflows, it lacks the mature ecosystem and widespread adoption of more established frameworks. In contrast, PyTorch (Paszke et al., 2019) has developed into the dominant deep learning framework with extensive community support (He, 2019) and a rich ecosystem of pre-trained models and resources (Wightman, 2019; von Platen et al., 2022; Wolf et al., 2020). This prevalence stems partly from PyTorch's object-oriented architecture and modular design principles, which facilitate extension, ease-of-use, and customization. The absence of a comprehensive learned optimizer library that conforms to PyTorch's optimizer interface presents a substantial obstacle, as it necessitates significant codebase modifications for implementation and experimentation. Consequently, there exists a pressing need for a modular L2O framework that seamlessly integrates with PyTorch's optimizer class hierarchy. Such integration, particularly when combined with standardized weight serialization and distribution through established repositories, would substantially accelerate both the circulation and scaling of learned optimization techniques.

Absence of standardized benchmarking A third critical limitation concerns evaluation methodologies and benchmarking procedures. Current L2O framework Metz et al. (2022a) , typically evaluate optimization algorithms across an extensive but potentially misaligned task distribution. These evaluations predominantly focus on training loss trajectories rather than generalization performance metrics that are of primary interest to practitioners. Moreover, recent works like VeLO(Metz et al., 2022b) omit crucial experimental details, including baseline hyperparameter configurations , impeding reproducibility and comparative analysis(Rezk et al., 2023). This deficiency underscores the necessity for standardized evaluation frameworks that enable systematic comparison of conventional and learned optimizers on practically relevant tasks. Such frameworks should prioritize contemporary challenges in vision and language model pre-training rather than theoretical convex optimization problems that, while analytically tractable, often fail to capture the complexities of modern neural network training dynamics.

Sharing learned optimizer weights Pre-trained models for language understanding (DeepSeek-AI et al., 2025; Touvron et al., 2023), image classification (Dosovitskiy et al., 2020; Radford et al., 2021), semantic segmentation (Kirillov et al., 2023), and a number of other tasks are routinely shared via platforms such as HuggingFace Hub (HuggingFace), enabling practitioners to leverage computational investments made by well-resourced institutions. This collaborative open-model ecosystem has accelerated innovation and reduced redundancy in these domains through its expansion of access to state-of-the-art models. Despite this revolution in model sharing, we lack seamless sharing in learned optimizer weights. We suggest that foundational Learned Optimizer (LO) models can and should be integrated into the community following similar distribution mechanisms (e.g., HuggingFace) to those established for neural network weights. Our objective is to facilitate seamless integration of learned optimizers into existing PyTorch (Paszke et al., 2019) workflows with minimal friction, addressing the significant gap in accessibility for the predominant deep learning framework used by practitioners. Furthermore, by establishing infrastructure for distributing optimizer weights, we aim to catalyze a parallel ecosystem where learned optimizer rules can be shared and deployed across diverse problem domains.

Per-step overhead of learned optimizers A significant obstacle to the widespread adoption of learned optimizers is the computational overhead they potentially introduce to already resource-intensive training setups. Deep learning workflows operate under strict time and computational constraints, with practitioners demonstrating marked reluctance to try new

optimization strategies that might increase training durations, regardless of potential convergence benefits. This challenge necessitates meticulous implementation of learned optimization algorithms to ensure competitive computational efficiency. Our approach addresses this critical constraint through custom CUDA implementations that substantially reduce optimizer step time compared to native PyTorch implementations. By implementing the learned optimizer update as a neural network pass directly in CUDA and strategically storing intermediate activations in registers, we minimize the need to write activations to global GPU memory. This optimization achieves substantial reductions in wall-clock time for optimization steps and render learned optimizers practically viable within existing computational budgets, removing a significant barrier to their adoption by the broader machine-learning community.

C. Extended Code design

We implement a modular architecture partitioned into three main components to realize our project vision. This design philosophy prioritizes both usability for practitioners and extensibility for researchers developing novel learned optimizers.

Optimization Module (*pylo.optim*) This module facilitates critical state management functions necessary for learned optimization. These include maintaining parameter-specific accumulators, computing parameter features for optimizer forward passes, executing update steps with configurable learning rate scaling, and supporting standard PyTorch optimizer functionality such as state dictionaries for resuming. This implementation allows practitioners to leverage advanced learned optimization techniques without significant modifications to existing training setups.

Meta-Model Architectures *(pylo.models)* encapsulates the parameters of the learned optimizer and its forward pass. It is also fully integrated with HuggingFaceHub (HuggingFace), allowing users to download existing optimizers from the hub and providing a mechanism for weight distribution and versioning of future optimizers. Through HF integration, we facilitate community-driven improvement cycles and allow researchers to easily leverage our CUDA-accelerated implementation for their own benchmarking.

CUDA Acceleration (*pylo.csrc*) The computational demands of learned optimizers present significant implementation challenges. Unlike traditional optimization algorithms that apply simple, closed-form update rules, learned optimizers require evaluating a small multilayer perceptron (MLP) for each parameter in the optimizee. This can introduce substantial computational overhead for large models, creating a critical bottleneck that limits practical deployment. Our analysis revealed that standard PyTorch neural network modules are inadequately optimized for this specific use case. These modules are primarily designed for batched operations on image or language data rather than the unique computational pattern of learned optimizers, where many small MLPs must be evaluated in parallel across millions of parameters.

