μLO: COMPUTE-EFFICIENT META-GENERALIZATION OF LEARNED OPTIMIZERS

Benjamin Thérien[1,2] **Charles-Étienne Joseph**[1,2] **Boris Knyazev**[1,2,4]
Edouard Oyallon[5] **Irina Rish**[1,2] **Eugene Belilovsky**[2,3]

[1]Université de Montréal; [2]Mila – Quebec AI Institute; [3]Concordia University, Montréal;
[4]Samsung AI Lab, Montréal; [5]ISIR, Sorbonne University, CNRS, Paris, France.

ABSTRACT

Learned optimizers (LOs) have the potential to significantly reduce the wall-clock training time of neural networks. However, they can struggle to optimize unseen tasks (*meta-generalize*), especially when training networks wider than those seen during meta-training. To address this, we derive the Maximal Update Parametrization (μP) for two state-of-the-art learned optimizer architectures and propose a simple meta-training recipe for μ-parameterized LOs (μLOs). Our empirical evaluation demonstrates that LOs meta-trained with our recipe substantially improve meta-generalization to wider unseen tasks when compared to LOs trained under standard parametrization (SP) using the same compute budget. We also empirically observe that μLOs exhibit unexpectedly improved meta-generalization to deeper networks ($5\times$ meta-training) and surprising generalization to much longer training horizons ($25\times$ meta-training) when compared to SP LOs.

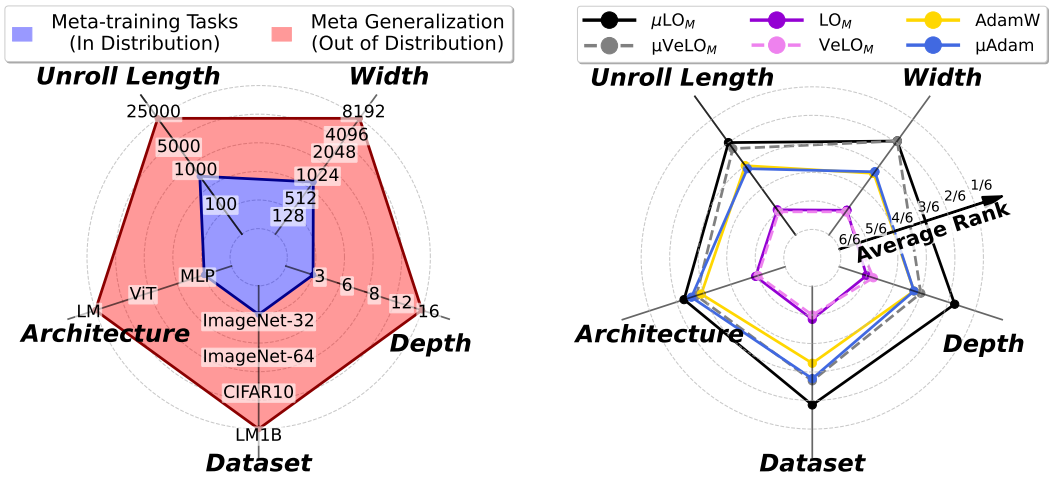
1 INTRODUCTION

While deep learning (DL) has largely replaced hand-designed algorithms, one crucial component of DL training remains hand-crafted: gradient-based optimizers. While popular optimizers such as Adam or SGD provably converge to a local minimum in non-convex settings (Kingma & Ba, 2017; Li et al., 2023a; Robbins, 1951), the existing literature provides no evidence that these optimizers converge to the global optimum at the optimal rate. With the lack of theory certifying the optimality of existing optimizers and the clear strength of data-driven methods, it is natural to turn towards data-driven solutions for improving the optimization of neural networks.

Taking a step in this direction, Andrychowicz et al. (2016); Wichrowska et al. (2017); Metz et al. (2019; 2022a) replace hand-designed optimizers with small neural networks called learned optimizers (LOs). LOs are meta-learned on a task distribution by minimizing the loss of the inner learning problem (e.g. neural network training in our case) across a batch of tasks. Being neural networks themselves, these optimizers are advantaged by their substantially larger parameter counts than Adam or SGD, making them suitable to large-scale meta-training. For instance, Metz et al. (2022b) showed that scaling up learned optimizer meta-training to 4000 TPU months can produce an optimizer, VeLO, that significantly outperforms well-tuned hand-designed optimizers without requiring hyperparameter tuning. However, even VeLO has limitations in *meta-generalization* – optimizing unseen problems. Specifically, VeLO (Metz et al., 2022b) is known to (1) have difficulty optimizing models much wider and deeper than those seen during meta-training (See Figures 6 and 9 of Metz et al. (2022b)) and (2) generalize poorly to longer optimization problems (e.g., training for more steps) than those seen during meta-training.

The problem of **meta-generalization** is fundamental to learned optimization due to the requirement for tractable meta-training and the expectation of strong performance across a combinatorially large set of downstream tasks. Meta-generalization refers to the ability of a meta-learned algorithm to *generalize*, that is, perform well when applied to unseen tasks. In the case of LOs, a learned optimizer

Correspondence to: Benjamin Thérien ⟨benjamin.therien@umontreal.ca⟩ and Eugene Belilovsky ⟨eugene.belilovsky@concordia.ca⟩. Our code is open-sourced: `https://github.com/bentherien/mu_learned_optimization`.



(a) Axes of Meta-Generalization

(b) Performance by Average Rank

Figure 1: **Meta-generalization is severely limited without our approach.** Subfigure (a) illustrates *meta-generalization* axes by distinguishing between meta-training tasks used herein (blue) and out-of-distribution tasks (red). Subfigure (b) reports the average rank across tasks within our evaluation suite that are out-of-distribution with respect to the corresponding axis. Both AdamW and μAdam undergo task-specific hyperparameter tuning across more than 500 configurations per task. Learned Optimizers of the same architecture are meta-learned on the same tasks with a FLOP-matched budget.

trained on a tractable and, thus, limited distribution of meta-training tasks should nevertheless exhibit strong performance when applied to out-of-distribution tasks: new combinations of architecture, dataset, and training objective (Figure 1). Even changes as small as increasing the hidden dimension of the architecture (width), the number of layers (depth), or the number of training steps (unroll length) can cause meaningful distribution shifts between meta-training and testing tasks, leading to poor generalization. Consequently, understanding and improving meta-generalization is central to making learned optimizers practical for real-world machine learning workloads.

In this work, we focus on the problem of LO meta-generalization to tasks of larger hidden dimension (width) than those seen during meta-learning. A related problem is that of transferring hyperparameters of hand-designed optimizers to wider tasks. Introduced by Yang et al. (2022), μP is an optimizer-dependent and width-dependent parameterization (e.g., a rule for initializing a model, scaling its pre-activations, and scaling the optimizer's updates) that allows hyperparameter transfer to larger width tasks for Adam and SGD. Making the connection between hyperparameter-transfer and meta-generalization, we ask: *Are existing learned optimizer architectures compatible with μP? Does meta-learning optimizers under μP improve meta-generalization?* To answer this question, we theoretically analyze two recent LO architectures (Metz et al., 2022a;b) (sec. 4), derive the appropriate maximal update parameterization for them, and carefully design a low-cost meta-training recipe to bring out their meta-generalization capabilities. We then provide extensive experimental evidence demonstrating that μLOs generalize to large unseen tasks.Our contributions are as follows:

- We derive μ-parameterization for two popular learned optimizer architectures (VeLO and small_fc_lopt) and demonstrate theoretically that our parameterization satisfies μP desiderata.
- We design a set of meta-training and meta-testing tasks enabling a systematic study of meta-generalization and demonstrate that our μLOs significantly outperform strong baseline LOs and hand-designed optimizers.
- We demonstrate empirically that our μLOs show surprisingly good generalization to deeper networks ($5\times$ meta-training) and longer training horizons ($25\times$ meta-training) when compared to baseline LOs.

2 BACKGROUND

Learned optimizer objective. A standard approach to learning optimizers (Metz et al., 2019) is to solve the following meta-learning problem:

$$\min_{\phi} \mathbb{E}_{(\mathcal{D},\mathcal{L},\boldsymbol{w}_0)\sim\mathcal{T}} \left[\mathbb{E}_{(X,Y)\sim\mathcal{D}} \left[\frac{1}{T} \sum_{t=0}^{T-1} \mathcal{L}(X, Y; f_\phi(\boldsymbol{u}_t), \boldsymbol{w}_t) \right] \right]. \tag{1}$$

Where $\mathcal{T}$ is a distribution over optimization tasks defined as tuples of dataset $\mathcal{D}$, objective function $\mathcal{L}$, and initial weights $\boldsymbol{w}_0$ associated with a particular neural architecture (we refer to this network as the *optimizee*); ϕ represents the weights of the learned optimizer,f_ϕ with input features $\boldsymbol{u}_t$; and T is the length of the unroll which we write as a fixed quantity for simplicity. In equation 1 and in our experiments, the sum of per-timestep loss is the quantity being optimized. It should be noted, however, that one could also optimize the final loss, final accuracy, or any other performance metric. Gradient descent is the preferred approach to solving equation 1. However, estimating meta-gradients via backpropagation is known to be problematic for long unrolls (Metz et al., 2019). Therefore, learned optimizer meta-gradients are estimated using evolution strategies and their variants (Vicol et al., 2021; Buckman et al., 2018; Nesterov & Spokoiny, 2017; Parmas et al., 2018; Vicol, 2023; Li et al., 2023b).

Learned optimizer input, output, and update. Learned optimizer neural architectures have taken many forms over the years, we will briefly review two recent architectures, **small_fc_lopt** of Metz et al. (2022a) and **VeLO** of Metz et al. (2022b), as they are used in our experiments. These learned optimizers construct input features $\boldsymbol{u}_t$ based on momentum accumulators, a variance accumulator, and multiple adafactor accumulators, we provide a full list in Tables 2, 3, and 4 of the Appendix. At every gradient descent step, small_fc_lopt and VeLO are applied to each parameter of the optimizee, producing two outputs: the magnitude (m) and direction (d) of the update. VeLO additionally outputs a tensor-level learning rate, $\alpha_{\boldsymbol{W}}$. The per-parameter update for both optimizers is given as

$$w_t = w_{t-1} - \alpha_{\boldsymbol{W}} \lambda_1 d \exp(\lambda_2 m), \tag{2}$$

where w is a parameter of weight matrix $\boldsymbol{W}$, λ_1 and λ_2 are constant values set to 0.001 to bias initial step sizes to be small. For small_fc_lopt, $\alpha_{\boldsymbol{W}} = 1$ always. We refer interested readers to appendix sections A.1.1 and A.1.2 for more details.

3 RELATED WORK

Generalization in LOs. There are three main difficulties of learned optimizer generalization (Chen et al., 2022; Amos, 2022): (1) optimizing unseen tasks; (2) optimizing beyond maximum unroll length seen during meta-training; (3) training optimizees that do not overfit. Among these, (3) has been most extensively addressed as this problem has been well studied in classic optimization literature. For example, extra-regularization terms can be directly applied to a learned optimizer (Harrison et al., 2022; Yang et al., 2023). In addition, (3) can be addressed by meta-training on a validation set objective (Metz et al., 2019) or parameterizing LOs as hyperparameter controllers (Almeida et al., 2021). The problem (2) has been mitigated by regularization (Harrison et al., 2022; Yang et al., 2023) and larger-scale meta-training (Metz et al., 2022b). However, (1) has remained a more difficult and understudied problem.

To the best of our knowledge, the only current approach to tackle this problem is to meta-train LOs on thousands of tasks (Metz et al., 2022b). However, this approach is extremely expensive and seems bound to fail in the regime where the optimizer is expected to generalize from small meta-training tasks in standard parameterization to large unseen tasks: figures 6 and 9 of Metz et al. (2022b) demonstrate that this was not achieved even when using 4000 TPU-months of compute. Generalization would be expected if all tasks, no matter the size, were included in the meta-training distribution, but such an approach is simply intractable and is likely to remain so.

Maximal Update Parametrization and Hyperparameter transfer. First proposed by Yang & Hu (2021), the Maximal Update Parametrization is the unique stable abc-Parametrization where every layer learns features. The parameterization was derived for adaptive optimizers by Yang & Littwin (2023) and was applied by Yang et al. (2022) to enable zero-shot hyperparameter transfer

for Adam and SGD. Most recently, in tensor programs VI, Yang et al. (2024) propose Depth-μP, a parameterization allowing for hyperparameter transfer in infinitely deep networks. While it is appealing, Depth-μP is only valid for residual networks with a block depth of 1, so it does not apply most practical architectures (e.g., transformers, resnets, etc.). For these reasons, we do not study Depth-μP herein. Following from the original discovery of hyperparameter transfer in Yang et al. (2022), a number of follow-up works have emerged that are not part of the tensor programs series. Dey et al. (2024) investigates transferring hyperparameters across different sparsity levels and widths. Blake et al. (2025) investigates a combination of μP and unit scaling, which results in easier tuning and more stable low-precision training. Everett et al. (2024) investigate the alignment assumptions of Yang et al. (2022) and find that appropriate per-layer learning rate prescriptions can also enable hyperparameter transfer in standard, mean field, and NTK parameterizations. In their empirical investigation of scaling exponents across these parameterizations, the authors find that SP with layer-wise learning rates outperforms μP. While we study the impact of meta-learning optimizers in μP on meta-generalization herein, it is still an open question which parameterization is best for meta-learning optimizers. Finally, in concurrent work, Dey et al. (2025) propose CompleteP, a parameterization that can achieve transfer of optimal hyperparameters across depth and width.

4 μ-PARAMETRIZATION FOR LEARNED OPTIMIZERS

Parameterizing an optimizee neural network in μP requires special handling of the initialization variance, pre-activation multipliers, and optimizer update for each weight matrix $\boldsymbol{W} \in \mathbb{R}^{n \times m}$ in the network. Specifically, these quantities will depend on the functional form of the optimizer and the dependence of n (FAN_OUT) and m (FAN_IN) on width. We will refer to weight matrices in a network of width h as hidden layers if $\Theta(n) = \Theta(m) = \Theta(h)$, as output layers if $\Theta(n) = \Theta(1) \wedge \Theta(m) = \Theta(h)$, and as input layers if $\Theta(n) = \Theta(h) \wedge \Theta(m) = \Theta(1)$. Here, Θ is standard asymptotic notation. Note that all biases and the weights of normalization layers are considered input layers and should be scaled as such. With this in mind, consider an arbitrary neural network[1] whose weight matrices are denoted $\boldsymbol{W}_l$, where l indexes the layers; the following modifications are then required to obtain μP for learned optimizers.

