

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 independently using the optimizers after meta-training. To address this gap, we introduce PyLO, a PyTorch-based library that brings learned optimizers to the remaining $\approx 70\%$ of machine learning community (Linux Foundation, 2024) via the familiar *torch.optim.Optimizer* interface. Unlike prior work focused on limited-scale academic tasks, our emphasis is on applying learned optimization to real-world large-scale pre-training tasks. Our systems contribution includes CUDA-accelerated implementations of the `small_fc_lopt`(Metz et al., 2022a) and VeLO(Metz et al., 2022b) learned optimizers, achieving substantial performance gains, with training throughput on ViT-B/16 (batch size 32) increasing from 39.36 and 49.73 to 205.59 and 191.18 samples per second, respectively. PyLO has the versatility that allows us to easily combine learned optimizers with existing optimization tools, such as learning rate schedules and weight decay. When doing so, we discover that learned optimizers can substantially benefit from it. 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-explored 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.

Several critical barriers have hindered the more widespread adoption of learned optimizers for deep learning applications: 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, and 4) 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 facilitate standardized sharing of task-specific learned optimizer weights (Figure 2).
2. We provide a CUDA-accelerated implementation of `small_fc_lopt` and VeLO'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

[*]Equal contribution [1]Concordia University [2]Mila Quebec AI Institute [3]Université de Montréal [4]Eluther AI. Correspondence to: Paul Janson <paul.janson@mila.quebec>, Benjamin Thérien <benjamin.therien@mila.quebec>.

Proceedings of the 9^{th} MLSys Conference, Bellevue, WA, USA, 2026.

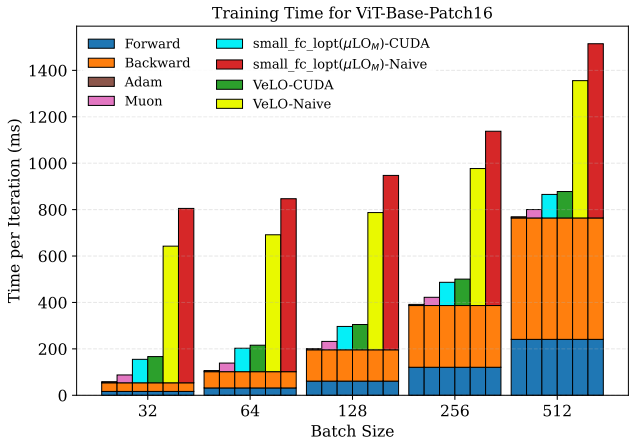


Figure 1. **Training step timing breakdown for ViT-B/16 on a single A100 GPU.** We measure the forward, backward, and optimizer times per training step. We observe that the CUDA-accelerated learned optimizer steps (cyan, green) show substantial improvements over the naive implementations (yellow, red). In all cases, as the batch size is increased, the relative overhead of the optimizer shrinks.

modeling workloads, showing that our implementation scales to larger workloads and decreases step times by more than 2× over the existing JAX implementation.

4. We demonstrate that learned optimizer step times can further be reduced in a data-parallel training setting by distributing the optimizer step across devices.
5. 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 Challenges of L2O adoption for the Wider Deep Learning Community

This section reviews existing open-source learned optimization repositories and reflects on the critical barriers hindering the widespread adoption of learned optimizers within the broader deep learning community, highlighting pitfalls in existing work.

2.1 Existing Open source libraries

Open-L2O (Chen et al., 2022) offers a comprehensive benchmarking suite designed primarily for evaluating learned optimizers (L2O) across a range of different optimization problems: convex functions, non-convex functions, minimax functions, and some neural network training. It includes both model-based and model-free L2O methods, facilitating reproducibility and fair comparisons, but only includes a limited evaluation of the learned optimizer for practical deep learning tasks. In contrast, PyLO's main focus is on learned optimization for deep learning.

Google's learned_optimization (Metz et al., 2022a) is a research-oriented repository written in JAX for training, designing, evaluating, and applying learned optimizers. It supports an extensive set of features for meta-training optimizers: many gradient estimation algorithms, abstractions for different types of tasks, various task modifiers, etc. While the repository also provides functionality for applying pre-trained optimizers to new tasks via the Optax interface (DeepMind et al., 2020), its main focus is clearly to provide meta-training functionality.

2.2 Challenges to adoptions

Overfocus on meta-learning: Existing libraries, such as the Learned Optimization (Metz et al., 2022a;b) and Open L2O (Chen et al., 2022), prioritize meta-training functionality over precisely focusing on the practical deployment of learned optimizers. This leads to a large proportion of the code being unnecessary for applying learned optimizers to new tasks. This can represent a barrier to entry, particularly for researchers and practitioners who seek to leverage pre-trained optimization algorithms without engaging in the complexities of meta-training (similar to using a pretrained transformer from `huggingface/transformers(Wolf et al., 2020)`). This highlights the necessity of decoupling learned optimizer implementations from their meta-training and meta-testing code.

PyTorch v.s. JAX: A third barrier relates to the underlying deep learning framework. While JAX (Bradbury et al., 2018) offers powerful functional programming capabilities and strong performance from compilation, it lacks the widespread community adoption of PyTorch (Paszke et al., 2019). Beyond having more users, widespread community adoption also comes with a more mature and feature-full open source ecosystem (He, 2019; Wightman, 2019; von Platen et al., 2022; Wolf et al., 2020). With so much demand for compatibility with PyTorch, there exists a pressing need for a modular L2O framework that seamlessly integrates with PyTorch's optimizer interface. Currently, Google's Learned Optimization does not provide any compatibility with PyTorch. While Open-L2O does support PyTorch, it lacks many other crucial aspects for adoption (see Table 1).