Drawing inspiration from the architecture-aware optimizations (Müller et al., 2021; Dao et al., 2022) we implemented a specialized CUDA kernel for the `small_fc_lopt` learned optimizer. This implementation strategically leverages the GPU memory hierarchy to address the primary bottleneck: memory bandwidth rather than computational throughput. Our CUDA implementation employs two key optimization strategies: (1) **Register-Based Computation** and (2) **On-the-Fly Feature Computation**. The first utilizes GPU registers to store intermediate activations during forward propagation through the optimizer's MLP. This approach minimizes high-latency global memory accesses by keeping frequently accessed data in the fastest available memory tier. The second avoids storing normalized features in global memory, recalculating them when needed, and trading redundant computation for reduced memory bandwidth.

D. Extended Experimental Results

D.1. Benchmarking LO step times in PyLO

We extend our analysis beyond the baseline results to examine per-step computational overhead across varying model scales representative of contemporary pretraining workloads. Table 4 presents a comprehensive timing breakdown across forward pass, backward pass, and optimizer step durations for both vision and language model architectures. Traditional optimizers including AdamW (Loshchilov & Hutter, 2019) and Adafactor (Shazeer & Stern, 2018) exhibit minimal computational overhead relative to the forward and backward passes. Our CUDA kernel implementations demonstrate substantial reductions in learned optimizer overhead, with improvements that scale favorably with model size. Notably, for ViT-L-16, our `small_fc_lopt`(CUDA) implementation achieves a 9× reduction in optimizer step time compared to the

Model	Optimizer	Batch Size	Samples per sec	Step time (ms)	Forward time (ms)	Backward time (ms)	Optimizer step time(ms)	Num parameters(M)
Vit-B-16	AdamW	32.00	524.09	60.55	17.52	38.13	4.90	86.57
	Adafactor		425.51	74.69			18.99	
	small_fc_lopt (naive)		39.36	812.51			756.80	
	`small_fc_lopt` (CUDA)		205.59	155.16			99.59	
Vit-S-16	AdamW	32.00	1,116.12	28.17	7.65	18.27	2.25	22.05
	Adafactor		711.97	44.47			18.53	
	`small_fc_lopt` (naive)		109.03	293.01			266.90	
	`small_fc_lopt` (CUDA)		382.73	83.28			57.45	
Vit-L-16	AdamW	32.00	175.95	181.36	52.18	113.89	15.28	304.33
	Adafactor		156.64	203.34			37.23	
	`small_fc_lopt` (naive)		11.64	2,748.52			2,582.30	
	`small_fc_lopt` (CUDA)		70.69	451.69			285.80	
GPT2-125 m	AdamW	4.00	17.80	224.72	73.20	144.24	7.29	125.26
	Adafactor		17.02	235.07			17.76	
	`small_fc_lopt` (naive)		3.38	1,182.33			964.89	
	`small_fc_lopt` (CUDA)		11.71	341.46			124.09	
GPT2-355 m	AdamW	4.00	6.56	609.83	197.41	392.29	20.12	355.92
	Adafactor		6.40	624.64			35.11	
	`small_fc_lopt` (naive)		1.16	3,461.96			2,872.17	
	`small_fc_lopt` (CUDA)		4.40	908.69			319.14	
GPT2-1B	AdamW	4.00	2.88	1,387.51	442.49	896.81	48.21	912.95
	Adafactor		2.87	1,395.03			54.08	
	`small_fc_lopt` (naive)		OOM	OOM			OOM	
	`small_fc_lopt` (CUDA)		1.98	2,025.26			681.78	

Table 4. Breakdown of time spent per training step, averaged over 40 steps. The CUDA-based implementation reduces overhead from the learned optimizer by minimizing memory usage and optimizer time, ultimately achieving competitive throughput compared to traditional optimizers.

naive implementation. This performance advantage becomes increasingly pronounced at scale. Consistent with our earlier findings regarding memory limitations—where JAX implementations encounter out-of-memory errors for 1B parameter language models—the naive learned optimizer implementation similarly fails at this scale. Our CUDA implementation addresses these scalability constraints while maintaining the computational efficiency gains observed with increasing batch sizes, making learned optimization viable for large-scale pretraining scenarios.