Optimizee Initialization-μ. If $\boldsymbol{W}_l$ belongs to a hidden or input layer, its weights should be initialized as $\mathcal{N}(0, \frac{1}{\text{FAN\_IN}})$. Output layers should have their weights initialized as $\mathcal{N}(0, 1)$.

Optimizee Multipliers-μ. Output layer pre-activations should be multiplied by $\frac{1}{\text{FAN\_IN}}$ during the forward pass.

Optimizer Update Scaling-μ. The learned optimizer's update (eq. 2) is re-scaled as follows:

$$w_t = \begin{cases} w_{t-1} - \frac{1}{\text{FAN\_IN}} \cdot \left(\alpha_{\boldsymbol{W}_l} \lambda_1 d \exp\left(\lambda_2 m\right)\right) & \boldsymbol{W}_l \text{ is a hidden layer} \\ w_{t-1} - \alpha_{\boldsymbol{W}_l} \lambda_1 d \exp\left(\lambda_2 m\right) & \text{otherwise.} \end{cases} \tag{3}$$

Where w is a parameter of the weight matrix, $\boldsymbol{W}_l$, and the dependence of d and m on w_{t-1} is not made explicit for simplicity. For transfer to the largest width optimizees, it may also become necessary to re-scale numerical underflow constants (ϵ) by $\frac{1}{\text{FAN\_IN}}$ as suggested by (Everett et al., 2024). However, for the scales reported on by our experiments, we did not find this to be necessary.

We now prove that our parameterization satisfies the μP Desiderata ((Yang et al., 2022) Sec. J.2.1).

Proposition 4.1 ($small\_fc\_lopt$ μP)**.** *Assume that the Learned Optimizer f_ϕ has the form $small\_fc\_lopt$ is fed with features given in Appendix A.1.1 and that during training the optimizee's parameters and input data become aligned, leading to Law of Large Numbers (LLN) scaling, then the update, initialization, and pre-activation multiplier above is sufficient to obtain a Maximal Update Parametrization.*

Proposition 4.2 (VeLO μP)**.** *Assume that ϕ in Proposition 4.1 is generated using an LSTM with the input features described in Appendix A.1.2 and that during training the optimizee's parameters and input data become aligned, leading to Law of Large Numbers (LLN) scaling, then the update, initialization, and pre-activation multiplier above is sufficient to obtain a Maximal Update Parametrization.*

Proof. The proof is provided in Appendix A.2. □

[1]The μLO parameterization can be applied to any neural network architecture.

5 EMPIRICAL EVALUATION

We construct a suite of optimization tasks of varying width to evaluate the meta-generalization properties of our μLOs meta-trained on MLPs vs per-task tuned μAdam (Yang et al., 2022), per-task tuned SP AdamW (Loshchilov & Hutter, 2019), and baseline SP LOs (meta-trained on MLP tasks). Our main focus is to evaluate meta-generalization to wider networks as this is a key weakness of learned optimizers in previous works. However, we also establish the generalization properties of μLOs to deeper networks and longer training horizons. Please note that while μLOs inherit the theoretical properties of μP for width scaling, our findings with respect to deeper networks and longer training are purely empirical.

5.1 SETUP

Baseline LOs and μLOs. The meta-training configuration of each learned optimizer is summarized in Table 5. Each learned optimizer (ours and the baselines) in our empirical evaluation is meta-trained using the multiple-width single-task meta-training recipe proposed in section 5.2.1. **Notably, these tasks only include MLPs (see Fig 1), while the hand-desinged optimizers in our study are tuned individually on each task.** The SP baselines sheds light on whether simply varying the SP optimizee width during meta-training is enough to achieve generalization of the LO to wider networks in SP. During meta-training, we set the inner problem length to be 1000 iterations. Therefore, any optimization beyond this length is considered out-of-distribution. For all meta-training and hyperparameter tuning details, including ablation experiments, see section C of the appendix.

μAdam is a strong hand-designed μP baseline. It follows the Adam μ-parametrization and does not use weight decay as this is incompatible with μP (Yang et al., 2022). μAdam is tuned on a width=1024 version of each task as this is the width of the largest meta-training task seen by our learned optimizers (see Table 5). We tune the learning rate (η) and accumulator coefficients (β_1 and β_2) using a grid search over more than 500 different configurations. This is repeated once for each task in our suite. Section B.1 of the appendix provides more details about the grid search including the values swept and the best values found.

AdamW (Loshchilov & Hutter, 2019) is a strong hand-designed SP baseline. It is tuned on the largest meta-training task seen by our learned optimizers (Table 5). AdamW is tuned on a width=1024 version of each task as this is the width of the largest meta-training task seen by our learned optimizers (see Table 5). We tune the learning rate (η), accumulator coefficients (β_1 and β_2), and weight decay (λ) using a grid search over more than 500 different configurations. This is repeated once for each task in our suite. Section B.2 of the appendix provides more details about the grid search including the values swept and the best values found.

Evaluation tasks. Our evaluation suite includes 35 tasks spanning image classification (CIFAR-10, ImageNet) using MLPs and Vision Transformers (ViTs) (Dosovitskiy et al., 2020) and autoregressive language modeling with a decoder-only transformer on LM1B (Chelba et al., 2013). To create the tasks, we further vary image size (for image classification), width, and depth of the optimizee network, and the number of optimization steps. See Table 10 of the appendix for an extended description of all the tasks.

5.2 RESULTS

In the following sections, we evaluate different meta-training distributions for training μLOs (Sec. 5.2.1); we present results empirically verifying the pre-activation stability of our μLOs (Sec. 5.2.2); we present the results of our main empirical evaluation of meta-generalization to wider networks (Sec. 5.2.2); a study of μLOs generalization to deeper networks (Sec. 5.2.4); and a study of μLOs generalization to longer training horizons (Sec. 5.2.4). All of our figures report training loss and report the average loss across 5 random seeds. Each seed corresponds to a different ordering of training data and a different initialization of the optimizee. All error bars in our plots report standard error across seeds. Standard error is $\frac{\sigma}{\sqrt{n}}$ where σ is the population standard deviation and n is the number of samples.

5.2.1 EVALUATING META-TRAINING DISTRIBUTIONS FOR μLOS

In μ-transfer (Yang et al., 2022), hyperparameters are typically tuned on a small proxy task before being transferred to the large target task. In contrast, learned optimizers are typically meta-trained

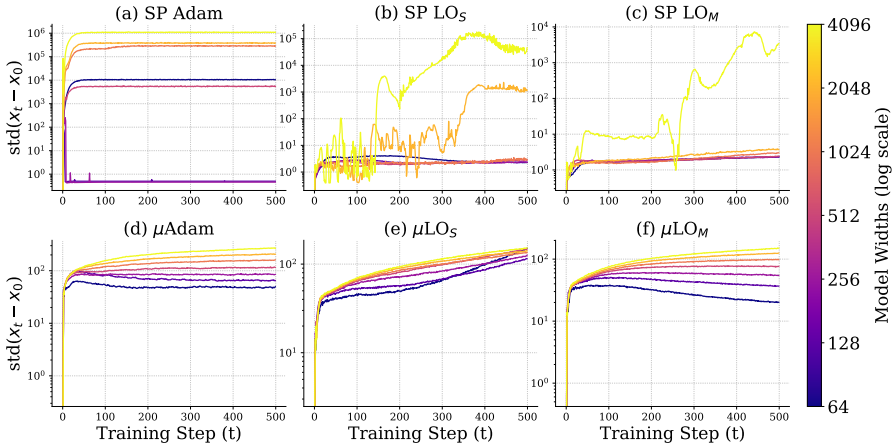
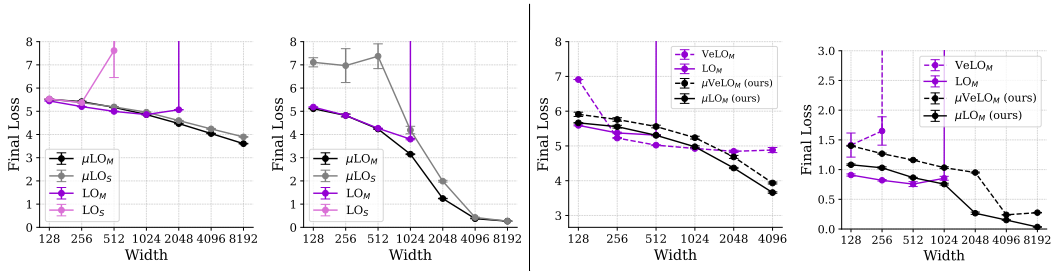


Figure 2: **Layer 2 pre-activations behave harmoniously in μP for μLOs and μAdam alike.** We report the evolution of coordinate-wise standard deviation of the difference between the initial ($t = 0$) and t-th second-layer pre-activations of an MLP during training for the first 500 steps of a single run (the remaining layers behave similarly, see Sec. G). We observe that all models parameterized in μP enjoy stable coordinates across widths, while the pre-activations of larger-width models in SP blow up after a number of training steps.



(a) Iteration 1000 (b) Iteration 5000 (c) $64 \times 64 \times 3$ ImageNet (d) $32 \times 32 \times 3$ Cifar10

Figure 3: **Generalization beyond meta-training widths is severely limited without our approach.** Each point is the average final training loss over 5 seeds with standard error bars. Subfigures (a) and (b) report the results of our meta-training task ablation on the ImageNet-32 meta-training tasks at 1000 and 5000 steps. Subfigures (c) and (d) report the performance of μLO$_M$ and μVeLO$_M$ on OOD datasets.

on a distribution of tasks. To verify the effectiveness of each approach for meta-training μLOs, we compare μLO$_S$, meta-trained on a single width=128 MLP ImageNet classification task (see Tab. 5), to μLO$_M$, meta-trained on width $\in \{128, 512, 1024\}$ MLP ImageNet classification tasks. Each optimizer targets 1000 step problems. We include equivalent standard parameterization baselines for reference (LO$_S$ and LO$_M$). Figure 3 reports the performance of each optimizer on a suite of MLP classification tasks of increasing width. When training for 1000 steps (meta-training unroll length), we observe that μLO$_M$ outperforms μLO$_S$ as the width of the model is increased (Fig. 3 (a)). Moreover, we observe that there is a discrepancy in performance between both models after 5000 steps (Fig. 3 (b)), showing that meta-training with multiple tasks of different widths has benefits for generalization to longer unrolls in addition to improved generalization to larger optimizees. Given the improved generalization of μLO$_M$ compared to μLO$_S$, we adopt the multiple-width meta-training recipe as part of our method. Subsequent experiments (e.g., Figures 3 and 4) will show that our recipe is also effective for meta-training μVeLO.

5.2.2 Evaluating pre-activation stability

We now verify that desiderata J.1 of Yang et al. (2022) is satisfied empirically. In Figure 2, we report the evolution of the coordinate-wise standard deviation of the difference between initial (t=0) and current (t) second-layer pre-activations of an MLP during the first 500 steps of training for a

single trial. We observe that all models parameterized in μP enjoy stable coordinates across widths, suggesting that desiderata J.1 is satisfied by our parameterization. In contrast, the pre-activations of the larger MLPs in SP blow up immediately for SP Adam while they take noticeably longer for LO_S and LO_M. Section G of the appendix contains similar plots for the remaining layers of the MLP which show similar trends. In summary, we find, empirically, that pre-activations of μLOs and μAdam are similarly stable across widths, while the activations of SP Adam and SP LOs both blow up but behave qualitatively differently.

5.2.3 META-GENERALIZATION TO WIDER NETWORKS

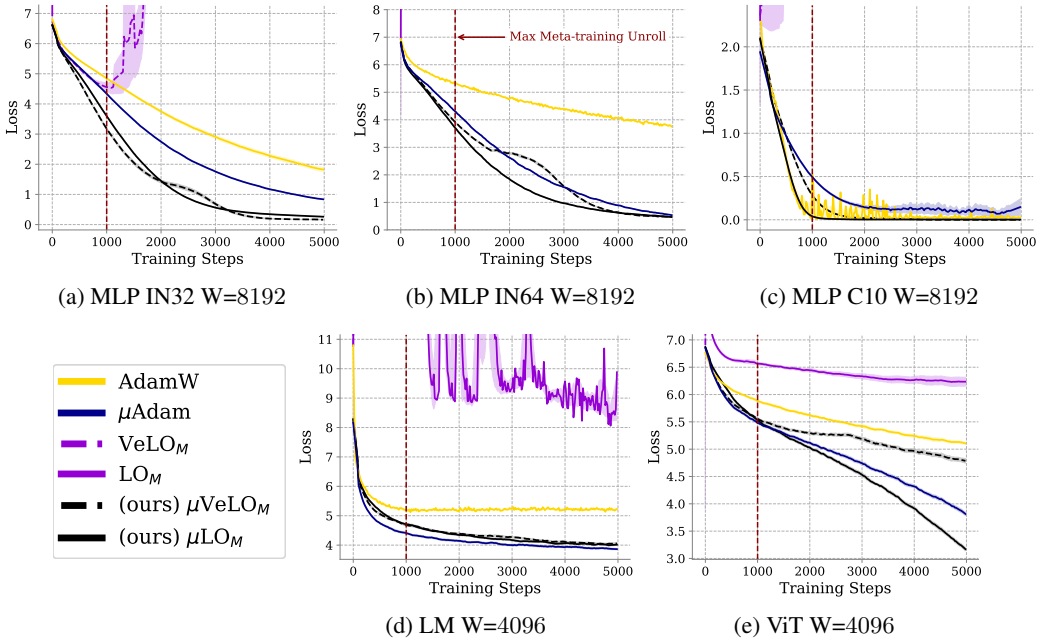
Given our goal of improving LO generalization to unseen wider tasks, the bulk of our empirical evaluation is presented in this section. Specifically, we evaluate the behavior of μLOs as the width of tasks increases well beyond what was seen during meta-training. To accomplish this, we fix the depth of each task and vary the width (see Table 10 for a full list of tasks), leading to a testbed of 32 different tasks. We then train each task using the baselines and μ-optimizers outlined in section 5 for 5000 steps for 5 different random seeds. This involves training 1120 different neural networks. To make the results easily digestible, we summarize them by width and final performance in Figure 4 and by average optimizer rank in Table 1. We also highlight the smooth training dynamics of our optimizers at the largest widths in Figure 4.