Sharing learned optimizer weights: Being able to effectively share optimizer weights and document their training distribution and evaluation performance is essential for building an effective open source ecosystem around learned optimizers. However, no such mechanism currently exists. In contrast, pre-trained models for language understanding (DeepSeek-AI et al., 2025; Touvron et al., 2023), image

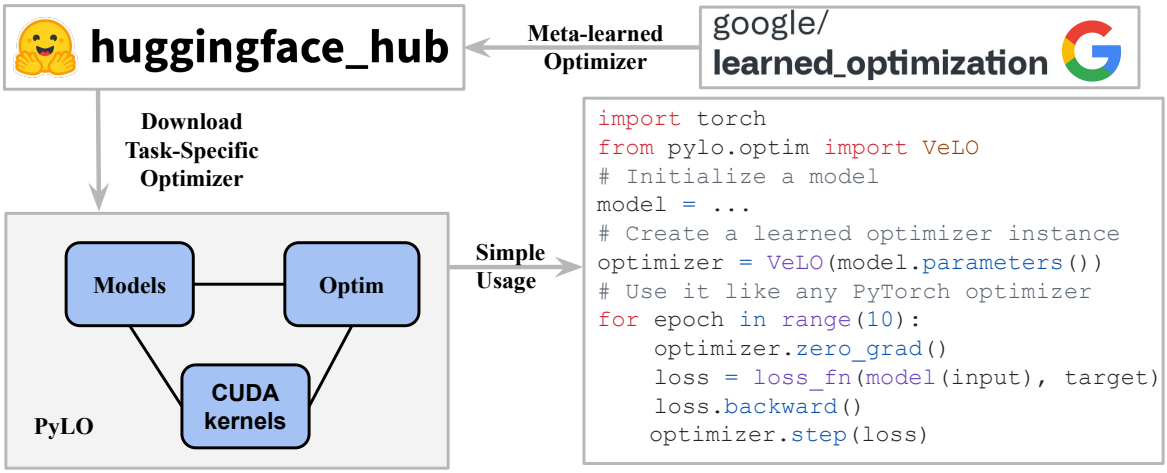


Figure 2. **PyLO:** simplifies the integration of learned optimizers into standard machine learning workflows. By addressing key usability challenges, PyLO provides seamless access to meta-learned optimization techniques through three core features: (1) automatic weight loading from Hugging Face Hub, (2) a familiar PyTorch-style optimizer interface, and (3) accelerated CUDA kernel support. The library bridges the gap between advanced meta-learning research (`google/learned_optimization`(Metz et al., 2022a)) and practical machine-learning applications, enabling researchers and practitioners to easily leverage state-of-the-art learned optimization techniques in PyTorch.

Table 1. **Comparison of Open-source Learned Optimization Repositories.** We list the current open-source repositories for learned optimization and list their status relative to the difficulties of L2O adoption from section 2.2. A Decoupled framework separates learned optimizer implementation from meta-training and meta-testing code.

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

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 believe that Learned Optimizer (LO) models can and should be integrated into the community through the HuggingFace ecosystem. Current open-source L2O frameworks, Learned Optimization and OpenL2O, do not provide any mechanism for sharing learned optimizer weights.

Per-step overhead of learned optimizers: A significant obstacle to the widespread adoption of learned optimizers is the computational overhead they can introduce to already resource-intensive training setups. Deep learning workflows operate under strict time and computational constraints, with practitioners demonstrating marked reluctance to trying 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, which can be achieved by writing custom CUDA kernels.

3 Project Goal

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, PyLO ensures seamless integration with existing training frameworks and compatibility with popular optimizer modifiers. This allows practitioners to use learned optimizers with minimal modification to their existing code—including modifiers like decoupled weight decay and learning rate schedules.

4 LIBRARY DESIGN

We implement a modular architecture partitioned into four 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. It also composes well with torch.compile and activation checkpointing. 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. We emphasize 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 or billions of parameters.

Drawing inspiration from architecture-aware optimizations (Müller et al., 2021; Dao et al., 2022), we implemented specialized CUDA kernels for applying learned optimizers. 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) **Fused MLP Inference** and (2) **On-demand 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. The full implementation and performance analysis are detailed in Section 5.

Decoupled evaluation Providing easy-to-use examples for new open-source code can go a long way for improving adoption, as it illustrates how the library can be used and provides a starting point for exploration. However, it is important to decouple such examples from the learned optimizer implementation itself to keep the lines of code within PyLO to a minimum. As such, we provide PyLO-Examples alongside PyLO as a supporting repository for evaluating learned optimizers on popular language modelling (pre-training on FineWeb EDU) and image classification (classical ImageNet pre-training) tasks. Each task is implemented within a single directory to maximize readability and re-use.

5 CUDA IMPLEMENTATION

In this section, we describe the implementation details of our CUDA kernels and the motivation behind our design choices. We begin by analyzing the performance bottlenecks in the naive PyTorch implementation, then present our optimized approach.