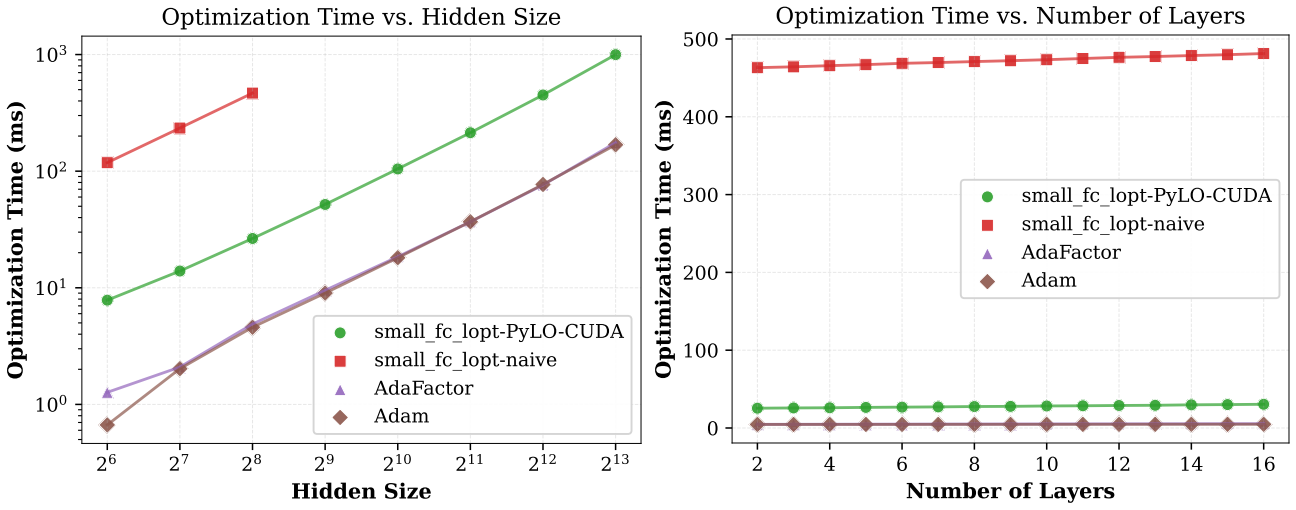
Scaling analysis with MLP: To systematically characterize the scaling behavior of our implementation, we conduct controlled experiments using synthetic MLP architectures with varying architectural parameters. Figure 5 presents two complementary scaling analyses that isolate the effects of network width and depth on optimizer computational overhead. Figure 5(a) examines the relationship between hidden layer dimensionality and optimization time for a fixed 5-layer MLP architecture. The results reveal a stark performance disparity: the naive implementation of `small_fc_lopt` exhibits optimization times an order of magnitude greater than our CUDA implementation and encounters memory limitations beyond hidden dimensions of 256. While our CUDA implementation remains slower than traditional optimizers (AdamW and Adafactor), it demonstrates favorable scaling characteristics that maintain computational tractability across the tested range.

Complementary analysis in Figure 5(b) evaluates depth scaling behavior using MLPs with fixed width of 256 units but varying layer counts (depth). The naive implementation consistently operates in a computationally prohibitive regime, requiring over 400ms for optimization steps, while our CUDA implementation maintains optimization times below 50ms—approaching the performance envelope of traditional optimizers despite the inherent complexity of learned optimization. These scaling experiments demonstrate that our CUDA implementation successfully addresses the computational bottlenecks that render naive learned optimizer implementation impractical for realistic model architectures.

E. Training Curves for the Experiments

This section presents a comprehensive empirical evaluation comparing learned optimizers VeLO (Metz et al., 2022b) and μLO (Thérien et al., 2024) against the standard AdamW optimizer (Loshchilov & Hutter, 2019; Kingma & Ba, 2017) with cosine learning rate scheduling (Loshchilov & Hutter, 2017). Our evaluation spans Vision Transformer architectures of varying scales and GPT-2 models, extending beyond the main paper's validation accuracy results for ViT-Base/16 and final loss metrics for GPT-2 to provide detailed training dynamics across multiple configurations.

Vision Transformer Experiments: We evaluate three ViT configurations—Small, Base, and Large with patch size