Performance measured by final loss as a function of width. Figure 3 compares the training loss after 1000 steps of SP learned optimizers to μ-parameterized learned optimizers for different widths. This is shown in three subfigures for three MLP image classification tasks: (a) Imagenet $32 \times 32 \times 3$ (IN32), (c) Imagenet $64 \times 64 \times 3$ (IN64), and (d) Cifar-10 $32 \times 32 \times 3$ (C10). Subfigure (a) shows the performance of learned optimizers on larger versions of the meta-training tasks. We observe that the μLOs achieve lower final training loss as the width of the task is increased. In contrast, LO_M diverges for widths larger than 2048. Subfigure (b) evaluates our μLOs on $64 \times 64 \times 3$ ImageNet images (e.g., when the input width is larger). Similarly, we observe smooth improvements in the loss as the optimizee width increases for μLOs, while their SP counterparts either diverge at width 512 (LO_M) or fail to substantially improve the loss beyond width 1024 (VeLO_M). Finally, Subfigure (c) shows the performance of our μLOs on Cifar-10 (smaller output width) as the width is increased. Similarly, we observe smooth improvements in the loss as the width increases for μLOs, while their SP counterparts either diverge immediately at small widths (VeLO_M) or diverge by width 1024 (LO_M).

Training dynamics at the largest widths Figure 4 reports the training curves of different optimizers on the largest width tasks in our suite. Despite training for $5\times$ longer than the maximum meta-training unroll length, our μLOs are capable of smoothly decreasing the loss for the largest out-of-distribution tasks in our suite. In contrast, the strong SP LO baselines diverge by 1000 steps (subfigures (a),(b),(c),(d)), or fail to decrease the training loss (subfigure (e)), demonstrating the clear benefit of μLOs for learned optimization. Our μLOs also substantially best the per-task-tuned AdamW and μAdam baselines (subfigures (a) and (b)), match the best performing hand-designed optimizer in subfigure (c), and nearly matches or outperforms the strongest hand-designed baseline performance on far out-of-distribution LM and ViT tasks (subfigures (d) and (e)). These results demonstrate that, under our μLO meta-training recipe, learning optimizers that smoothly train large neural networks (e.g., demonstrated an 8B parameter model typically uses width=4096) is possible at low cost (μLO_M is meta-trained for 100 GPU hours).

Table 1: **Summary of optimizer performance on large tasks.** We report the average rank of different optimizers across the five tasks in our suite. We evaluate each optimizer on large-width tasks: Large (2048), XL (4096 for MLPs and 3072 for vit and LM), and XXL (largest size for each task see Tab.10 of the appendix). We bold the strongest, underline the second strongest, and italicize the third strongest average rank in each column. We observe that, across all iterations, μLO_M and μVeLO_M consistently obtain the best and second-best ranks for all tasks.

	Loss at 1k steps			Loss at 3k steps			Loss at 5k steps		
Optimizer	**OoD (Large)**	**OoD (XL)**	**OoD (XXL)**	**OoD (Large)**	**OoD (XL)**	**OoD (XXL)**	**OoD (Large)**	**OoD (XL)**	**OoD (XXL)**
AdamW	*3.00*	3.60	4.40	*2.80*	2.60	4.00	*2.60*	*2.40*	3.80
μAdam	3.40	*2.20*	*2.20*	3.00	*2.40*	*2.40*	3.20	2.60	*2.60*
VeLO_M	4.60	4.00	5.00	5.40	5.40	5.80	6.00	5.40	5.80
LO_M	5.60	5.40	5.60	5.60	4.80	5.20	5.00	4.80	5.20
μVeLO_M (ours)	<u>2.60</u>	**1.60**	**1.80**	<u>2.40</u>	<u>2.00</u>	<u>2.40</u>	<u>2.40</u>	**1.40**	<u>2.00</u>
μLO_M (ours)	**1.80**	<u>2.00</u>	<u>2.00</u>	**1.80**	**1.60**	**1.20**	**1.80**	<u>2.20</u>	**1.60**



(a) MLP IN32 W=8192 (b) MLP IN64 W=8192 (c) MLP C10 W=8192

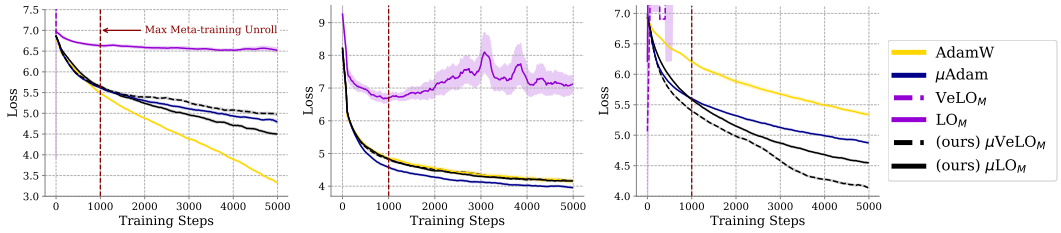
(d) LM W=4096 (e) ViT W=4096

Figure 4: **Evaluating generalization to wider networks for different tasks.** All optimizers are meta-trained or hyperparameter tuned for 1000 inner steps (dotted red line), therefore, any optimization beyond 1000 steps is considered out-of-distribution. We plot average training loss over 5 seeds with standard error bars. We observe that μLO$_M$ and μVeLO$_M$generalize smoothly to longer unrolls and all unseen tasks, unlike their SP counterparts which diverge or fail to make progress. μLOs outperform the extensively tuned AdamW and μAdam baselines in subfigures (a),(b), match or surpass them in subfigure (c), and exceed or nearly match their performance on far out-of-distribution LM and ViT tasks (subfigures (d) and (e)). Note that all AdamW and μAdam are tuned on smaller versions of each task, while our μLOs are only meta-trained on MLP tasks.

Performance measured by average optimizer rank Table 1 reports the average rank of different optimizers on out-of-distribution w.r.t. width tasks (Large (width 2048), XL (width 3072 for transformer and 4096 for MLPs), and XXL (maximum width)). Each entry of the table corresponds to the optimizer's average rank (within the 6 optimizers evaluated) over the 5 tasks in our suite: Cifar 10 MLP image classification, ImageNet 32 MLP image classification, ImageNet 64 MLP image classification, ImageNet 32 ViT image classification, and LM1B transformer language modeling. The optimizers are ranked by their training loss at the given iteration. We report average ranks for 1000 iterations (inner-problem length), 3000 iterations, and 5000 iterations. We **bold** the strongest, underline the second strongest, and *italicize* the third strongest average rank in each column. We observe that, across all iterations and all task sizes (Large, XL, XXL), either μLO$_M$ or μVeLO$_M$ consistently obtain the best and second-best ranks for all tasks. The per-task-tune hand-designed baselines consistently occupy third and fourth rank, while the SP learned optimizer baselines perform worst, typically failing to optimize at this size. These results demonstrate that meta-training learned optimizers under the μ-parameterization we propose and using our simple meta-training recipe yields substantial improvements in meta-generalization (across various tasks and widths) over SP LOs (previous work) and strong per-task tuned hand-designed baselines.

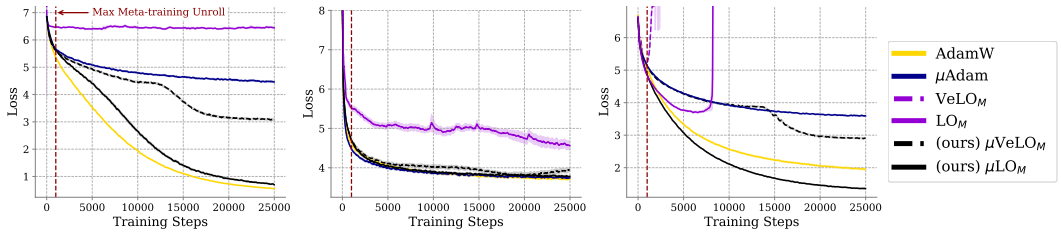
5.2.4 Evaluating Meta-generalization Beyond Width

While our main focus is meta-generalization to wider networks While the focus of our paper is improving the meta-generalization of LOs on wider tasks, it is also important to evaluate how these modifications to learned optimizer meta-training impact other axes of generalization. As such, we now study meta-generalization to deeper networks and longer training. While we provide strong AdamW and μAdam baselines for reference, our focus will be to establish the relative performance μLOs to SP LOs. Note that μP theory leveraged by μLOs specifically concerns transferring hyperparameters to



(a) ViT W=1024 D=16 (b) LM W=1024 D=16 (c) MLP IN32 W=1024 D=16

Figure 5: **Evaluating generalization capabilities of μLOs to deeper networks**. Our focus is on comparing the meta-generalization to deeper tasks of μLOs to SP LOs (all meta-trained exclusively on MLPs). We also report the performance per-task tuned AdamW and μAdam for reference. Each plot reports average training loss over 5 seeds with standard error bars. In each case, μLOs show improved generalization and performance when compared to their SP counterparts.



(a) ViT W=1024 (b) LM W=1024 (c) MLP IN32 W=1024

Figure 6: **Evaluating meta-generalization to longer training horizons**. Note that AdamW and μAdam are evaluated on their tuning tasks here, while LOs are trained on MLPs. We plot average training loss over 5 seeds with standard error bars. We observe that μLOs seamlessly generalize to training horizons $25\times$ longer than meta-training. In contrast, the best performing SP LO fails to decrease training loss (a), decreases it but suffers instabilities (b), or diverges after 8000 steps (c).

larger-width networks, not longer training horizons or deeper networks. Therefore, any improvements we observe are purely empirical.

Meta-generalization to deeper networks In this section, we evaluate LO meta-generalization to deeper networks. Specifically, we increase the number of layers used in MLP, ViT, and LM tasks from 3 to 16, while keeping width=1024 within the range of tuning/meta-training. Figure 5 reports the performance of our learned optimizers on deeper networks. We observe that both μLO_M and μVeLO_M optimize stably throughout and generally outperform their counterparts, LO_M and VeLO_M, by the end of training on each task, despite being meta-trained on MLPs of exactly the same depth. Moreover, LO_M immediately diverges when optimizing the deep MLP while μLO_M experiences no instability. Similarly, VeLO_M diverges on ViTs and Transformers, while μVeLO_M performs well, especially on ViTs. This is remarkable as, unlike width, there is no theoretical justification for μP's benefit to deeper networks. We hypothesize that μP's stabilizing effect on the optimizee's activations leads to this improvement in generalization (see Sec. F.1.2 for more details).

Meta-generalization to longer training In this subsection, we empirically evaluate the capability of μLOs to generalize to much longer training horizons than those seen during meta-training. Specifically, we use μLO_M and LO_M as well as μVeLO_M and VeLO_M to train three networks with width $w = 1024$: a 3-layer MLP, ViT on $32 \times 32 \times 3$ ImageNet and a 3-layer Transformer for autoregressive language modeling on LM1B. Each model is trained for $25,000$ steps ($25\times$ the longest unroll seen at meta-training time). Figure 6 reports the training loss averaged over 5 random seeds. We observe that μLO_M and μVeLO_M stably decrease training loss over time for each task, while LO_M and VeLO_M fail to decrease training loss (a), decreases it but becomes unstable (b), or diverges after 8000 steps (c). While we are uncertain of the exact cause of this improved generalization, we hypothesize that it may be due to the improved pre-activation stability (see Sec. F.1.2 for more details). These results suggest that generalization to longer training horizons is another benefit of using μLOs.

6 LIMITATIONS

We have conducted a systematic empirical study and shown strong results within the scope of our study, there are some limitations of our work. Specifically, (1) we do not meta-train on tasks other than MLPs for image classification, (2) we do not provide an evaluation of models wider than 8192 (MLPs) and $3072/12288$ (transformer hidden/FFN size) due to computational constraints in our academic environment, and (3) We did not include an oracle SP AdamW baseline whose hyperparameters are swept at every width due to computational constraints in our academic environment.

7 CONCLUSION

We have theoretically and empirically demonstrated that it is possible to obtain a valid μ-parameterization for two state-of-the-art learned optimizer architectures. Under or proposed meta-training recipe, meta-learned optimizers show substantial improvements in meta-generalization properties when compared to strong baselines from previous work. Remarkably, our μLOs, meta-trained only on MLP tasks, surpass the performance of per-task-tuned hand-designed baselines in terms of average rank on wide OOD tasks. Moreover, our experiments also show that μLOs meta-trained with our recipe generalize better to wider and, unexpectedly, deeper out-of-distribution tasks than their SP counterparts. When evaluated on much longer training tasks, we observe that μLOs have a stabilizing effect, enabling meta-generalization to much longer unrolls ($25\times$ maximum meta-training unroll length). All of the aforementioned benefits of μLOs come at *zero* extra computational cost compared to SP LOs. Our results outline a promising path forward for low-cost meta-training of learned optimizers that can generalize to large unseen tasks.

In future work, it will be important to investigate the benefits of meta-learning optimizers under parameterizations other than μP that have been shown to admit hyperparameter transfer (Everett et al., 2024). Another important direction of inquiry is to investigate the meta-learning optimizers under parameterizations, like CompleteP (Dey et al., 2025), that have the potential to improve meta-generalization across depth and width. Finally, combining such parameterizations with improved meta-generalization and scalable meta-learning recipes is required for learning truly general-purpose optimizers.

ACKNOWLEDGMENTS

We acknowledge support from the Mila-Samsung Research Grant, FRQNT New Scholar [*E.B.*], the FRQNT Doctoral (B2X) scholarship [*B.T.*], the Canada CIFAR AI Chair Program [*I.R.*], and the Canada Excellence Research Chairs Program in Autonomous AI [*I.R.*]. We also acknowledge resources provided by Compute Canada, Calcul Québec, and Mila. [*E.O.*] acknowledges funding from PEPR IA (grant SHARP ANR-23-PEIA-0008). He was granted access to the AI resources of IDRIS under the allocation 2025- AD011015884R1.