5.1 Analysis of Naive Implementation

A learned optimizer step comprises several distinct operations, visualized in Figure 3. Consider a parameter tensor of dimensions $m \times n$. The naive implementation constructs a feature vector of size $mn \times d_{\text{feat}}$ where d_{feat} denotes the learned optimizer's input dimensionality. This construction starts after the accumulator update (update of momentum, second moment, etc) and proceeds in three stages:

Feature Construction After updating accumulators the `construct_features()` procedure starts. The gradient g, parameter p, momentum buffer m, second-moment estimate v, and factored second-moment terms (row and column factors), are used to compute a set of derived features (see Appendix H for all features). Each derived feature

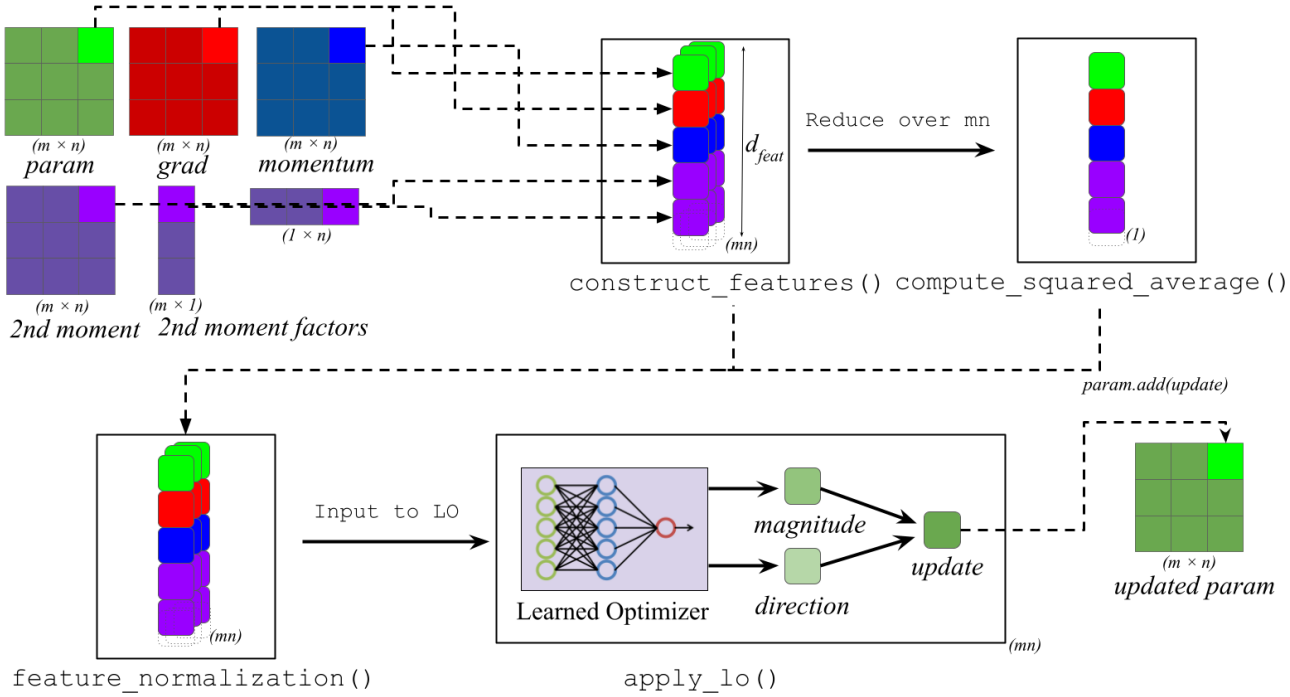


Figure 3. **Overview of the Learned Optimizer (LO) update mechanism:** The figure depicts the computation pipeline used by learned optimizers such as `small_fc_lopt` and VeLO. Model parameters (θ), gradients ($\mathbf{g}$), momentum (m), and second-moment accumulators (v, row & column factors of v) are combined within `construct_features()` to form a rich set of derivative features, including elementwise interactions such as $\mathbf{g} \times$ momentum and momentum $\times$ factors (see Appendix H for a complete description). These features are reduced over the parameter dimensions ($m \times n$) in `compute_squared_average()` to obtain normalization statistics. In `feature_normalization()`, features are normalized before being processed by the Learned Optimizer, which predicts an update direction and magnitude in `apply_lo()`, and this predicted update is then applied to yield the updated parameters.

requires individual kernel launches for element-wise operations.

Feature normalization Before applying the learned optimizer's neural network, features must be normalized by their squared average across all parameter elements. This requires: (1) computing $\mathbb{E}[\text{feature}^2]$ via reduction operations (`compute_squared_average()`), and (2) dividing each feature by $\sqrt{\mathbb{E}[\text{feature}^2] + \epsilon}$ (`feature_normalization()`). PyTorch's general-purpose reduction kernels require additional kernel launches after materializing the full feature vector.

Apply Learned Optimizer Following feature preparation and normalization, the `apply_lo()` procedure is invoked. A learned optimizer, which is implemented as a neural network, takes the normalized feature vector as input and predicts two scalar outputs: one representing the *magnitude* and the other the *direction* of the parameter update. These predictions are combined to compute the update according to the formula

$$\Delta\theta = \text{direction} \times e^{\text{magnitude} \times \alpha} \times \beta,$$

where α and β are hyperparameters fixed at 0.01. The parameter θ is then updated as $\theta \leftarrow \theta + \Delta\theta$. In the PyTorch-based MLP implementation, this process launches a separate kernel for each layer and stores intermediate activations in global memory. Consequently, memory bandwidth emerges as the primary bottleneck, resulting in underutilized GPU compute resources and reduced overall efficiency.