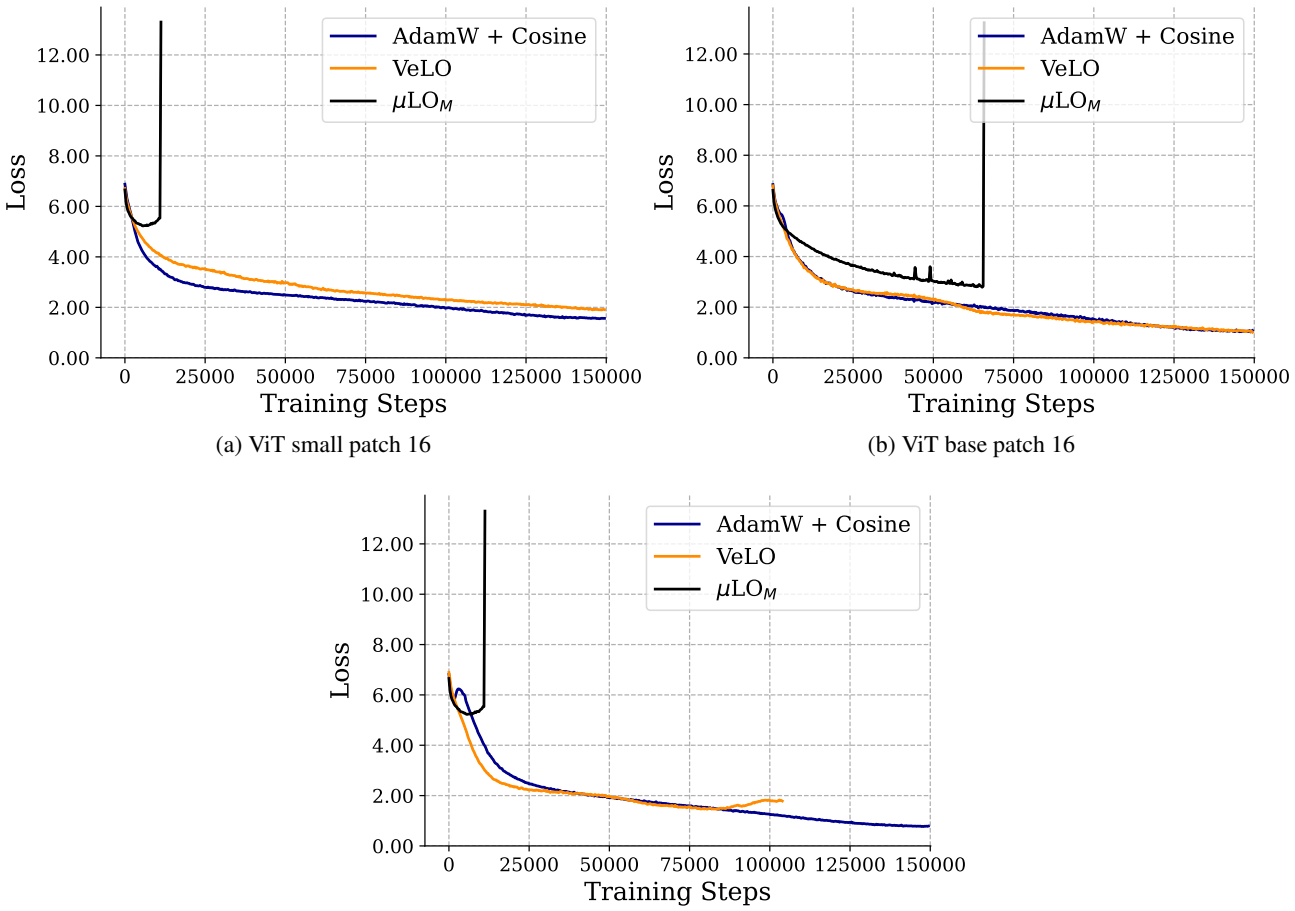


(a) Width (b) Depth

Figure 5. **Step time for deeper and wider MLPs.** Increasing the width and depth of MLPs provides a systematic way to measure the impact of larger and more tensors on optimizer step time. Subfigure (a) reports optimizer step time as the hidden size of the 5-layer optimizee MLP is increased, while Subfigure (b) reports step time as the depth of a 256 hidden-layer MLP is increased. We observe that in each case, the CUDA implementation reduces the optimizer step time by more than an order of magnitude.

16—on ImageNet-1k classification tasks. All configurations maintain consistent training protocols with 150,000 optimization steps and batch sizes of 4,096 across model variants, using hyperparameters detailed in Table 5. The extended evaluation presented in Figure 6 reveals critical scale-dependent optimization characteristics. MuLO demonstrates pronounced early divergence patterns across both small and large model configurations, reinforcing our hypothesis that its constrained meta-training horizon limits generalization to extended optimization trajectories. In contrast, VeLO exhibits remarkable consistency in convergence properties across the full spectrum of model scales examined, maintaining stable training dynamics while achieving competitive performance metrics relative to the AdamW baseline.

Language Modeling Experiments For GPT-2 pretraining on FineWeb-EDU data, we train models for 19,073 steps with batch size 512 and sequence length 1024. Complete hyperparameters are presented in Table 7, with full training curves shown in Figure 7. VeLO demonstrates competitive performance against AdamW with cosine scheduling, while MuLO exhibits significantly degraded performance across the extended training horizon, consistent with the stability patterns observed in vision tasks.

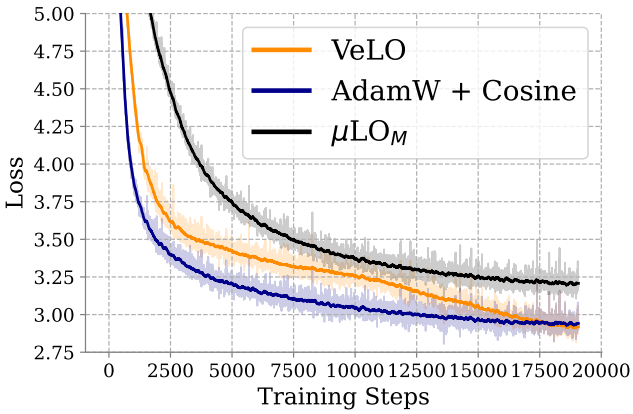


(a) ViT small patch 16

(b) ViT base patch 16

(c) ViT large patch 16

Figure 6. **Vision Transformer training dynamics.** Training loss curves for Vision Transformers trained on ImageNet-1k classification across three model configurations: (a) ViT-small with patch size 16, (b) ViT-base with patch size 16, and (c) ViT-large with patch size 16. We compare AdamW with cosine learning rate scheduling (blue), VeLO learned optimizer (orange), and μLO learned optimizer (black) over 150,000 training steps with batch size 4,096. μLO exhibits early training instability across all scales, while VeLO demonstrates competitive or superior convergence properties, particularly evident in the base model configuration where it achieves lower final loss than the AdamW baseline.



(a) 355M LM

Figure 7. **GPT-2 pretraining performance.** We pre-train decoder-only transformers of different sizes on a causal language modeling objective. We estimate gradients from 512 sequences of length 1024, resulting in a $\sim$0.5M token batch size per step. We train all models for 10B tokens of FineWeb-EDU data.

F. Extended Ablation Studies

This section presents comprehensive training curves across all hyperparameter configurations examined in our weight decay and cosine annealing ablations. Our analysis encompasses systematic sweeps of weight decay coefficients and maximum learning rates for cosine annealing schedules.