REFERENCES

Diogo Almeida, Clemens Winter, Jie Tang, and Wojciech Zaremba. A generalizable approach to learning optimizers. *arXiv preprint arXiv:2106.00958*, 2021. 3, 26

Brandon Amos. Tutorial on amortized optimization for learning to optimize over continuous domains. *arXiv e-prints*, pp. arXiv–2202, 2022. 3, 26

Marcin Andrychowicz, Misha Denil, Sergio Gomez, Matthew W Hoffman, David Pfau, Tom Schaul, Brendan Shillingford, and Nando De Freitas. Learning to learn by gradient descent by gradient descent. *Advances in neural information processing systems*, 29, 2016. 1, 26

Charlie Blake, Constantin Eichenberg, Josef Dean, Lukas Balles, Luke Y. Prince, Björn Deiseroth, Andres Felipe Cruz-Salinas, Carlo Luschi, Samuel Weinbach, and Douglas Orr. u-μp: The unit-scaled maximal update parametrization, 2025. URL https://arxiv.org/abs/2407.17465. 4

Jacob Buckman, Danijar Hafner, George Tucker, Eugene Brevdo, and Honglak Lee. Sample-efficient reinforcement learning with stochastic ensemble value expansion. In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett (eds.),

Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada, pp. 8234–8244, 2018. 3, 26

Ciprian Chelba, Tomas Mikolov, Mike Schuster, Qi Ge, Thorsten Brants, and Phillipp Koehn. One billion word benchmark for measuring progress in statistical language modeling. *CoRR*, abs/1312.3005, 2013. 5

Tianlong Chen, Weiyi Zhang, Zhou Jingyang, Shiyu Chang, Sijia Liu, Lisa Amini, and Zhangyang Wang. Training stronger baselines for learning to optimize. *Advances in Neural Information Processing Systems*, 33:7332–7343, 2020. 26

Tianlong Chen, Xiaohan Chen, Wuyang Chen, Zhangyang Wang, Howard Heaton, Jialin Liu, and Wotao Yin. Learning to optimize: A primer and a benchmark. *The Journal of Machine Learning Research*, 23(1):8562–8620, 2022. 3, 26

Nolan Dey, Shane Bergsma, and Joel Hestness. Sparse maximal update parameterization: A holistic approach to sparse training dynamics. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 33836–33862. Curran Associates, Inc., 2024. doi: 10.52202/079017-1066. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/3b6aaffec941f98930753fa6d6de7263-Paper-Conference.pdf. 4

Nolan Dey, Bin Claire Zhang, Lorenzo Noci, Mufan Bill Li, Blake Bordelon, Shane Bergsma, Cengiz Pehlevan, Boris Hanin, and Joel Hestness. Don't be lazy: Completep enables compute-efficient deep transformers. *CoRR*, abs/2505.01618, 2025. URL https://doi.org/10.48550/arXiv.2505.01618. 4, 10

Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020. 5

Katie E. Everett, Lechao Xiao, Mitchell Wortsman, Alexander A. Alemi, Roman Novak, Peter J. Liu, Izzeddin Gur, Jascha Sohl-Dickstein, Leslie Pack Kaelbling, Jaehoon Lee, and Jeffrey Pennington. Scaling exponents across parameterizations and optimizers. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net, 2024. 4, 10

James Harrison, Luke Metz, and Jascha Sohl-Dickstein. A closer look at learned optimization: Stability, robustness, and inductive biases. *Advances in Neural Information Processing Systems*, 35:3758–3773, 2022. 3, 26

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017. 1

Taku Kudo and John Richardson. Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. In Eduardo Blanco and Wei Lu (eds.), *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, EMNLP 2018: System Demonstrations, Brussels, Belgium, October 31 - November 4, 2018*, pp. 66–71. Association for Computational Linguistics, 2018. 27

Haochuan Li, Alexander Rakhlin, and Ali Jadbabaie. Convergence of adam under relaxed assumptions. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine (eds.), *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023a. 1

Oscar Li, James Harrison, Jascha Sohl-Dickstein, Virginia Smith, and Luke Metz. Variance-reduced gradient estimation via noise-reuse in online evolution strategies. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023b. 3

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. 5, 25

Luke Metz, Niru Maheswaranathan, Jeremy Nixon, Daniel Freeman, and Jascha Sohl-Dickstein. Understanding and correcting pathologies in the training of learned optimizers. In *International Conference on Machine Learning*, pp. 4556–4565. PMLR, 2019. 1, 3, 26

Luke Metz, C. Daniel Freeman, James Harrison, Niru Maheswaranathan, and Jascha Sohl-Dickstein. Practical tradeoffs between memory, compute, and performance in learned optimizers, 2022a. 1, 2, 3, 14, 15, 23, 25, 26

Luke Metz, James Harrison, C Daniel Freeman, Amil Merchant, Lucas Beyer, James Bradbury, Naman Agrawal, Ben Poole, Igor Mordatch, Adam Roberts, et al. Velo: Training versatile learned optimizers by scaling up. *arXiv preprint arXiv:2211.09760*, 2022b. 1, 2, 3, 14, 16, 22, 23, 26, 27

Yurii E. Nesterov and Vladimir G. Spokoiny. Random gradient-free minimization of convex functions. *Found. Comput. Math.*, 17(2):527–566, 2017. 3, 26

Paavo Parmas, Carl Edward Rasmussen, Jan Peters, and Kenji Doya. PIPPS: flexible model-based policy search robust to the curse of chaos. In Jennifer G. Dy and Andreas Krause (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. 4062–4071. PMLR, 2018. 3, 26

Isabeau Premont-Schwarz, Jaroslav Vitkuu, and Jan Feyereisl. A simple guard for learned optimizers. *arXiv preprint arXiv:2201.12426*, 2022. 26

Herbert E. Robbins. A stochastic approximation method. *Annals of Mathematical Statistics*, 22: 400–407, 1951. 1

Jürgen Schmidhuber. Learning to control fast-weight memories: An alternative to dynamic recurrent networks. *Neural Computation*, 4(1):131–139, 1992. 26

Sebastian Thrun and Lorien Pratt. *Learning to learn*. Springer Science & Business Media, 2012. 26

Paul Vicol. Low-variance gradient estimation in unrolled computation graphs with es-single. In *International Conference on Machine Learning*, pp. 35084–35119. PMLR, 2023. 3, 26

Paul Vicol, Luke Metz, and Jascha Sohl-Dickstein. Unbiased gradient estimation in unrolled computation graphs with persistent evolution strategies. In Marina Meila and Tong Zhang (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. 10553–10563. PMLR, 2021. 3, 25, 26

Olga Wichrowska, Niru Maheswaranathan, Matthew W Hoffman, Sergio Gomez Colmenarejo, Misha Denil, Nando Freitas, and Jascha Sohl-Dickstein. Learned optimizers that scale and generalize. In *International conference on machine learning*, pp. 3751–3760. PMLR, 2017. 1, 26

Greg Yang. Tensor programs I: wide feedforward or recurrent neural networks of any architecture are gaussian processes. *CoRR*, abs/1910.12478, 2019. 27

Greg Yang. Tensor programs II: neural tangent kernel for any architecture. *CoRR*, abs/2006.14548, 2020a. 27

Greg Yang. Tensor programs III: neural matrix laws. *CoRR*, abs/2009.10685, 2020b. 27

Greg Yang and Edward J. Hu. Tensor programs IV: feature learning in infinite-width neural networks. In Marina Meila and Tong Zhang (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. 11727–11737. PMLR, 2021. 3, 27

Greg Yang and Etai Littwin. Tensor programs ivb: Adaptive optimization in the infinite-width limit. *CoRR*, abs/2308.01814, 2023. 3, 17, 27

Greg Yang, Edward J. Hu, Igor Babuschkin, Szymon Sidor, David Farhi, Jakub Pachocki, Xiaodong Liu, Weizhu Chen, and Jianfeng Gao. Tensor programs v: Tuning large neural networks via zero-shot hyperparameter transfer. In *NeurIPS 2021*, March 2022. 2, 3, 4, 5, 6, 17, 25, 26, 27

Greg Yang, Dingli Yu, Chen Zhu, and Soufiane Hayou. Tensor programs VI: feature learning in infinite depth neural networks. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. 4, 27, 29

Junjie Yang, Tianlong Chen, Mingkang Zhu, Fengxiang He, Dacheng Tao, Yingbin Liang, and Zhangyang Wang. Learning to generalize provably in learning to optimize. In *International Conference on Artificial Intelligence and Statistics*, pp. 9807–9825. PMLR, 2023. 3, 26

Biao Zhang and Rico Sennrich. Root mean square layer normalization. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (eds.), *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pp. 12360–12371, 2019. URL https://proceedings.neurips.cc/paper/2019/hash/1e8a19426224ca89e83cef47f1e7f53b-Abstract.html. 15

A PROOF OF PROPOSITION 4.1

For the reader's convenience, we will first review the input, output, update, and scaling of the per-parameter `small_fc_lopt` Metz et al. (2022a) learned optimizer as it is necessary background for understanding our proof. This corresponds to the architecture of the μLO_M, μLO_S, LO_M, and LO_S optimizers used throughout our experiments. In section A.1.2, we will also review the input, output, update, and scaling of VeLO, the architecture used for μVeLO_M and VeLO_M. Note that the VeLO Metz et al. (2022b) architecture uses an almost-identical `small_fc_lopt` network to produce per-parameter updates. The main difference is that VeLO uses an LSTM to generate the parameters of `small_fc_lopt` for each tensor in the network at each optimization step.

A.1 μLO_M AND μVeLO_M INPUT, OUTPUT, UPDATE, AND SCALING.

A.1.1 THE SMALL_FC_LOPT ARCHITECTURE

`small_fc_lopt` maintains three different per-parameter momentum accumulators ($\boldsymbol{M}_{t,i}$) and one variance accumulator ($\boldsymbol{V}_t$). In addition, it also maintains six adafactor-style accumulators of the column-wise ($\boldsymbol{c}_{t,i}$) and row-wise ($\boldsymbol{r}_{t,i}$) mean of the squared gradient. The accumulator update is given as follows:

$$
\begin{aligned}
\boldsymbol{M}_{t,i} &= \beta_i \boldsymbol{M}_{t-1,i} + (1-\beta_i)\nabla_t && i \in \{1,2,3\},\\
\boldsymbol{V}_t &= \beta_4 \boldsymbol{V}_{t-1} + (1-\beta_4)\nabla_t^2, &&\\
\boldsymbol{r}_{t,i} &= \beta_i \boldsymbol{r}_{t-1,i} + (1-\beta_i)\ \texttt{row\_mean}(\nabla_t^2), && i \in \{5,6,7\},\\
\boldsymbol{c}_{t,i} &= \beta_i \boldsymbol{c}_{t-1,i} + (1-\beta_i)\ \texttt{col\_mean}(\nabla_t^2), && i \in \{5,6,7\},\\
\boldsymbol{U}_t &:= [\boldsymbol{M}_{t,1}, \boldsymbol{M}_{t,2}, \boldsymbol{M}_{t,3}, \boldsymbol{V}_t, \boldsymbol{r}_{t,5}, \boldsymbol{r}_{t,6}, \boldsymbol{r}_{t,7}, \boldsymbol{c}_{t,5}, \boldsymbol{c}_{t,6}, \boldsymbol{c}_{t,7}].
\end{aligned}
$$

Here, we slightly abuse notation and define $\boldsymbol{U}_t$ to be the entire accumulator state for all parameters in the optimizee (column-wise and row-wise features are repeated for notational convenience). After updating these accumulators, `small_fc_lopt` computes additional learned optimizer input features:

$$
\begin{aligned}
\boldsymbol{F}_i^{(\nabla)} &= \nabla_t \odot \sqrt{\frac{\frac{1}{m}\sum_{h=1}^{m}(\boldsymbol{r}_{t,i})_h}{\boldsymbol{r}_{t,i}\boldsymbol{c}_{t,i}^T}},\\
\boldsymbol{F}_i^{(\boldsymbol{M})} &= \boldsymbol{M}_{t,j} \odot \sqrt{\frac{\frac{1}{m}\sum_{h=1}^{m}(\boldsymbol{r}_{t,i})_h}{\boldsymbol{r}_{t,i}\boldsymbol{c}_{t,i}^T}},\\
\boldsymbol{R}_t &= \left[\frac{1}{\sqrt{\boldsymbol{r}_{t,5}}}, \frac{1}{\sqrt{\boldsymbol{r}_{t,6}}}, \frac{1}{\sqrt{\boldsymbol{r}_{t,7}}}, \frac{1}{\sqrt{\boldsymbol{c}_{t,5}}}, \frac{1}{\sqrt{\boldsymbol{c}_{t,6}}}, \frac{1}{\sqrt{\boldsymbol{c}_{t,7}}}, \frac{\boldsymbol{M}_{t,1}}{\sqrt{v}}, \frac{\boldsymbol{M}_{t,2}}{\sqrt{v}}, \frac{\boldsymbol{M}_{t,3}}{\sqrt{v}}, \frac{1}{\sqrt{v}}\right],\\
\boldsymbol{H}_t &= \left[\boldsymbol{F}_1^{(\nabla)}, \boldsymbol{F}_2^{(\nabla)}, \boldsymbol{F}_3^{(\nabla)}, \boldsymbol{F}_1^{(\boldsymbol{M})}, \boldsymbol{F}_2^{(\boldsymbol{M})}, \boldsymbol{F}_3^{(\boldsymbol{M})}\right],\\
\hat{\boldsymbol{A}}_t &= \boldsymbol{\theta}_t \odot \nabla_t \odot \boldsymbol{H}_t \odot \boldsymbol{R}_t \odot \boldsymbol{U}_t.
\end{aligned}
$$

Where $\odot$ denotes matrix concatenation across the feature dimension, $\boldsymbol{\theta}_t$ are the optimizee's parameters, ∇_t is the optimizee's gradient, $\boldsymbol{H}_t$ are adafactor normalized features, and $\boldsymbol{R}_t$ are reciprocal features. Note that $\hat{\boldsymbol{A}}_t \in \mathbb{R}^{|\boldsymbol{\theta}|\times 28}$. The features within a parameter tensor are now normalized by their RMS-norm. Let $\boldsymbol{W}^{(j)} \in \mathbb{R}^{m\times n}$ be the optimizee's j'th tensor and take $\hat{\boldsymbol{A}}^{(j)} \in \mathbb{R}^{mn\times 28}$ to be the features of this tensor at timestep t. Each feature i within $\hat{\boldsymbol{A}}^{(j)}$ is then then normalized as follows:

$$
\bar{\boldsymbol{A}}_{:,i}^{(j)} = \frac{\hat{\boldsymbol{A}}_{:,i}^{(j)}}{\sqrt{\frac{1}{mn}\sum_{h=1}^{mn}(\hat{\boldsymbol{A}}_{h,i}^{(j)})^2}}. \tag{4}
$$

Finally, the normalized features $\bar{\boldsymbol{A}}$ are concatenated with timestep embeddings from step t to form the complete input features for `small_fc_lopt`:

$$
\begin{aligned}
\boldsymbol{T}_t &= [\tanh\left(\frac{t}{x}\right) \text{ for } x \in \{1,3,10,30,100,300,1000,3000,10000,30000,100000\}],\\
\boldsymbol{A}_t &= \bar{\boldsymbol{A}}_t \odot \boldsymbol{T}_t.
\end{aligned}
$$

Concretely, in Metz et al. (2022a), `small_fc_lopt`'s architecture is a two-hidden-layer 4 hidden-dimension MLP with ReLU activations: $f_{\boldsymbol{\phi}}(\boldsymbol{A}) = \boldsymbol{W}_2(ReLU(\boldsymbol{W}_1 ReLU(\boldsymbol{W}_0\boldsymbol{A} + \boldsymbol{b}_0) + \boldsymbol{b}_1) + \boldsymbol{b}_2$. At each step, the learned optimizer maps the input features for each parameter, p, in the optimizee to a two-dimensional vector, $[d, m]$. At step t, the learned optimizer update for all parameters p is given as follows:

$$\begin{aligned} f_{\boldsymbol{\phi}}(\boldsymbol{A}_p) &= [d_p, m_p]; \\ p_t &= p_{t-1} - \lambda_1 d_p e^{(\lambda_2 m_p)}. \end{aligned} \tag{5}$$

Where $\lambda_1 = \lambda_2 = 0.001$ to bias initial steps towards being small. We will now show that the inputs to `small_fc_lopt` scales like $\Theta(1)$ as $n \to \infty$. Let's first see that any RMS-normalized quantity (e.g., the input to `small_fc_lopt`) is $\Theta(1)$, which we will subsequently use in our proof of propositions 4.1 and 4.2.