The naive implementation thus requires 74-252 kernel launches per parameter tensor, with each intermediate result materialized in global memory. For a parameter tensor of size mn, this generates approximately $(num\_of\_accum) * mn$ words of memory traffic and allocates $mn \times d_{\text{feat}}$ words of temporary storage, which a substantial burden for large models. This is evidenced by the profiler visualization as shown in Figure 4.

5.2 Optimized Kernel Design

Our optimization strategy rests on two key observations: (1) feature construction can be performed in a single pass over the parameter tensor, and (2) the neural network forward pass can be fused with feature construction to eliminate intermediate storage. To this end, we implement a two-pass method in which we decompose the learned optimizer step

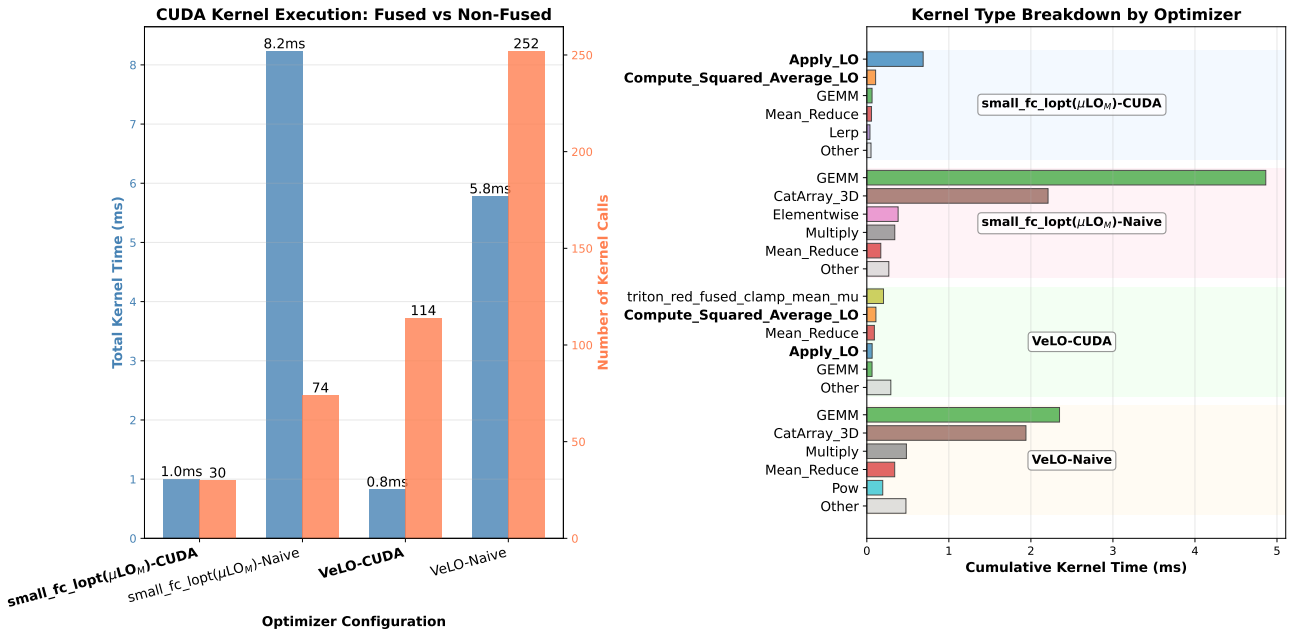


Figure 4. Comparison of CUDA kernel execution characteristics for the learned optimizers `small_fc_lopt` and VeLO, shown for both naïve and fused implementations applied to a single layer MLP (shape [1000x1000]) with no bias. Note that kernel fusion here refers exclusively to fusing the learned optimizer's feature construction, normalization, and MLP inference — the optimizee's forward and backward passes remain unchanged. (Left) The total cumulative kernel execution time (blue) and the number of kernel launches (orange) reveal that the naïve implementations spend substantially more time executing a far greater number of small kernels, due to performing each arithmetic or reduction step as an independent operation. In contrast, the fused CUDA versions combine these operations into fewer, more computationally dense kernels, reducing both launch overhead and memory bandwidth pressure. (Right) Breakdown of cumulative kernel time by kernel type for each optimizer shows that the fused versions **(bolded)** consolidate many repetitive operations into specialized fused kernels. This pattern is consistent across both VeLO and `small_fc_lopt`, demonstrating that kernel fusion is broadly beneficial for improving learned optimizer efficiency. Descriptions about the kernel names provided in Appendix J

into two kernels:

Kernel 1: Feature statistics collection. We perform a single pass over all parameters, computing features on-the-fly in registers and accumulating their squared values for normalization. Rather than materializing the full feature tensor, we maintain only a d_{feat}-dimensional accumulator per thread. Reduction proceeds hierarchically: thread-local accumulators are reduced within each warp using shuffle instructions, warp results are aggregated via shared memory, and block-level results are combined through atomic operations to global memory. This approach computes the required statistics using only $O(d_{\text{feat}})$ shared memory per block and $O(d_{\text{feat}})$ global memory total, independent of parameter count. This combines the `construct_features()` and `compute_squared_average()` procedures.