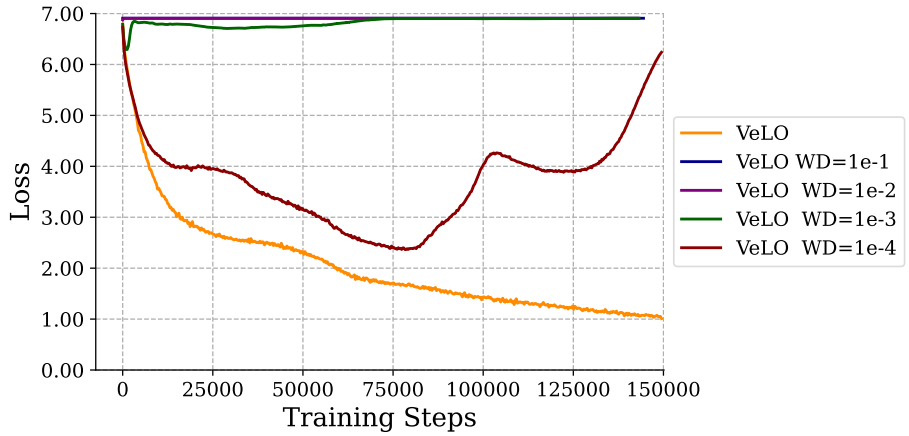
Vision Transformer Ablations Figures 8 and 9 demonstrate the variation of training dynamics with varying maximum learning rates and weight decay values for the ViT-B/16 model. The results reveal that VeLO exhibits minimal sensitivity to weight decay and scheduler modifications in this experimental setup, suggesting robust inherent regularization properties. Conversely, MuLO shows improved training horizon stability when augmented with cosine scheduling, enabling successful completion of the full 150,000 training steps. Weight decay modifications yield marginal improvements for MuLO across the evaluated parameter ranges.

Language Modeling Ablations

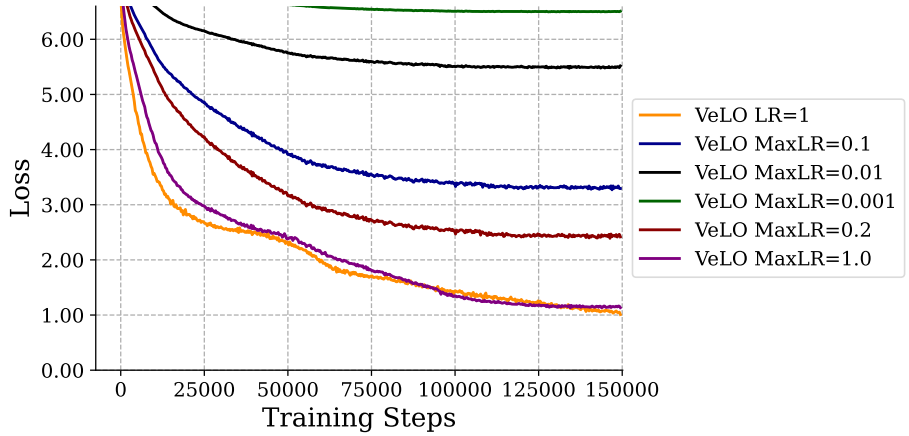
The language modeling experiments, presented in Figures 10 and 11, demonstrate more pronounced sensitivity to weight decay and scheduling compared to vision tasks. VeLO shows measurable performance improvements when augmented with weight decay and scheduling for specific parameter configurations. Similarly, MuLO benefits substantially from both weight decay regularization and cosine scheduling in the language modeling domain, suggesting that learned optimizers may benefit from scheduling and weight decay.

Architectural Configuration Effects

Figure 12 presents an additional ablation examining the impact of separate versus concatenated query-key-value (QKV) matrices on the optimizee's training loss. We observe that separating the QKV matrices in attention blocks improves performance across VeLO and μLO_M.

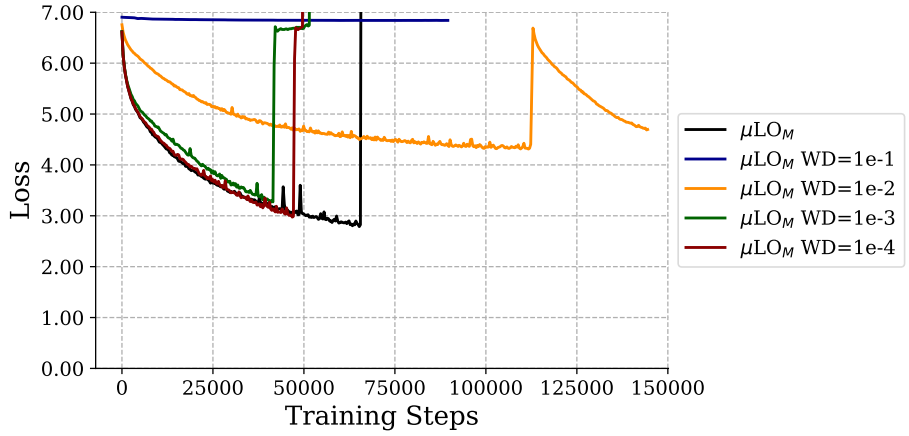


(a) VeLO Decoupled Weight decay ablation

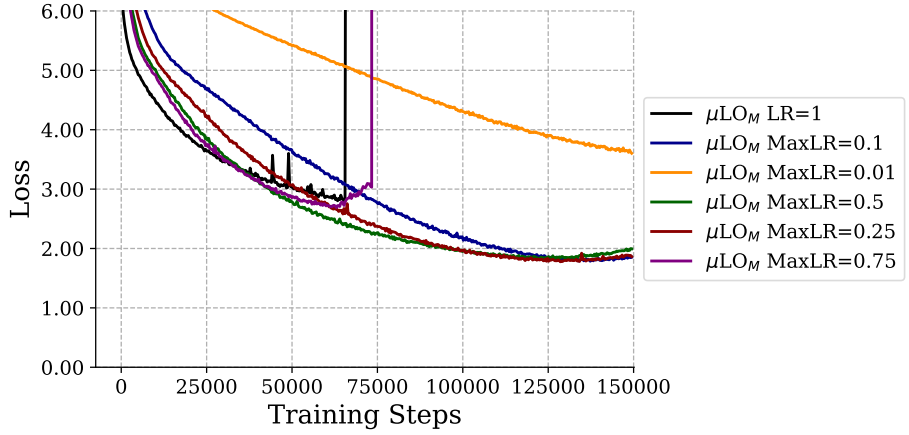


(b) VeLO Cosine annealing Ablation

Figure 8. **Training ViT-B-16 with VeLO on Imagenet 1k with decoupled weight decay and cosine annealing.** We see no improvement in using weight decay or Scheduler for VeLO in this setup