Definition A.1. Let $\boldsymbol{W} \in \mathbb{R}^{m \times n}$ be the weight matrix of a neural network. Let $\boldsymbol{v} \in \mathbb{R}^{mn}$ be a vector, whose entries are statistics of parameters in $\boldsymbol{W}$. We call

$$\bar{\boldsymbol{v}} = \frac{\boldsymbol{v}}{\text{RMS}(\boldsymbol{v})} \;\; ; \;\; \text{RMS}(\boldsymbol{v}) = \sqrt{\frac{1}{mn}}\|\boldsymbol{v}\|_2. \tag{6}$$

The RMS-normalized Zhang & Sennrich (2019) version of $\boldsymbol{v}$.

Proposition A.2. *Let $\boldsymbol{v} \in \mathbb{R}^{mn}$ be a vector whose entries scale like $\Theta(f(n))$, where $f : \mathbb{R} \to \mathbb{R}$ is a continuous function. Then, the entries of the RMS-normalized counterpart of $\boldsymbol{v}$, $\bar{\boldsymbol{v}} \in \mathbb{R}^{mn}$ will scale like $\Theta(1)$.*

Proof. Let $\boldsymbol{v} \in \mathbb{R}^{mn}$ be a vector and $\bar{\boldsymbol{v}} \in \mathbb{R}^{mn}$ denote its RMS-normalized counterpart. Then,

$$\bar{\boldsymbol{v}} = \frac{\boldsymbol{v}}{\sqrt{\frac{1}{mn}\sum_{h=1}^{mn} \boldsymbol{v}_h^2}} \tag{7}$$

where the division is elementwise. From the definition of Θ, we know there exist constants $c_1, c_2 > 0$ and $N \in \mathbb{N}$ such that for all $n \geq N$ and every $h \in \{1, \ldots, mn\}$,

$$c_1\,|f(n)| \;\leq\; |v_h| \;\leq\; c_2\,|f(n)|.$$

Thus we have:

$$\begin{aligned} v_h^2 \;&\in\; \big[c_1^2\,f(n)^2,\; c_2^2\,f(n)^2\big], \\ \sum_{h=1}^{mn} v_h^2 \;&\in\; \big[mn\,c_1^2\,f(n)^2,\; mn\,c_2^2\,f(n)^2\big]. \\ \frac{1}{mn}\sum_{h=1}^{mn} v_h^2 \;&\in\; \big[c_1^2\,f(n)^2,\; c_2^2\,f(n)^2\big], \\ \sqrt{\frac{1}{mn}\sum_{h=1}^{mn} v_h^2} \;&\in\; \big[c_1\,|f(n)|,\; c_2\,|f(n)|\big] \;=\; \Theta\big(f(n)\big). \end{aligned} \tag{8}$$

Since both numerator and denominator of 7 are $\Theta(f(n))$, their ratio is $\bar{v}_h = \Theta(1)$ for each h. This completes the proof. □

Corollary A.3. *Assuming that time features are independent of width n, the coordinates of the input features to `small_fc_lopt`, as defined above, are $\Theta(1)$ as $n \to \infty$.*

Proof. This follows directly from proposition A.2 since all non-time features in `small_fc_lopt` are RMS-normalized. □

A.1.2 THE VELO ARCHITECTURE

VeLO uses an LSTM hypernetwork to produce the parameters, $\boldsymbol{\phi_W}$, of a `small_fc_lopt` optimizer for each weight matrix $\boldsymbol{W}$ in the optmizee network. Therefore, VeLO has the same accumulators as `small_fc_lopt`. VeLO's LSTM also outputs a learning rate multiplier, $\alpha_{\boldsymbol{W}}$. For a parameter p of $\boldsymbol{W}$, the update becomes:

$$\begin{aligned} f_{\boldsymbol{\phi_W}}(\boldsymbol{A}_p^*) &= [d_p, m_p]; \\ p_t &= p_{t-1} - \alpha_{\boldsymbol{W}} \lambda_1 d_p e^{(\lambda_2 m_p)}. \end{aligned} \tag{9}$$

Where $\boldsymbol{A}_p^*$ is a slightly modified version of the features outlined in the previous section (see Tab. 3 for details), crucially, the features $\boldsymbol{A}_p^*$ are all RMS-normalized as illustrated in the previous section.

To produce $\phi_{\boldsymbol{W}}$ and $\alpha_{\boldsymbol{W}}$, VeLO's LSTM takes as input 9 remaining time features ($\boldsymbol{T}$), 9 EMA loss features ($\boldsymbol{L}$), a one-hot vector representing the tensor's rank, three momentum features ($\texttt{var\_mom}_k$ for $k \in \{1, 2, 3\}$), and two variance features (`mean_rms` `var_rms`). For our goal of understanding valid parameterizations for VeLO, the most important LSTM features are the variance and momentum features as they are the only features that require further analysis of width scaling:

$$\begin{aligned} \hat{m}_k &= \frac{1}{mn} \sum_i^m \sum_j^n \frac{\boldsymbol{M}_{i,j}^{(k)}}{\mathrm{RMS}(\boldsymbol{W})}, \\ \texttt{var\_mom}_k &= c_1 \ \mathrm{clip}\Big(\log\Big[\tfrac{c_2}{mn} \sum_{i,j} \Big(\tfrac{\boldsymbol{M}_{ij}^{(k)}}{\mathrm{RMS}(\boldsymbol{W})} - \hat{m}_k\Big)^2\Big], -\tau, \tau\Big), \\ \texttt{mean\_rms} &= c_1 \ \mathrm{clip}\Big(\log\Big[\tfrac{c_2}{mn} \sum_{i,j} \tfrac{\boldsymbol{V}_{ij}}{\mathrm{RMS}(\boldsymbol{W})}\Big], -\tau, \tau\Big), \text{ and} \\ \texttt{var\_rms}_k &= c_1 \ \mathrm{clip}\Big(\log\Big[\tfrac{c_2}{mn} \sum_{i,j} \Big(\tfrac{\boldsymbol{V}_{ij}}{\mathrm{RMS}(\boldsymbol{W})} - \hat{m}_k\Big)^2\Big], -\tau, \tau\Big). \end{aligned}$$

Where we set $c_1 = \frac{1}{2}, c_2 = 10$, and $\tau = 5$ following Metz et al. (2022b). Note that, in general, the quantities calculated within the log may not be nicely bounded, but since these features are clipped, straightforward analysis shows these features are $\Theta(1)$.

Proposition A.4. *Let $\boldsymbol{W} \in \mathbb{R}^{m \times n}$ be a weight matrix whose entries scale as $\Theta(n^p)$. Let $\hat{m}_k$, $\texttt{var\_mom}_k$, `mean_rms`, and $\texttt{var\_rms}_k$ be defined as above. Assume $\boldsymbol{M}^{(k)}$ has the same per-entry scaling as $\boldsymbol{W}$, and $\boldsymbol{V}$ has entries scaling as $\Theta(n^{2p})$. Then each of $\hat{m}_k$, $\texttt{var\_mom}_k$, `mean_rms`, and $\texttt{var\_rms}_k$ is $\Theta(1)$ as $n \to \infty$.*

Proof. First, observe that

$$\mathrm{RMS}(\boldsymbol{W}) = \sqrt{\frac{1}{mn} \sum_{i,j} \boldsymbol{W}_{i,j}^2} = \sqrt{\Theta(n^{2p})} = \Theta(n^p).$$

Since $\boldsymbol{M}_{i,j}^{(k)} = \Theta(n^p)$, it follows that

$$\frac{\boldsymbol{M}_{i,j}^{(k)}}{\mathrm{RMS}(\boldsymbol{W})} = \Theta\big(n^p / n^p\big) = \Theta(1).$$

Hence

$$\hat{m}_k = \frac{1}{mn} \sum_{i,j} \Theta(1) = \Theta(1).$$

Next, consider the argument of the logarithm in $\texttt{var\_mom}_k$:

$$\frac{c_2}{mn} \sum_{i,j} \Big(\frac{\boldsymbol{M}_{i,j}^{(k)}}{\mathrm{RMS}(\boldsymbol{W})} - \hat{m}_k\Big)^2.$$

Each term $\frac{\boldsymbol{M}_{i,j}^{(k)}}{\mathrm{RMS}(\boldsymbol{W})} - \hat{m}_k$ is the difference of two $\Theta(1)$ quantities, hence $\Theta(1)$. Summing mn such terms and dividing by mn yields $\Theta(1)$. Thus

$$\log\Big[\tfrac{c_2}{mn}\sum_{i,j}\Big(\tfrac{\boldsymbol{M}_{i,j}^{(k)}}{\mathrm{RMS}(\boldsymbol{W})} - \hat{m}_k\Big)\Big] = \Theta(1),$$

and clipping to $[-\tau, \tau]$ gives $\Theta(1)$. Multiplying by the constant c_1 preserves $\Theta(1)$. Therefore $\texttt{var\_mom}_k = \Theta(1)$.

For $\texttt{mean\_rms}$, note $V_{i,j} = \Theta(n^{2p})$, so

$$\frac{\boldsymbol{V}_{i,j}}{\mathrm{RMS}(\boldsymbol{W})} = \Theta\big(n^{2p}/n^p\big) = \Theta(n^p).$$

Hence

$$\frac{c_2}{mn}\sum_{i,j}\frac{\boldsymbol{V}_{i,j}}{\mathrm{RMS}(\boldsymbol{W})} = \Theta(n^p),$$

and

$$\log\big[\Theta(n^p)\big] = \Theta(\log n).$$

Clipping $\log(n^p)$ to $[-\tau, \tau]$ yields a bounded constant $\Theta(1)$, and multiplication by c_1 gives $\texttt{mean\_rms} = \Theta(1)$.

Finally, for $\texttt{var\_rms}_k$, we have

$$\frac{\boldsymbol{V}_{i,j}}{\mathrm{RMS}(\boldsymbol{W})} - \hat{m}_k = \Theta(n^p) - \Theta(1) = \Theta(n^p),$$

so

$$\Big(\tfrac{\boldsymbol{V}_{i,j}}{\mathrm{RMS}(\boldsymbol{W})} - \hat{m}_k\Big)^2 = \Theta(n^{2p}).$$

Summing over mn entries and dividing by mn yields $\Theta(n^{2p})$. Taking the logarithm gives $\Theta(\log n)$, clipping to $[-\tau, \tau]$ yields $\Theta(1)$, and multiplying by c_1 preserves $\Theta(1)$. Hence $\texttt{var\_rms}_k = \Theta(1)$, completing the proof. □

Corollary A.5. *Assuming that time features are independent of width n, the coordinates of the input features to $\texttt{VeLO}'s$ LSTM, as defined above, are $\Theta(1)$ as $n \to \infty$.*

Proof. This follows directly from proposition A.4 since all the other input features in $\texttt{VeLO}$ trivially $\Theta(1)$ as $n \to \infty$. □

A.2 Proof: μ-parameterization for learned optimizers

For the reader's convenience, we will now restate the μP desiderata (Appendix J.2 Yang et al. (2022)) which will be used by our proof. When using a maximal update parameterization, at any point during training, the following conditions should be met:

1. **(Activation Scale)** Every (pre-)activation vector $x \in \mathbb{R}^n$ in the network should have $\Theta(1)$-sized coordinates.
2. **(Output Scale)** The output of the neural network $f_\theta(x)$ should be $O(1)$.
3. **(Maximal Updates)** All parameters should be updated as much as possible without divergence. In particular, updates should scale in width so that each parameter has nontrivial dynamics in the infinite-width limit.

While we do not go into the level of mathematical detail of Yang & Littwin (2023), our intention in propositions 4.1 and 4.2 is to show that the above desiderata are satisfied in practice by the two popular learned optimizer architectures we study.

Proposition 4.1. *Assume that the Learned Optimizer f_ϕ has the form $small\_fc\_lopt$ is fed with features given in Appendix A.1.1 and that during training the optimizee's parameters and input data*

become aligned leading to Law of Large Numbers (LLN) scaling, then the update, initialization, and pre-activation multiplier above is sufficient to obtain a Maximal Update Parametrization.

Proposition 4.2. *Assume that ϕ in Proposition 4.1 is generated using an LSTM with the input features described in Appendix A.1.2 and that during training the optimizee's parameters and input data become aligned leading to Law of Large Numbers (LLN) scaling, then the update, initialization, and pre-activation multiplier above is sufficient to obtain a Maximal Update Parametrization.*

Proof. We will now prove both statements by arguing that, in each case, the update of f_ϕ is in $\Theta(1)$, implying that our parameterization is correct. Without loss of generality, we will assume that the optimizee network has input dimension d, hidden dimension n (width), and output dimension c. Let $\boldsymbol{W}$ be some weight matrix in the optimizee network, let the update produced by f_ϕ be $\Delta\boldsymbol{W}$ and let $\boldsymbol{A}$ be the corresponding input features such that $\Delta\boldsymbol{W} = f_\phi(\boldsymbol{A})$.

- In the case of `small_fc_lopt`, $f_\phi(\boldsymbol{x}) = \Theta(1)$ since its input features, $\boldsymbol{A}$, are $\Theta(1)$ due to normalization (see corollary A.3).
- In the case of VeLO, we must also show that the LSTM hypernetwork does not introduce additional dependence on the width, n. From corollary A.5 we know that the LSTM hypernetwork will produce parameters, $\phi_{\boldsymbol{W}}$, of `small_fc_lopt` and an LR multiplier, $\alpha_{\boldsymbol{W}}$ which are $\Theta(1)$ since all inputs to the LSTM are $\Theta(1)$. Therefore, $f_\phi(\boldsymbol{x}) = \Theta(1)$ for VeLO aswell.