Pseudocode for Kernel 1

```
// Phase 1: Fused Compute Statistics
Kernel_1(g, p, m, v, ...) {
    thread_accum[d_feat] = {0};  // Register array
    for (i in grid_stride_loop) {
        features = compute_features(g[i], p[i], m[i], v[i], ...);
        thread_accum += features * features;
    }
    warp_reduce(thread_accum);  // Shuffle operations
    block_reduce(warp_results);  // Shared memory
    atomic_add(global_stats, block_result);
}
```

Kernel 2: Second Fused feature construction and network application. With normalized statistics available in global memory, we perform a second pass that: (1) loads the normalization factors into shared memory for efficient broadcast, (2) constructs features in registers from parameter and accumulator values for the second time, (3) evaluates the learned optimizer network in-line, and (4) applies the computed update. No intermediate tensors are materialized—all computation flows through the register file. This combines `construct_features()`,`feature_normalization()` and `apply_lo()` procedures.

Memory hierarchy exploitation Our design carefully manages data placement across the GPU memory hierarchy:

Registers: Feature vectors ($d_{\text{feat}} \approx 39$ elements for `small_fc_lopt` and $d_{\text{feat}} \approx 30$ elements for VeLO) and network activations reside entirely in registers during computation, providing single-cycle access latency. Our kernels are optimized for the current feature dimensions, but the feature dimension can be changed by tuning kernel launch parameters, with advantages maintained up to the SRAM capacity per streaming multiprocessor. We note that these feature sets have been heavily ablated in prior work (Metz et al., 2022a), with additional features showing diminishing returns.

Shared memory: The d_{feat}-dimensional normalization statistics are loaded once per thread block and broadcast to all threads, amortizing the cost of global memory access across all threads in the block.

Global memory: Only parameter values, gradients, and optimizer state buffers are read from global memory, and only updated parameters are written back. This reduces memory traffic substantially per optimization step.

This design eliminates kernel launch overhead (reducing from 74-252 to 30-114 kernel invocations, see Figure 4), removes intermediate allocations (reducing memory footprint by $mn \times d_{\text{feat}}$ words), and minimizes memory bandwidth consumption (Data movement is minimized as only accumulators are read and final outputs are written). The result is an implementation that achieves good memory bandwidth utilization while maintaining numerical equivalence to the naive approach.

Pseudocode for Kernel 2

```
// Phase 2: Fused Apply
Kernel_2(g, p, m, v, ..., global_stats) {
    __shared__ normalized_stats[d_feat];
    if (tid < d_feat) {
        normalized_stats[tid] = rsqrt(
            global_stats[tid] / N); // N =
            number of params
    }
    __syncthreads();

    for (i in grid_stride_loop) {
        // All computation in registers
        features[d_feat] = compute_features
            (g[i], p[i], m[i], v[i], ...);
        features *= normalized_stats;  //
            Broadcast from SMEM

        // Inline MLP (weights in __ldg
            cache)
        hidden = relu(Linear1(features));
        hidden = relu(Linear2(hidden));
        direction, magnitude = Linear3(
            hidden);

        update = direction * exp(magnitude
            * alpha) * beta;
        p[i] -= update;
    }
}
```

6 Benchmarking LO step times in PyLO

In this section, we establish the per-step overhead of learned optimizers for common pre-training workloads and showcase 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 vision transformers and transformer language models of different sizes 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 relative to the forward and backward passes diminishes. This indicates that large-scale training with prolonged per-step compute may be particularly well-suited for learned optimizers, as the optimizer cost can be effectively amortized. This trend is further evidenced in Table 2 for both vision and language modeling, which presents timing comparisons across optimizers for ViT-B/16 and a medium-sized GPT-2 model (355M parameters). We find that relative throughput compared to Adam improves with model scale, holding true for language modeling as well. Additionally, per-step time for `small_fc_lopt` (CUDA) scales with model size, yet our CUDA kernels deliver substantial speedups over the naive baseline. For ViT-B/16, `small_fc_lopt` (CUDA) achieves an 86% reduction in optimizer step time versus the naive version; for the GPT-2 task, the reduction is 88%. (Full results, including larger ViT and GPT variants, are provided in Appendix G.)

Scaling behavior of our CUDA implementation vs the JAX baseline Figure 5 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)) and VeLO(Metz et al., 2022b). 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 implementations results in a substantial memory savings as it can optimize a 1B parameter transformer, while the JAX implementation runs out of memory for an 80 GB A100 GPU. This is because it materializes the full feature tensor of size $mn \times d_{feat}$ in global memory, which exceeds GPU memory at the 1B

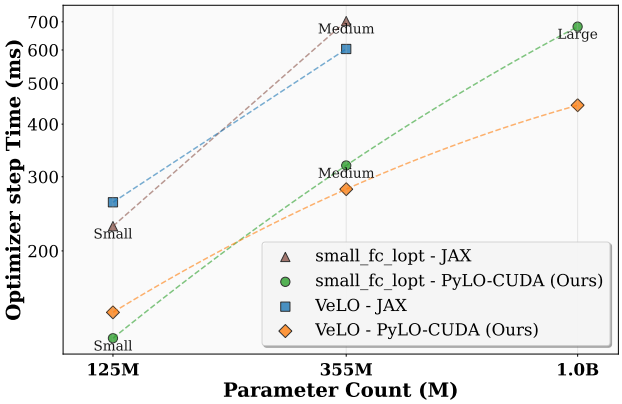


Figure 5. **Step Time Scaling of Learned Optimizers.** We present a comparison of optimizer step time between our custom CUDA implementation and the original JAX versions of `small_fc_lopt` and VeLO, evaluated during the training of GPT-2 style transformer models across a range of model sizes. The results 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.