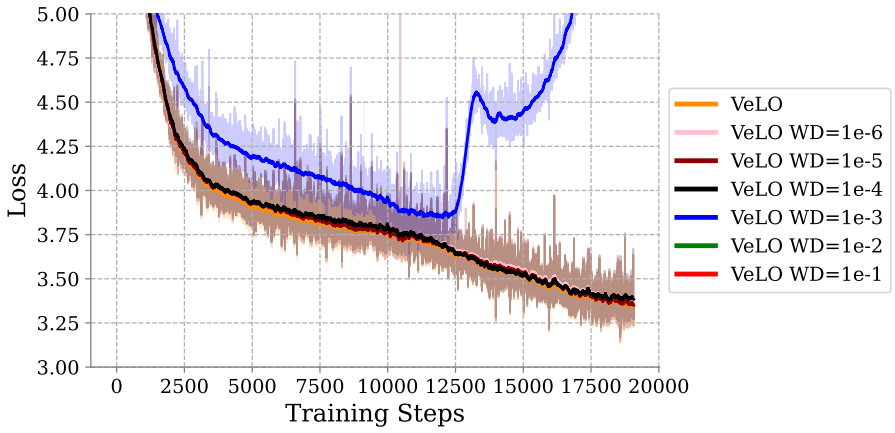


(a) μLO Decoupled Weight decay ablation

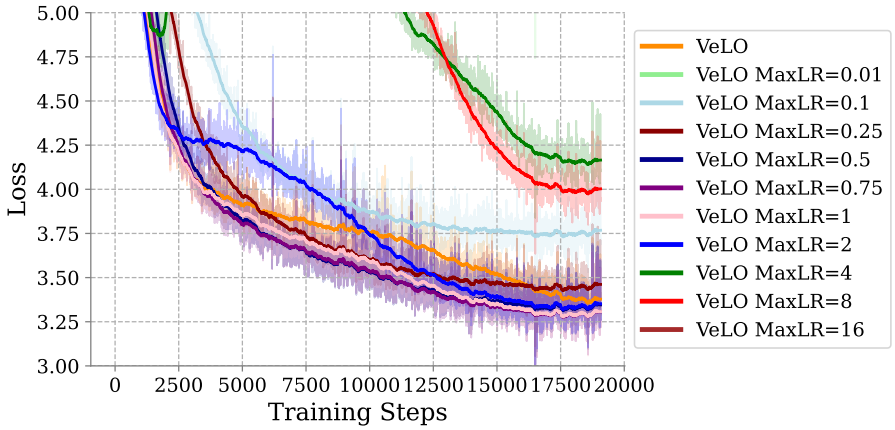


(b) μLO Cosine annealing Ablation

Figure 9. **Training ViT-B/16 with μLO on Imagenet1k with decoupled weight decay and cosine annealing.** We see that the training horizon of μLO is improved with using a scheduler that helps it to run up to 150000 training steps. We also see that weight decay did not have substantial improvement