This fact is henceforth referred to as property (A). We will assume that the optimizee network follows our proposed μ-parameterization from Sec. 4, and show that we satisfy the desiderata of μP (outlined above) for any weight layer, $\boldsymbol{W}$, in the network. Concretely, we will show that for input and hidden layers,

$$\boldsymbol{x}_i = \Theta(1) \Rightarrow (\boldsymbol{W}\boldsymbol{x})_i = \Theta(1) \text{ and } ((\boldsymbol{W} + \Delta\boldsymbol{W})\boldsymbol{x})_i = \Theta(1) \tag{10}$$

that for the output layer

$$\boldsymbol{x}_i = \Theta(1) \Rightarrow (\boldsymbol{W}\boldsymbol{x})_i = O(1) \text{ and } ((\boldsymbol{W} + \Delta\boldsymbol{W})\boldsymbol{x})_i = O(1) \tag{11}$$

and that for all layers

$$(\Delta\boldsymbol{W}\boldsymbol{x})_i = \Theta(1). \tag{12}$$

Statements 10, 11, and 12 correspond to desiderata (1) activation scale, (2) output scale, and (3) maximal updates, respectively. In satisfying these statements, we will show that our parameterization is indeed a maximal update parameterization.

Output weights. Here, the input $\boldsymbol{x}$ has $\Theta(1)$ coordinates, we initialize the output matrix $\boldsymbol{W}$ with entries of variance 1 (which is necessary) and rescale the logits with $1/n$. Therefore, the output, $(1/n)\boldsymbol{W}\boldsymbol{x}$, is $O(1)$ by the LLN (**Output Scale Property**). From property (A), we know that $\Delta\boldsymbol{W} = f_\phi(\nabla\boldsymbol{W})$ has coordinates in $\Theta(1)$, so the entries of $\boldsymbol{W} + \Delta\boldsymbol{W}$ still have variance 1 and $\frac{1}{n}((\boldsymbol{W} + \Delta\boldsymbol{W})\boldsymbol{x})_i$ is $O(1)$. Moreover, $\frac{1}{n}(\Delta\boldsymbol{W}x)_i = \Theta(1)$ by LLN (**Maximal updates**).

Hidden weights. Since hidden weights are initialized with variance $1/n$ and $\boldsymbol{x}_i = \Theta(1)$, the coordinates of $\boldsymbol{W}\boldsymbol{x}$ are $\Theta(1)$ by LLN. From property (A), we know that $f_\phi(\boldsymbol{A}) = \Theta(1)$. Therefore, to ensure $\Delta\boldsymbol{W} \cdot \boldsymbol{x}$ is coordinate-wise bounded, we must re-scale the parameter updates:

$$\Delta\boldsymbol{W} = \frac{1}{n} f_\phi(\boldsymbol{A}).$$

Since this rescaling implies that $\Delta\boldsymbol{W}$ is $\Theta(1/n)$, the entries of $\boldsymbol{W} + \Delta\boldsymbol{W}$ still scale like $1/n$ and $((\boldsymbol{W} + \Delta\boldsymbol{W})\boldsymbol{x})_i$ is $\Theta(1)$. Moreover, since $\Delta\boldsymbol{W}$ is $\Theta(1/n)$ and $\boldsymbol{x}_i = \Theta(1)$, then $1/n(\Delta\boldsymbol{W}x)_i = \Theta(1)$ by LLN (**Maximal updates**).

Input weights. Recall that d, the input dimension, is fixed and does not grow with n. Since the input $\boldsymbol{x}_i = \Theta(1)$ and $\boldsymbol{W}$ has entries with variance $1/d$ in $\Theta(1)$, then the coordinates of pre-activation $\boldsymbol{W}x$ are $\Theta(1)$. From property (A), we know that $f_\phi(\boldsymbol{A}) = \Theta(1)$. Therefore, $\Delta\boldsymbol{W}$ is $\Theta(1)$, the entries of $\boldsymbol{W} + \Delta\boldsymbol{W}$ still have $\Theta(1)$ coordinates and $((\boldsymbol{W} + \Delta\boldsymbol{W})\boldsymbol{x})_i$ is $\Theta(1)$ (as d is fixed). Moreover, $\Delta\boldsymbol{W}x$ will have coordinate sizes that depend on the input dimension, d, but not the width. Therefore, $(\Delta Wx)_i = \Theta(1)$(**Maximal updates**). □

A.3 SUMMARY OF LEARNED OPTIMIZER INPUT FEATURES

The following section contains easy-to-read tables which report the exact learned optimizer input features for `small_fc_lopt` (Table 2) and `VeLO` (Tables 3 and 4). The tables also report the entry-wise scaling of the features before RMS-normalization and the number of features of each type. Entry-wise scaling is reported assuming a hidden weight matrix. The original implementation of these optimizers along with features calculation can be accessed here[2].

[2] `https://github.com/google/learned_optimization/blob/main/learned_optimization/learned_optimizers/adafac_mlp_lopt.py` and `https://github.com/google/learned_optimization/blob/main/learned_optimization/research/general_lopt/hyper_v2.py`

Table 2: **μP scaling for hidden layers of per-parameter features input to μLO$_M$.** All the coefficients, β_i, are learnable parameters adjusted during meta-optimization. All feature calculations and scalings are reported for a hidden weight matrix $\boldsymbol{W} \in \mathbb{R}^{m \times n}$ in an optimizee network following our proposed μ-parameterization. Here, n is the width and $m = kn$ for some constant $k \in \mathbb{R}$. In this case, the entries of the gradient of $\boldsymbol{W}$, ∇_t, scale like $\Theta(\frac{1}{n})$, where n is the width of the model. **Notation.** The table will use $\nabla_{t,i}$ or $\nabla_{t,j}$ to indicate the variable's dependence on time t and coefficient β_i or β_j, respectively. $(\nabla_{t,j})_{r,c}$ will designate indexing into row r and column c of the quantity $\nabla_{t,j}$. **DISCLAIMER: All features in our tables report scaling before the RMS-normalization.**

Type	#	Description	Accumulator Update/Equation	Scaling
Accumulators	3	Momentum accumulators with coefficients $\beta_i, i \in \{1,2,3\}$.	$\boldsymbol{M}_{t,i} = \beta_i \boldsymbol{M}_{t-1,i} + (1-\beta_i)\nabla_t$	$\Theta(\frac{1}{n})$
	1	Second moment accumulator with coefficient β_4.	$\boldsymbol{V}_t = \beta_4 \boldsymbol{V}_{t-1} + (1-\beta_4)\nabla_t^2$	$\Theta(\frac{1}{n^2})$
	3	Adafactor row accumulator with coefficients $\beta_i, i \in \{5,6,7\}$.	$\boldsymbol{r}_{t,i} = \beta_i \boldsymbol{r}_{t-1,i} + (1-\beta_i)\,\texttt{row\_mean}(\nabla_t^2)$	$\Theta(\frac{1}{n^2})$
	3	Adafactor accumulator with coefficients $\beta_i, i \in \{5,6,7\}$.	$\boldsymbol{c}_{t,i} = \beta_i \boldsymbol{c}_{t-1,i} + (1-\beta_i)\,\texttt{col\_mean}(\nabla_t^2)$	$\Theta(\frac{1}{n^2})$
Accumulator Features	3	Momentum values normalized by the square root of the second moment for $i \in \{5,6,7\}$.	$\frac{\boldsymbol{M}_{t,i}}{\sqrt{\boldsymbol{V}_t}}$	$\Theta(1)$
	1	The reciprocal square root of the second moment value.	$\frac{1}{\sqrt{\boldsymbol{V}}}$	$\Theta(n)$
	6	The reciprocal square root of the Adafactor accumulators.	$\frac{1}{\sqrt{\boldsymbol{r}_{t,i}}}$ OR $\frac{1}{\sqrt{\boldsymbol{c}_{t,i}}}$	$\Theta(n)$
	3	Adafactor gradient features for $i \in \{5,6,7\}$.	$\nabla_t \cdot \sqrt{\frac{\frac{1}{m}\sum_{h=1}^{m}(\boldsymbol{r}_{t,i})_h}{\boldsymbol{r}_{t,i}\boldsymbol{c}_{t,i}^T}}$	$\Theta(1)$
	3	Adafactor momentum features for $i,j \in \{(5,1),(6,2),(7,3)\}$.	$\boldsymbol{M}_{t,j} \cdot \sqrt{\frac{\frac{1}{m}\sum_{h=1}^{m}(\boldsymbol{r}_{t,i})_h}{\boldsymbol{r}_{t,i}\boldsymbol{c}_{t,i}^T}}$	$\Theta(1)$
Time Features	11	Time Features for $x \in \{1,3,10,30,100,300,1000,3000,10^4,3\cdot 10^4,10^5\}$.	$\tanh\left(\frac{t}{x}\right)$	$\Theta(1)$
Parameters	1	Parameter value.	$\boldsymbol{W}_t$	$\Theta(\frac{1}{n})$
	1	Gradient value.	∇_t	$\Theta(\frac{1}{n})$
Total	39	–	–	–

Table 3: **μP scaling of per-parameter features input to the per-parameter network of μVeLO$_M$.** All feature calculations and scalings are reported for a hidden weight matrix $\boldsymbol{W} \in \mathbb{R}^{m \times n}$ in an optimizee network following our proposed μ-parameterization. Here, n is the width and $m = kn$ for some constant $k \in \mathbb{R}$. In this case, the entries of the gradient of $\boldsymbol{W}$, ∇_t, scale like $\Theta(\frac{1}{n})$, where n is the width of the model. **Notation.** The table will use $\nabla_{t,i}$ or $\nabla_{t,j}$ to indicate the variable's dependence on time t and coefficient β_i or β_j, respectively. $(\nabla_{t,j})_{r,c}$ will designate indexing into row r and column c of the quantity $\nabla_{t,j}$. **DISCLAIMER: All features in our tables report scaling before the RMS-normalization.**

Type	#	Description	Accumulator Update/Equation	Scaling
Accumulators	3	Momentum accumulators with coefficients $\beta_i, i \in \{1, 2, 3\}$.	$\boldsymbol{M}_{t,i} = \beta_i \boldsymbol{M}_{t-1,i} + (1 - \beta_i)\nabla_t$	$\Theta(\frac{1}{n})$
	1	Second moment accumulator with coefficient β_4.	$\boldsymbol{V}_t = \beta_4 \boldsymbol{V}_{t-1} + (1 - \beta_4)\nabla_t^2$	$\Theta(\frac{1}{n^2})$
	3	Adafactor row accumulator with coefficients $\beta_i, i \in \{5, 6, 7\}$.	$\boldsymbol{r}_{t,i} = \beta_i \boldsymbol{r}_{t-1,i} + (1 - \beta_i)\ \texttt{row\_mean}(\nabla_t^2)$	$\Theta(\frac{1}{n^2})$
	3	Adafactor accumulator with coefficients $\beta_i, i \in \{5, 6, 7\}$.	$\boldsymbol{c}_{t,i} = \beta_i \boldsymbol{c}_{t-1,i} + (1 - \beta_i)\ \texttt{col\_mean}(\nabla_t^2)$	$\Theta(\frac{1}{n^2})$
Accumulator Features	3	Momentum values normalized by the square root of the second moment for $i \in \{5, 6, 7\}$.	$\frac{\boldsymbol{M}_{t,i}}{\sqrt{\boldsymbol{V}_t}}$	$\Theta(1)$
	1	The reciprocal square root of the second moment value.	$\frac{1}{\sqrt{\boldsymbol{V}}}$	$\Theta(n)$
	6	The reciprocal square root of the Adafactor accumulators.	$\frac{1}{\sqrt{\boldsymbol{r}_{t,i}}}$ OR $\frac{1}{\sqrt{\boldsymbol{c}_{t,i}}}$	$\Theta(n)$
	3	Adafactor gradient features for $i \in \{5, 6, 7\}$.	$\nabla_t \cdot \sqrt{\frac{\frac{1}{m}\sum_{h=1}^{m}(\boldsymbol{r}_{t,i})_h}{\boldsymbol{r}_{t,i}\boldsymbol{c}_{t,i}^T}}$	$\Theta(1)$
	3	Adafactor momentum features for $i, j \in \{(5, 1), (6, 2), (7, 3)\}$.	$\boldsymbol{M}_{t,j} \cdot \sqrt{\frac{\frac{1}{m}\sum_{h=1}^{m}(\boldsymbol{r}_{t,i})_h}{\boldsymbol{r}_{t,i}\boldsymbol{c}_{t,i}^T}}$	$\Theta(1)$
Parameters	1	Parameter value.	$\boldsymbol{W}_t$	$\Theta(\frac{1}{n})$
	1	Gradient value.	∇_t	$\Theta(\frac{1}{n})$
	1	Gradient value.	$\texttt{clip}(\nabla_t, -0.1, 0.1)$	$\Theta(\frac{1}{n})$
Total	29	–	–	–

Table 4: **Per-tensor features used as input to VeLO's LSTM.** All feature calculations and scalings are reported for a hidden weight matrix $\boldsymbol{W} \in \mathbb{R}^{m \times n}$ in an optimizee network following our proposed μ-parameterization. Here, n is the width and $m = kn$ for some constant $k \in \mathbb{R}$. In this case, the entries of the gradient of $\boldsymbol{W}$, ∇_t, scale like $\Theta(\frac{1}{n})$, where n is the width of the model.

Type	#	Description	Equation	Scaling
Accumulator Features	3	Variance across coordinates of the 3 momentum accumulator matrices normalized by the RMS of the current parameter values $i \in \{1, 2, 3\}$	$\texttt{var\_mom}_i$ (Sec. A.1.2)	$\Theta(1)$
	1	Mean across coordinates of variance accumulator normalized by the parameter RMS	$\texttt{mean\_rms}$ (Sec. A.1.2)	$\Theta(1)$
	3	Coordinate-wise mean of the variance accumulator. $i \in \{1, 2, 3\}$	$\texttt{var\_rms}_i$(Sec. A.1.2)	$\Theta(1)$
Tensor Rank	5	A one hot vector representing the tensor's rank, r.	e_r	$\Theta(1)$
EMA Loss Features	9	EMAs of the loss at different timescales chosen based on the number of steps. Values are normalized by the max and min losses seen so far.	see Metz et al. (2022b)	$\Theta(1)$
Remaining Time Features	9	Time Features for $x \in \{0.03, 0.1, 0.2, 0.4, 0.6, 0.8, 0.9, 1.0, 1.1\}$.	$\tanh(t/T - 10x)$	$\Theta(1)$
Total	30	–	–	–

Table 5: **Meta-training and hyperparameter configurations of LOs and baselines in our empirical evaluation.** The small_fc_lopt and VeLO architectures were initially proposed in (Metz et al., 2022a) and (Metz et al., 2022b). See Tab. 10 for a list of all tasks used in this work.