Model	Optimizer	BS/SL	Fwd (ms)	Bwd (ms)	Opt step (ms)	#Params (M)
ViT-B/16	Adam(Loshchilov & Hutter, 2019)	32/197	17.52	38.13	4.90	86.57
	Adafactor(Shazeer & Stern, 2018)				18.99	
	`small_fc_lopt` (naive)				756.80	
	`small_fc_lopt` (CUDA)				99.59 (-86%)	
	VeLO (naive)				585.11	
	VeLO (CUDA)				113.58 (-80%)	
GPT2-355M	Adam(Loshchilov & Hutter, 2019)	4/1024	197.41	392.29	20.12	355.92
	Adafactor(Shazeer & Stern, 2018)				35.11	
	`small_fc_lopt` (naive)				2872.17	
	`small_fc_lopt` (CUDA)				319.14 (-88%)	
	VeLO (naive)				2378.93	
	VeLO (CUDA)				284.37 (-88%)	

Table 2. **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 (reduction shown in red) compared to naive implementation, enabling more efficient training in practical settings.

scale, whereas the CUDA kernels avoid this intermediate allocation entirely. Moreover, in cases where the JAX implementation does not run out of memory, it is outperformed by our CUDA implementation, resulting in a 2× reduction in step time.

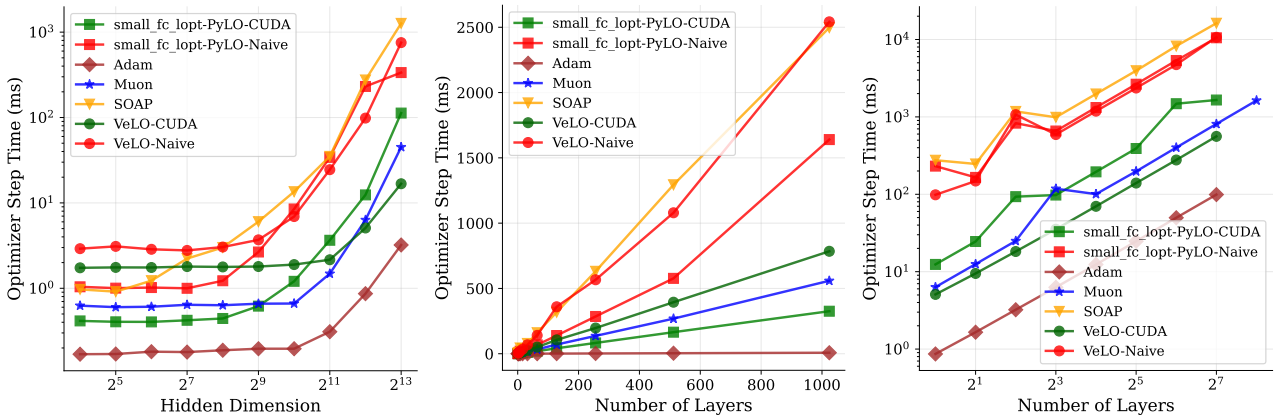
Isolating Scaling behavior of our CUDA implementation with MLP taks To systematically characterize the scaling behavior of our implementation, we conduct controlled experiments using synthetic MLP architectures with varying numbers of layers and hidden dimensions. This allows us to explicitly control for tensor sizes and the number of tensors to measure the effect on our implementation. Figure 6 presents three complementary scaling analyses that isolate the effects of network width and depth on optimizer computational overhead. Figure 6(a) examines the relationship between hidden layer dimensionality and optimization time for a fixed 1-layer MLP architecture. The results reveal a stark performance disparity: the naive implementations of learned optimizers exhibit optimization times an order of magnitude greater than our CUDA implementation. While our CUDA implementations remains slower than traditional optimizers, it demonstrates favorable scaling characteristics. We further enhance this on H100 hardware in Appendix B.

Complementary analysis in Figure 6(b),(c) evaluates depth scaling behavior using MLPs with fixed width of 256 and 4096 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 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 implementations impractical for realistic model architectures.

7 Distributed Optimizer Step

It is possible to further reduce optimizer step time by distributing computation across multiple devices. So far, our analysis has focused on optimizer step times for fully replicated data-parallel training where the gradient is first all-reduced and, subsequently, the optimizer step for the entire model is performed *redundantly* and simultaneously on all devices. While not ideal, this approach is reasonable for coordinate-wise optimizers like Adam, which require little computation during the optimizer step. However, optimizers like Shampoo (Gupta et al., 2018), Muon (Jordan et al., 2024), or Learned optimizers (Metz et al., 2022a;b), that perform significant computation at each step, can benefit from distributing the optimizer step across multiple devices. As illustrated in figure 8, the optimizer step can effectively be distributed across devices by splitting the all-reduce operation into its parts: (1) reduce scatter the gradients evenly for each parameter tensor across devices, (2) perform the optimizer step *non-redundantly* across all devices, and (3) all-gather the updated model parameters.