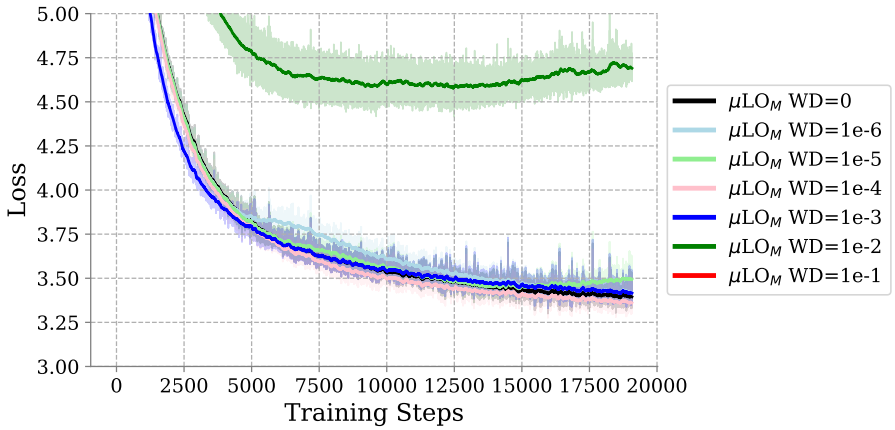


(a) VeLO Decoupled Weight decay ablation

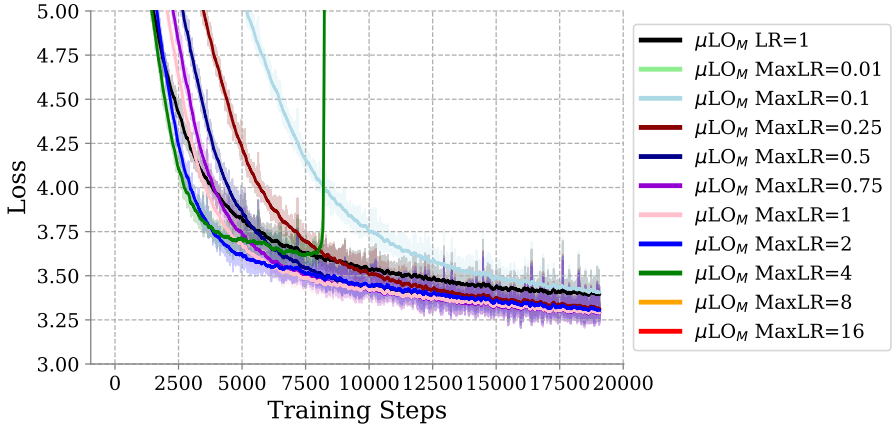


(b) VeLO Cosine annealing Ablation

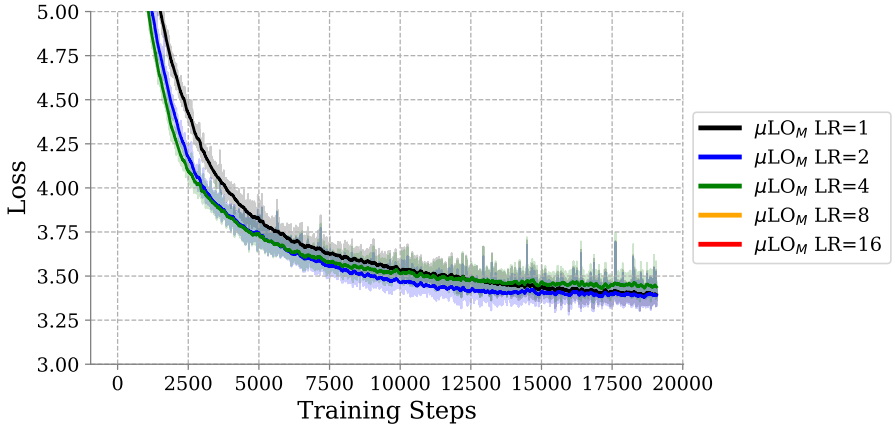
Figure 10. **GPT2 Pre-training VeLO ablations: decoupled weight decay and cosine annealing.** We pre-train decoder-only transformers of different sizes on a causal language modeling objective. We estimate gradients from 512 sequences of length 1024, resulting in a ~ 0.5 M token batch size per step. We train all models for $10B$ tokens of FineWeb-EDU data.



(a) μLO Decoupled Weight decay ablation

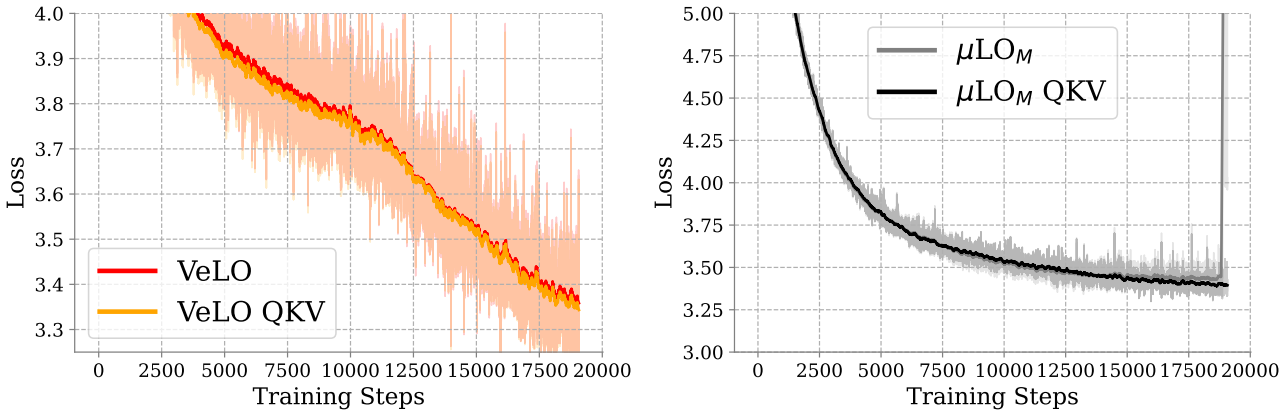


(b) μLO Cosine annealing Ablation



(c) μLO LR Tuning Ablation

Figure 11. **GPT2 Pre-training VeLO ablations: decoupled weight decay, LR-tuning, and cosine annealing.** We pre-train decoder-only transformers of different sizes on a causal language modeling objective. We estimate gradients from 512 sequences of length 1024, resulting in a ~ 0.5 M token batch size per step. We train all models for $10B$ tokens of FineWeb-EDU data.



(a) VeLO

(b) μLO

Figure 12. **Separate QKV weight ablation.** We ablate using concatenated or separate QKV matrices in attention layers. From the perspective of the learned optimizer, these two settings will be treated differently. We observe that performance improves for both μLO$_M$ and VeLO when QKV matrices are separated.