Identifier	Type	Architecture	Optimizee Par.	Meta-Training / Tuning Task(s)
μLO$_S$	Ours	small_fc_lopt	μLO Sec. 4	IN32$\mathcal{T}^{\text{MLP}}_{(3,128)}$
μLO$_M$	Ours	small_fc_lopt	μLO Sec. 4	IN32$\mathcal{T}^{\text{MLP}}_{(3,128)}$,IN32$\mathcal{T}^{\text{MLP}}_{(3,512)}$,IN32$\mathcal{T}^{\text{MLP}}_{(3,1024)}$
μVeLO$_M$	Ours	VeLO	μLO Sec. 4	IN32$\mathcal{T}^{\text{MLP}}_{(3,128)}$,IN32$\mathcal{T}^{\text{MLP}}_{(3,512)}$,IN32$\mathcal{T}^{\text{MLP}}_{(3,1024)}$
LO$_S$	LO Baseline	small_fc_lopt	SP	IN32$\mathcal{T}^{\text{MLP}}_{(3,128)}$
LO$_M$	LO Baseline	small_fc_lopt	SP	IN32$\mathcal{T}^{\text{MLP}}_{(3,128)}$,IN32$\mathcal{T}^{\text{MLP}}_{(3,512)}$,IN32$\mathcal{T}^{\text{MLP}}_{(3,1024)}$
VeLO$_M$	LO Baseline	VeLO	SP	IN32$\mathcal{T}^{\text{MLP}}_{(3,128)}$,IN32$\mathcal{T}^{\text{MLP}}_{(3,512)}$,IN32$\mathcal{T}^{\text{MLP}}_{(3,1024)}$
VeLO-4000	Oracle LO Baseline	VeLO	SP	See Metz et al. (2022b) (Appendix C.2)
μAdam	Baseline	–	μP Adam	per-task tuning (see Tab. 7)
AdamW	Baseline	–	SP	per-task tuning (see Tab. 9)

B Hand Designed Optimizer Hyperparameter Tuning

To provide strong baselines for our study, we tune the hyperparameters of AdamW and μAdam for more than 500 trials on one instance of each task in our evaluation suite. Since the largest width task seen by μLO$_M$ and μVeLO$_M$ is 1024, we select this width for all our hyperparameter sweeps. Similarly, we use the same depth=3 and training steps=1000 as for the meta-training of μLO$_M$ and μVeLO$_M$.

B.1 Tuning μAdam

We tune μAdam's learning rate (η) and accumulator coefficients (β_1, and β_2). Table 6 reports all hyperparameter values that we swept for each task. Table 7 reports the best-performing hyperparameter values found by selecting the values that achieved the lowest final smoothed training loss on each task.

Table 6: **Hyperparameter sweep values for μAdam.**

Hyperparameter	#	Values
η	32	$\{10^{-6}, 1.56\times10^{-6}, 2.44\times10^{-6}, 3.81\times10^{-6}, 5.95\times10^{-6}, 9.28\times10^{-6},$ $1.45\times10^{-5}, 2.26\times10^{-5}, 3.53\times10^{-5}, 5.52\times10^{-5}, 8.62\times10^{-5},$ $1.35\times10^{-4}, 2.10\times10^{-4}, 3.28\times10^{-4}, 5.12\times10^{-4}, 8.00\times10^{-4},$ $1.25\times10^{-3}, 1.95\times10^{-3}, 3.05\times10^{-3}, 4.76\times10^{-3}, 7.43\times10^{-3},$ $1.16\times10^{-2}, 1.81\times10^{-2}, 2.83\times10^{-2}, 4.42\times10^{-2}, 6.90\times10^{-2},$ $1.08\times10^{-1}, 1.68\times10^{-1}, 2.63\times10^{-1}, 4.10\times10^{-1}, 6.40\times10^{-1}, 1\}$
β_1	4	$\{0.85, 0.9, 0.95, 0.99\}$
β_2	4	$\{0.9, 0.95, 0.99, 0.999\}$
Total	512	–

B.2 Tuning AdamW

We tune AdamW's learning rate (η), accumulator coefficients (β_1, and β_2), and weight decay (λ). Table 8 reports all hyperparameter values that we swept for each task. Table 9 reports the best-performing hyperparameter values found by selecting the values that achieved the lowest final smoothed training loss on each task.

Table 7: **Strongest performing hyperparameter values of μAdam for each task, with and without a schedule.** All optimizers with a schedule use a linear warmup and cosine decay schedule with the minimum learning rate set to 0.1η.

Task	η	β_1	β_2	GPU Hours
$\mathcal{T}^{\text{LM}}_{(3,1024)}$	0.1077	0.85	0.999	48
$\mathcal{T}^{\text{ViT}}_{(3,1024)}$	0.044173	0.85	0.999	17
IN32$\mathcal{T}^{\text{MLP}}_{(3,1024)}$	0.044173	0.85	0.999	9
IN64$\mathcal{T}^{\text{MLP}}_{(3,1024)}$	0.028289	0.85	0.99	19
C10$\mathcal{T}^{\text{MLP}}_{(3,1024)}$	0.1473	0.9	0.95	4
Total	–	–	–	97

Table 8: **Hyperparameter sweep values for AdamW.**

Hyperparameter	**#**	**Values**
η	14	$\{0.1,\ 4.92 \times 10^{-2},\ 2.42 \times 10^{-2},\ 1.19 \times 10^{-2},$ $5.88 \times 10^{-3}, 2.89 \times 10^{-3},\ 1.43 \times 10^{-3},$ $7.02 \times 10^{-4},\ 3.46 \times 10^{-4},\ 1.70 \times 10^{-4},$ $8.38 \times 10^{-5},\ 4.12 \times 10^{-5},\ 2.03 \times 10^{-5},\ 1.00 \times 10^{-5}\}$
β_1	3	$\{0.9,\ 0.95,\ 0.99\}$
β_2	3	$\{0.95,\ 0.99,\ 0.999\}$
λ	4	$\{0.1,\ 0.01,\ 0.001,\ 0.0001\}$
Total	504	–

Table 9: **Strongest performing hyperparameter values of AdamW for each task, with and without a schedule.** All optimizers with a schedule use a linear warmup and cosine decay schedule with the minimum learning rate set to 0.1η.

Task	η	β_1	β_2	λ	GPU Hours
$\mathcal{T}^{\text{LM}}_{(3,1024)}$	7.02×10^{-4}	0.9	0.99	0.001	48
$\mathcal{T}^{\text{ViT}}_{(3,1024)}$	1.70×10^{-4}	0.9	0.999	0.01	18
IN32$\mathcal{T}^{\text{MLP}}_{(3,1024)}$	7.02×10^{-4}	0.9	0.999	0.01	9
IN64$\mathcal{T}^{\text{MLP}}_{(3,1024)}$	7.02×10^{-4}	0.9	0.99	0.001	20
C10$\mathcal{T}^{\text{MLP}}_{(3,1024)}$	2.89×10^{-3}	0.9	0.95	0.0001	4
Total	–	–	–	–	99

C META-TRAINING WITH μLOS

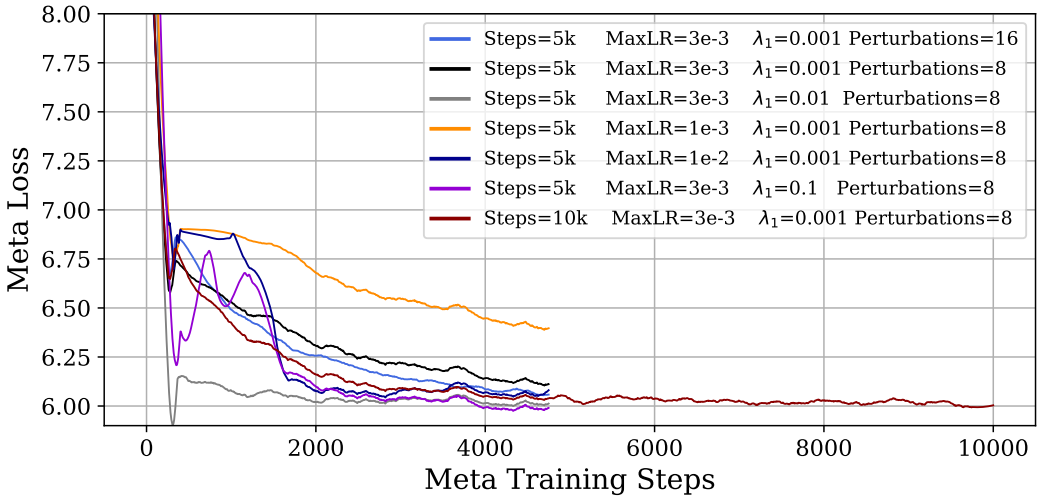


Figure 7: **Ablating Meta-training Hyperparameter for μLO$_S$.** All curves show a single meta-training run. Using AdamW with a linear warmup and cosine annealing schedule, we meta-train μLO$_S$ to train 3-layer width 128 MLPs for classifying $32 \times 32 \times 3$ ImageNet Images. By default, we warmup linearly for 100 steps to a maximum learning rate of $3e-3$ and anneal the learning rate for 4900 steps to a value of $1e-3$ with $\lambda_1 = 0.001$ (from Equation 3) and sampling 8 perturbations per step in PESVicol et al. (2021). The above ablation varies the maximum learning rate $\in \{1e-2, 3e-3, 1e-3\}$ (always using 100 steps of warmup and decaying to $0.3\times$MaxLR), $\lambda_1 \in \{0.001, 0.01, 0.1\}$, the number of steps (5k or 10k), and the number of perturbations (8 or 16). We observe that using all default values except for $\lambda_1 = 0.01$ yields one of the best solutions while being fast to train and stable during meta-training.

General meta-training setup for small_fc_lopt Each small_fc_lopt (Metz et al., 2022a) learned optimizer is meta-trained for 5000 steps of gradient descent using AdamW (Loshchilov & Hutter, 2019) and a linear warmup and cosine annealing schedule. We use PES (Vicol et al., 2021) to estimate meta-gradients with a truncation length of 50 steps and sampling 8 perturbations per task at each step with standard deviation 0.01. For the inner optimization task, we used a maximum unroll length of 1000 iterations; that is, all our learned optimizers see at most 1000 steps of the inner optimization problem during meta-training. Unlike with μAdam, we do not tune the μP multipliers when meta-training μLO$_S$ and μLO$_M$, instead, we set them all to 1. All optimizers are meta-trained on a single A6000 GPU. μLO$_S$ and LO$_S$ take 8 hours each to meta-train, while μLO$_M$ and LO$_M$ take 103 hours.

General meta-training setup for VeLO Each VeLO (Metz et al., 2022a) learned optimizer is meta-trained for 45000 steps of gradient descent using AdamW (Loshchilov & Hutter, 2019) and a linear warmup and cosine annealing schedule. We using PES (Vicol et al., 2021) to estimate meta-gradients with a truncation length of 20 steps and sampling 8 perturbations per task at each step with standard deviation 0.01. For the inner optimization task, we used a maximum unroll length of 1000 iterations; that is, all our learned optimizers see at most 1000 steps of the inner optimization problem during meta-training. Unlike Yang et al. (2022), we do not tune the μP multipliers when meta-training μLO$_S$ and μLO$_M$, instead, we set them all to 1. All optimizers are meta-trained on a single A6000 GPU. μVeLO$_M$and VeLO$_M$ each take 250 GPU-hours to meta-train.

Meta-training hyperparameters for small_fc_lopt in μP While there are very few differences between μLOs and SP LOs, the effective step size for hidden layers is changed (see eq. 3) which could alter the optimal meta-training hyperparameters. Consequently, we conduct an ablation study on hyper-parameters choices for μLO$_S$. Specifically, using AdamW and gradient clipping with a linear warmup and cosine annealing LR schedule, we meta-train μLO$_S$ to train 3-layer width 128

MLPs to classify $32 \times 32 \times 3$ ImageNet Images. By default, we warmup linearly for 100 steps to a maximum learning rate of $3e-3$ and anneal the learning rate for 4900 steps to a value of $1e-3$ with $\lambda_1 = 0.001$ (from Equation 3) and sampling 8 perturbations per step in PESVicol et al. (2021). The above ablation varies the maximum learning rate $\in \{1e-2, 3e-3, 1e-3\}$ (always using 100 steps of warmup and decaying to $0.3\times$MaxLR), $\lambda_1 \in \{0.001, 0.01, 0.1\}$, the number of steps (5k or 10k), and the number of perturbations (8 or 16). We observe that using all default values except for $\lambda_1 = 0.01$ yields one of the best solutions while being fast to train and stable during meta-training. We, therefore, select these hyperparameters to meta-train μLO_S and μLO_M.

Meta-training hyperparameters for VeLO in μP Unlike for small_fc_lopt, we do not find it necessary to change λ_1 from its default value of 0.001. However, we do slightly alter the VeLO update by removing the multiplication by the current parameter norm. This causes problems when initializing tensors to zero, as we do in our experiments.

μP at Meta-training time It is important to carefully choose meta-training tasks that can effectively be transferred to larger tasks. In (Yang et al., 2022), authors discuss these points and provide two notable guidelines: initialize the output weight matrix to zero (as it will approach zero in the limit) and use a relatively large key size when meta-training transformers. For all our tasks, we initialize the network's final layer to zeros following this guidance. While we do not meta-train on transformers, we suspect that the aforementioned transformer-specific guidelines may be useful for doing so.

D Extended Related Work

Learned optimization. While research on learned optimizers (LOs) spans several decades (Schmidhuber, 1992; Thrun & Pratt, 2012; Chen et al., 2022; Amos, 2022), our work is primarily related to the recent meta-learning approaches utilizing efficient per-parameter optimizer architectures of Metz et al. (2022a). Unlike prior work (Andrychowicz et al., 2016; Wichrowska et al., 2017; Chen et al., 2020), which computes meta-gradients (the gradients of the learned optimizer) using backpropagation, Metz et al. (2022a) use Persistent Evolutionary Strategies (PES) (Vicol et al., 2021), a truncated variant of evolutionary strategies (ES) (Buckman et al., 2018; Nesterov & Spokoiny, 2017; Parmas et al., 2018). ES improves meta-training of LOs by having more stable meta-gradient estimates compared to backpropagation through time, especially for longer sequences (i.e. long parameter update unrolls inherent in meta-training) (Metz et al., 2019). PES and most recently ES-Single (Vicol, 2023) are more efficient and accurate variants of ES, among which PES is more well-established in practice making it a favourable approach to meta-training.