Improvements from using a distributed optimizer step Table 3 reports the time taken for the forward and backward pass, communication, and optimizer step for different optimizers when training a 125M transformer language model across four H100 GPUs. We observe that distributing the optimizer step can lead to substantial improvements for all optimizers, but that learned optimizers improve the most from a distributed step: by 31.86 ms for small_fc_lopt and 18.96 ms for VeLO. This reduces the overhead relative to Adam for these optimizers to just 9% and 13%, respectively.



(a) 1-layer MLP (b) 256-width MLP (c) 4096-width MLP

Figure 6. **Scaling behavior of CUDA implementation with varying MLP configurations** We evaluate optimizer step time across various MLPs with controlled width and depth. Subfigure (a) reports optimizer step time as the hidden size of the 1-layer optimizee MLP is increased. Subfigure (b) reports step time as the depth of a 256 hidden-dimension optimizee MLP is increased. Subfigure (c) reports step time as the depth of a 4096 hidden-dimension optimizee is increased. Missing datapoints correspond to OOM errors. We observe that in each case, our CUDA implementation reduces the optimizer step time by an order of magnitude relative to the naive implementation.

Table 3. **Comparing improvements from distributed optimizer steps.** We report the time taken for the forward and backward pass, communication, and optimizer step for different optimizers when training a 125M transformer language model across four H100 GPUs. All times are reported in milliseconds (ms). The all-reduce column performs an all-reduce of the gradient before the optimizer step, while reduce-scatter distributes the optimizer step across devices (Figure 8 illustrates the difference between these steps.). The FSDP A2A column uses ZeRO-1 style sharding of optimizer states with all-to-all communication to distribute tensors across devices for the optimizer step. Difference (Δ) columns show improvement relative to the all-reduce baseline.

Optimizer	All-reduce	Reduce-scatter	Δ	FSDP A2A	Δ
small_fc_lopt CUDA	136.53	104.67	−31.86	110.53	−26.00
VeLO CUDA	127.67	108.71	−18.96	116.00	−11.67
Muon	106.92	98.10	−8.82	100.92	−6.00
Adam	99.93	95.93	−4.00	96.46	−3.47

These results suggest that, in large batch training settings, the overhead or learned optimizers relative to Adam and Muon can further be reduced by distributing the optimizer step over more devices.

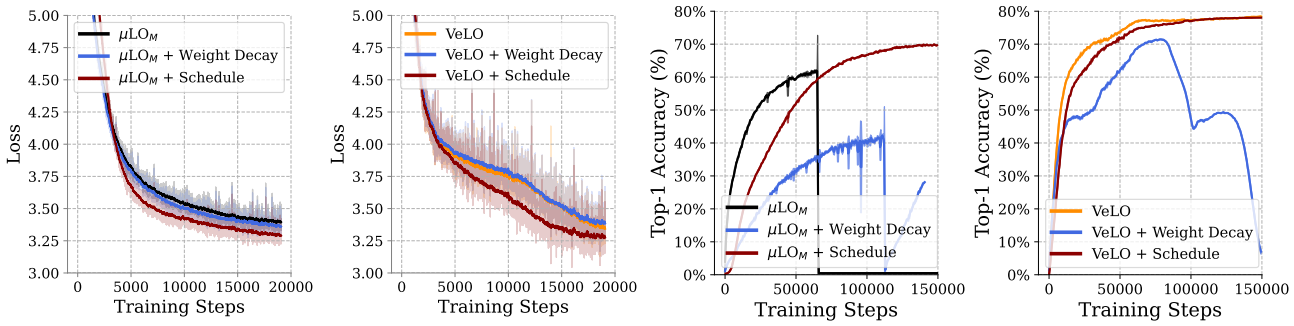
Sharded Optimizer States. Our analysis shows that tensor-level reduce-scatter significantly accelerates distributed optimizer steps. Further gains in memory and compute are possible by sharding optimizer states across devices, as in ZeRO (Rajbhandari et al., 2020) (i.e., fully sharded data parallelism (Zhao et al., 2023)). To evaluate this, we implement a ZeRO-1 style optimizer step that shards only optimizer states across devices and uses an all-to-all operation to distribute tensors for the optimizer step. The FSDP A2A column in Table 3 reports these results. We find that the FSDP A2A approach provides substantial improvements over the all-reduce baseline for all optimizers, reducing step time by 26.00 ms for small_fc_lopt and 11.67 ms for VeLO. While the reduce-scatter approach remains faster overall, the FSDP A2A approach offers additional memory savings by sharding optimizer states across devices. Since learned optimizers currently operate per-parameter (except normalization), the costly all-to-all could potentially be replaced with a distributed normalization step that communicates only d_{feat} scalars. We leave this optimization for future work.

8 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 `small_fc_lopt` to the performance of tuned Adam with a cosine annealing schedule. We use μLO_M (Thérien et al., 2024) pretrained weights for the `small_fc_lopt`. Our goal is to establish the performance of readily available learned optimizers in PyLO for these practical vision and language pre-training tasks.

8.1 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.



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

Figure 7. **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.