G. Hyperparameters for Experiments

Table 5. **Vision Transformer Training Hyperparameters.**

Description	Value
Model Architecture	
Model	ViT-Base/16
Image Size	224×224
Patch Size	16×16
Training Configuration	
Batch Size	4096
Training Epochs	480
Optimizer (for baseline)	AdamW
Learning Rate (η)	4×10^{-3}
LR Scheduler	Cosine
Warmup Epochs	32
Weight Decay	0.03
Gradient Clipping	1.0
Data Augmentation	
Crop Percentage	0.95
Random Horizontal Flip	0.5
Mixup	0.1
CutMix	1.0
AutoAugment	rand-m7-mstd0.5
Regularization	
Dropout Rate	0.1
Stochastic Depth	0.1

Table 6. **Vision Transformer Model Variants Architecture.**

Description	Value
ViT-Small/16	
Parameters	22.05M
Hidden Dimension (d_{model})	384
MLP Hidden Dimension	1536
Number of Heads	6
Number of Layers	12
Patch Embedding Dim	384
ViT-Base/16	
Parameters	86.57M
Hidden Dimension (d_{model})	768
MLP Hidden Dimension	3072
Number of Heads	12
Number of Layers	12
Patch Embedding Dim	768
ViT-Large/16	
Parameters	307.40M
Hidden Dimension (d_{model})	1024
MLP Hidden Dimension	4096
Number of Heads	16
Number of Layers	24
Patch Embedding Dim	1024

Table 7. **GPT-2 Training Hyperparameters.**

Description	Value
Model Configuration	
Base Model	GPT-2
Architecture Type	Decoder-only Transformer
Attention Mechanism	separate_kqv
Sequence Length (T)	1024
Vocabulary Size	50257
Training Configuration	
Batch Size	512
Training Iterations	19073
Optimizer (for baseline)	AdamW
Learning Rate Scheduler (for baseline)	Cosine
Warmup Iterations	381
Regularization	
Attention Dropout	0.1
Embedding Dropout	0.1
Residual Dropout	0.1
Initialization	
Muon Initialization	Enabled
Weight Initialization	Standard GPT-2

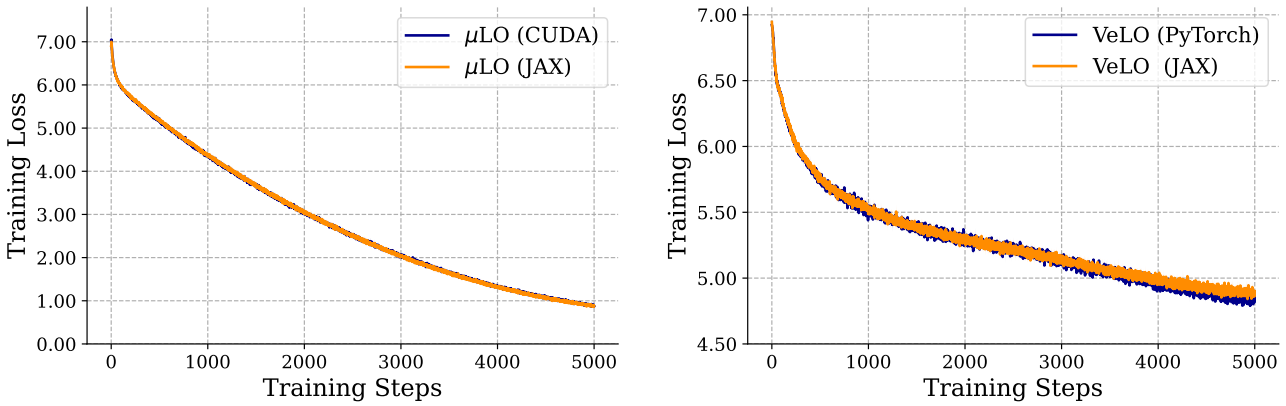
Table 8. **GPT-2 Model Variants Architecture.**

Description	Value
GPT-2 Small	
Parameters	$\sim$ 36M
Hidden Dimension (d_{model})	768
Number of Heads	12
Number of Layers	12
Head Dimension	64
FFN Hidden Dimension	3072
GPT-2 Medium	
Parameters	$\sim$ 345M
Hidden Dimension (d_{model})	1024
Number of Heads	16
Number of Layers	24
Head Dimension	64
FFN Hidden Dimension	4096
GPT-2 Large	
Parameters	$\sim$ 762M
Hidden Dimension (d_{model})	2048
Number of Heads	32
Number of Layers	16
Head Dimension	64
FFN Hidden Dimension	8192

H. Validation of PyLO Implementation Against Original JAX Codebase

To validate the correctness of our implementation of the popular learned optimizers in PyLO, we conducted a direct comparison with the original JAX implementation from Google's learned optimization(Metz et al., 2022a) codebase. We evaluated both μLO(Thérien et al., 2024) and VeLO(Metz et al., 2022b) algorithms on a simplified image classification benchmark using ImageNet resized to 64×64 pixels with a 3-layer MLP architecture (width 128). The comparison spans the first 5,000 training steps to assess implementation accuracy in a practical and economical setup.

The results demonstrate strong agreement between the two implementations across both optimizers. As shown in Figure 13, the training curves exhibit nearly identical convergence patterns across all 5 different runs. Minor variations arise from inherent differences in how PyTorch and JAX compute gradients. Accounting for this, we confirm that our PyTorch implementation faithfully reproduces the behavior of the original JAX code. We further plan to compare both implementations across a range of models and tasks.



(a) μLO implementation comparison

(b) VeLO implementation comparison

Figure 13. Training curve comparison between original JAX implementation and PyLO library for μLO and VeLO optimizers on ImageNet 64×64 classification task using a 3-layer MLP (width 128). Both implementations show nearly identical convergence behavior over 5,000 training steps, validating the correctness of our implementation.