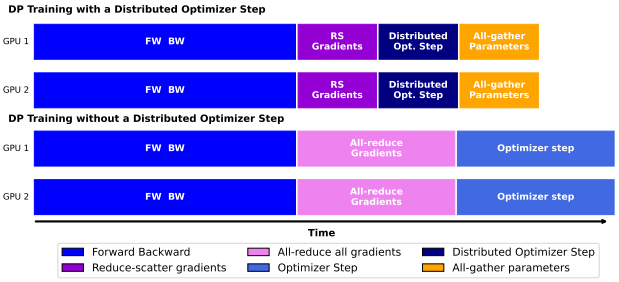


Figure 8. **Overlapping Optimizer steps with Communication and distributed Optimizer steps.**

Results As shown in Table 4 VeLO demonstrates competitive performance in this context, achieving 78.45% top-1 accuracy compared to Adam's 77.22%. This scenario demonstrates a case where the learned optimizer practically competes with Adam. 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.

Table 4. 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**
Adam + Cosine	2.91	77.22

8.2 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, employing a batch size of 512 and sequences of length 1024 (see section E for additional hyperparameter details).r more details.

Results

Table 4 reports the final loss achieved for all optimizers in our study, while figure 11 of the appendix plots the training curves. We observe that VeLO outperforms Adam and μLO$_M$, reaching the lowest final loss. This demonstrates that PyLO has the potential to deliver optimization performance improvements to PyTorch users.

8.3 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 7 reports ViT and GPT training and accuracy curves for the best-performing MaxLR while training curves for more values are reported in the appendix (D). 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 7 reports ViT and GPT training and accuracy curves for the best-performing λ while training curves for more values are reported in the appendix (D). We observe that μLO$_M$ benefits from weight decay for training GPT but not VeLO. Neither optimizer benefits from weight decay for ViT tasks.

9 CONCLUSION

We have presented PyLO, an open-source PyTorch library that removes key barriers to the practical use of learned optimizers in large-scale deep learning. PyLO offers drop-in implementations of state-of-the-art methods such as VeLO and `small_fc_lopt` through the standard `torch.optim.Optimizer` interface, with seamless Hugging Face Hub integration for model sharing. Central to our systems contributions are custom CUDA kernels that fuse feature construction, normalization, and MLP inference, and a distributed implementation that distributes tensors across multiple devices. This achieves 86–88% lower overhead than naive PyTorch, and can efficiently scale to training a 1B-parameter transformer on a single A100 GPU. Our evaluation shows that PyLO reduces learned optimizer overhead to acceptable levels at scale and they can deliver competitive accuracy on ImageNet and large-scale language pretraining. As future work, we envision PyLO enabling (1) hardware-optimizer co-design (2) behavior-cloned distillation of expensive second-order optimizers such as SOAP(Vyas et al., 2025). PyLO thus establishes a robust foundation for community-driven research and practical deployment of learned optimizers in modern ML systems.

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.

DeepMind, Babuschkin, I., Baumli, K., Bell, A., Bhupatiraju, S., Bruce, J., Buchlovsky, P., Budden, D., Cai, T., Clark, A., Danihelka, I., Dedieu, A., Fantacci, C., Godwin, J., Jones, C., Hemsley, R., Hennigan, T., Hessel, M., Hou, S., Kapturowski, S., Keck, T., Kemaev, I., King, M., Kunesch, M., Martens, L., Merzic, H., Mikulik, V., Norman, T., Papamakarios, G., Quan, J., Ring, R., Ruiz, F., Sanchez, A., Sartran, L., Schneider, R., Sezener, E., Spencer, S., Srinivasan, S., Stanojević, M., Stokowiec, W., Wang, L., Zhou, G., and Viola, F. The DeepMind JAX Ecosystem, 2020. URL http://github.com/google-deepmind.

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.

Gupta, V., Koren, T., and Singer, Y. Shampoo: Preconditioned stochastic tensor optimization. In *International Conference on Machine Learning*, pp. 1842–1850. PMLR, 2018.

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.

Jordan, K., Jin, Y., Boza, V., You, J., Cesista, F., Newhouse, L., and Bernstein, J. Muon: An optimizer for hidden layers in neural networks, 2024. URL https://kellerjordan.github.io/posts/muon/.

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.

Linux Foundation. Linux foundation annual report 2024: Accelerating industry innovation, 2024. URL https://www.linuxfoundation.org/hubfs/Reports/lf_ar24_121524a.pdf. Accessed: 2026-03-31.

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.

Rajbhandari, S., Rasley, J., Ruwase, O., and He, Y. Zero: memory optimizations toward training trillion parameter models. In Cuicchi, C., Qualters, I., and Kramer, W. T. (eds.), *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*, pp. 20. IEEE/ACM, 2020. URL https://doi.org/10.1109/SC41405.2020.00024.

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.

Vyas, N., Morwani, D., Zhao, R., Shapira, I., Brandfonbrener, D., Janson, L., and Kakade, S. M. SOAP: Improving and stabilizing shampoo using adam for language modeling. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=IDxZhXrpNf.

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.

Zhao, Y., Gu, A., Varma, R., Luo, L., Huang, C., Xu, M., Wright, L., Shojanazeri, H., Ott, M., Shleifer, S., Desmaison, A., Balioglu, C., Damania, P., Nguyen, B., Chauhan, G., Hao, Y., Mathews, A., and Li, S. Pytorch FSDP: experiences on scaling fully sharded data parallel. *Proc. VLDB Endow.*, 16(12):3848–3860, 2023. URL https://www.vldb.org/pvldb/vol16/p3848-huang.pdf.