Generalization in LOs. One of the critical issues in LOs is generalization in the three main aspects (Chen et al., 2022; Amos, 2022): (1) optimize novel tasks (often referred to as *meta-generalization*); (2) optimize for more iterations than the maximum unroll length used in meta-training; (3) avoid overfitting on the training set. Among these, (3) has been extensively addressed using different approaches, such as meta-training on the validation set objective (Metz et al., 2019), adding extra-regularization terms (Harrison et al., 2022), parameterizing LOs as hyperparameter controllers (Almeida et al., 2021) and introducing flatness-aware regularizations (Yang et al., 2023). The regularization terms (Harrison et al., 2022; Yang et al., 2023) often alleviate issue (2) as a byproduct. However, meta-generalization (1) has remained a more difficult problem.

One approach to tackle this problem is to meta-train LOs on thousands of tasks (Metz et al., 2022b). However, this approach is extremely expensive and does not address the issue in a principled way leading to poor meta-generalization in some cases, e.g. when the optimization task includes much larger networks. Alternatively, Premont-Schwarz et al. (2022) introduced Loss-Guarded L2O (LGL2O) that switches to Adam/SGD if the LO starts to diverge improving meta-generalization. However, this approach needs tuning Adam/SGD and requires additional computation (e.g. for loss check) limiting (or completely diminishing in some cases) the benefits of the LO. In this work, we study aspects (1) and (2) of LO generalization, demonstrating how existing SP LOs generalize poorly across these dimensions and showing how one can apply μP to learned optimizers to substantially improve generalization in both these aspects.

Maximal Update Parametrization. First proposed by Yang & Hu (2021), the Maximal Update Parametrization is the unique stable abc-Parametrization where every layer learns features. The parametrization was derived for adaptive optimizers by Yang & Littwin (2023) and was applied by Yang et al. (2022) to enable zero-shot hyperparameter transfer, constituting the first practical application of the tensor programs series of papers. Earlier works in the *tensor programs series* build the mathematical foundation that led to the discovery of μP. Yang (2019) shows that many modern neural networks with randomly initialized weights and biases are Gaussian Processes, providing a language, called Netsor, to formalize neural network computations. Yang (2020a) focuses on neural tangent kernels (NTK), proving that as a randomly initialized network's width tends to infinity, its NTK converges to a deterministic limit. Yang (2020b) shows that randomly initialized network's pre-activations become independent of its weights when its width tends to infinity. Most recently, in tensor programs VI, Yang et al. (2024) propose Depth-μP, a parameterization allowing for hyperparameter transfer in infinitely deep networks. However, Depth-μP is only valid for residual networks with a block depth of 1, making it unusable for most practical architectures (e.g., transformers, resnets, etc.). For these reasons, we do not study Depth-μP herein. Building on the latest works studying width μP (Yang & Littwin, 2023; Yang et al., 2022), in this work, we show that μP can be extended to the case of learned optimizers and empirically evaluate its benefits in this setting.

E List of Meta-testing Tasks

Table 10 reports the configuration of different testing tasks used to evaluate our optimizers. We note that we do not augment the ImageNet datasets we use in any way except for normalizing the images. We tokenize LM1B using a sentence piece tokenizer(Kudo & Richardson, 2018) with 32k vocabulary size. All evaluation tasks are run on A6000 48GB or A100 80GB GPUs for 5 random seeds.

F Additional Experiments

F.1 Comparison with VeLO-4000

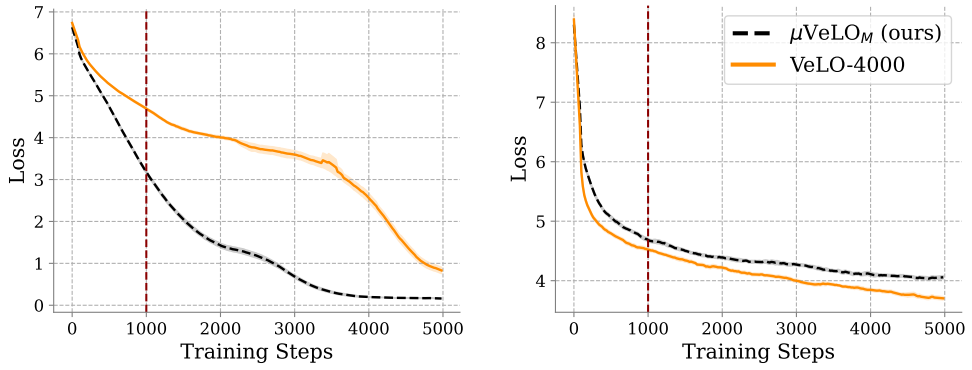
Pre-trained VeLO (VeLO-4000). VeLO (Metz et al., 2022b) is a learned optimizer that was meta-trained on a curriculum of progressively more expensive meta-training tasks for a total of 4000 TPU months. These tasks include but are not limited to image classification with MLPs, ViTs, ConvNets, and ResNets; compression with MLP auto-encoders; generative modeling with VAEs; and language modeling with transformers and recurrent neural networks. During meta-training, VeLO-4000 unrolls inner problems for up to 20k steps ($20\times$ ours); the final model was then fine-tuned on tasks with up to 200k steps of optimization. VeLO-4000, therefore represents a strong but unfair baseline as it is trained on far more data and with far more compute than our main VeLO experiments.

Is VeLO-4000 a fair baseline? While we believe the comparison is interesting given the relevance of our results to scaling up LOs, the comparison will unfairly advantage VeLO-4000 as **all tasks in our suite fall within its meta-training distribution** and VeLO-4000 was meta-trained on inner unroll horizons well beyond those we evaluate. Thus, when comparing our LOs to VeLO-4000, it is important to keep in mind that it is an unfair baseline since our learned optimizers meta-trained with only 0.004% of VeLO-4000's compute budget. We included a compute-matched fair baseline, VeLO$_M$ in the main manuscript.

Comparison Figures 8 reports the training curves of different optimizers, including VeLO-4000, on width 8192 and 3072 MLP and transformer language model tasks, respectively. We observe that μLO$_M$ and μVeLO$_M$ (trained with many orders of magnitude less compute) outperforms VeLO-4000 at this large width on the in-distribution tasks, but fall short despite still generalizing well when evaluated far out-of-distribution on a width 3072 language modeling task. We hypothesize that this is likely due to the task being nearly in-distribution for VeLO-4000 meta-training data while being OOD w.r.t. architecture, width, and training steps for μLO$_M$ and μVeLO$_M$. These results overall suggest that μVeLO$_M$ may be more scalable than its non-μP counterpart, particularly in the large model cases where VeLO-4000 struggled (Metz et al., 2022b).

Table 10: **Meta-testing settings.** We report the optimization tasks we will use to evaluate the LOs of Table 5.

Identifier	Dataset	Model	Depth	Width	Attn. Heads	FFN Size	Batch Size	Sequence Length
IN32$\mathcal{T}^{\text{MLP}}_{(3,128)}$	$32 \times 32 \times 3$ ImageNet	MLP	3	128	–	–	4096	–
IN32$\mathcal{T}^{\text{MLP}}_{(3,256)}$	$32 \times 32 \times 3$ ImageNet	MLP	3	256	–	–	4096	–
IN32$\mathcal{T}^{\text{MLP}}_{(3,512)}$	$32 \times 32 \times 3$ ImageNet	MLP	3	512	–	–	4096	–
IN32$\mathcal{T}^{\text{MLP}}_{(3,1024)}$	$32 \times 32 \times 3$ ImageNet	MLP	3	1024	–	–	4096	–
IN32$\mathcal{T}^{\text{MLP}}_{(3,2048)}$	$32 \times 32 \times 3$ ImageNet	MLP	3	2048	–	–	4096	–
IN32$\mathcal{T}^{\text{MLP}}_{(3,4096)}$	$32 \times 32 \times 3$ ImageNet	MLP	3	4096	–	–	4096	–
IN32$\mathcal{T}^{\text{MLP}}_{(3,8192)}$	$32 \times 32 \times 3$ ImageNet	MLP	3	8192	–	–	4096	–
IN64$\mathcal{T}^{\text{MLP}}_{(3,128)}$	$64 \times 64 \times 3$ ImageNet	MLP	3	128	–	–	4096	–
IN64$\mathcal{T}^{\text{MLP}}_{(3,256)}$	$64 \times 64 \times 3$ ImageNet	MLP	3	256	–	–	4096	–
IN64$\mathcal{T}^{\text{MLP}}_{(3,512)}$	$64 \times 64 \times 3$ ImageNet	MLP	3	512	–	–	4096	–
IN64$\mathcal{T}^{\text{MLP}}_{(3,1024)}$	$64 \times 64 \times 3$ ImageNet	MLP	3	1024	–	–	4096	–
IN64$\mathcal{T}^{\text{MLP}}_{(3,2048)}$	$64 \times 64 \times 3$ ImageNet	MLP	3	2048	–	–	4096	–
IN64$\mathcal{T}^{\text{MLP}}_{(3,4096)}$	$64 \times 64 \times 3$ ImageNet	MLP	3	4096	–	–	4096	–
C10$\mathcal{T}^{\text{MLP}}_{(3,128)}$	$32 \times 32 \times 3$ Cifar-10	MLP	3	128	–	–	4096	–
C10$\mathcal{T}^{\text{MLP}}_{(3,256)}$	$32 \times 32 \times 3$ Cifar-10	MLP	3	256	–	–	4096	–
C10$\mathcal{T}^{\text{MLP}}_{(3,512)}$	$32 \times 32 \times 3$ Cifar-10	MLP	3	512	–	–	4096	–
$\mathcal{T}^{\text{LM}}_{(3,1024)}$	$32 \times 32 \times 3$ Cifar-10	MLP	3	1024	–	–	4096	–
C10$\mathcal{T}^{\text{MLP}}_{(3,2048)}$	$32 \times 32 \times 3$ Cifar-10	MLP	3	2048	–	–	4096	–
C10$\mathcal{T}^{\text{MLP}}_{(3,4096)}$	$32 \times 32 \times 3$ Cifar-10	MLP	3	4096	–	–	4096	–
C10$\mathcal{T}^{\text{MLP}}_{(3,8192)}$	$32 \times 32 \times 3$ Cifar-10	MLP	3	8192	–	–	4096	–
$\mathcal{T}^{\text{ViT}}_{(3,192)}$	$32 \times 32 \times 3$ ImageNet	ViT	3	192	3	768	1024	–
$\mathcal{T}^{\text{ViT}}_{(3,384)}$	$32 \times 32 \times 3$ ImageNet	ViT	3	384	6	1536	1024	–
$\mathcal{T}^{\text{ViT}}_{(3,768)}$	$32 \times 32 \times 3$ ImageNet	ViT	3	768	8	3072	1024	–
$\mathcal{T}^{\text{ViT}}_{(3,1024)}$	$32 \times 32 \times 3$ ImageNet	ViT	3	1024	8	4096	1024	–
$\mathcal{T}^{\text{ViT}}_{(3,2048)}$	$32 \times 32 \times 3$ ImageNet	ViT	3	2048	16	8192	1024	–
$\mathcal{T}^{\text{ViT}}_{(3,3072)}$	$32 \times 32 \times 3$ ImageNet	ViT	3	3072	16	12288	1024	–
$\mathcal{T}^{\text{LM}}_{(3,192)}$	LM1B, 32k Vocab	Transformer LM	3	192	3	768	128	64
$\mathcal{T}^{\text{LM}}_{(3,384)}$	LM1B, 32k Vocab	Transformer LM	3	384	6	1536	128	64
$\mathcal{T}^{\text{LM}}_{(3,768)}$	LM1B, 32k Vocab	Transformer LM	3	768	8	3072	128	64
$\mathcal{T}^{\text{LM}}_{(3,1024)}$	LM1B, 32k Vocab	Transformer LM	3	1024	8	4096	128	64
$\mathcal{T}^{\text{LM}}_{(3,2048)}$	LM1B, 32k Vocab	Transformer LM	3	2048	16	8192	128	64
$\mathcal{T}^{\text{LM}}_{(3,3072)}$	LM1B, 32k Vocab	Transformer LM	3	3072	16	12288	128	64
$\mathcal{DT}^{\text{MLP}}_{(16,1024)}$	32×32 ImageNet	MLP	16	1024	–	–	128	–
$\mathcal{DT}^{\text{ViT}}_{(16,1024)}$	32×32 ImageNet	ViT	16	1024	3	4096	128	–
$\mathcal{DT}^{\text{LM}}_{(16,1024)}$	LM1B	Transformer LM	16	1024	3	4096	128	–

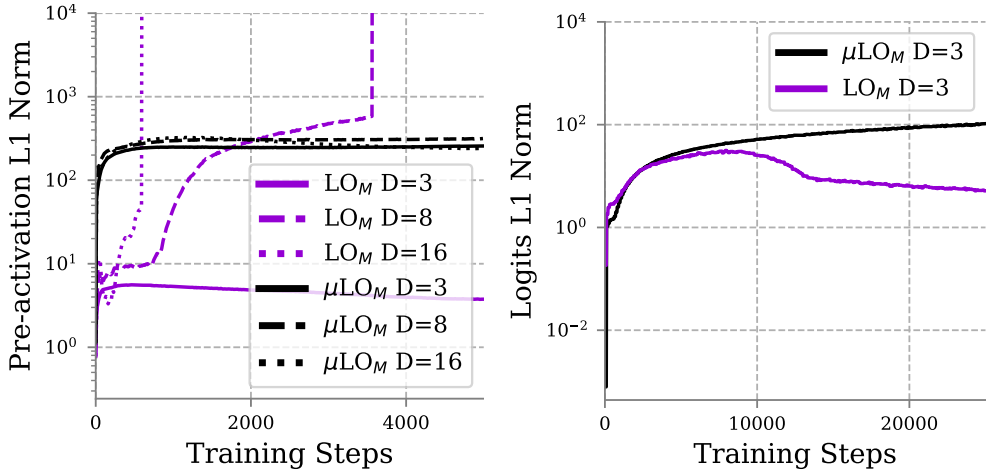


(a) MLP IN32 W=8192 (b) LM W=4096

Figure 8: **A comparison to VeLO-4000 on the widest tasks.** All optimizers except VeLO are meta-trained or hyperparameter tuned for 1000 inner steps (dotted red line), therefore, any optimization beyond 1000 steps is considered out-of-distribution. We plot average training loss over 5 seeds with standard error bars. We observe that μLO$_M$ and μVeLO$_M$ outperform VeLO on the widest in-distribution tasks, but fall short, despite still generalizing well when evaluated far out-of-distribution on a width 3072 language modeling task.

F.1.1 WHY DO μLOS IMPROVE GENERALIZATION TO DEPTH AND LONGER TRAINING HORIZONS?

While our goal was to improve the meta-generalization of learned optimizers to unseen wider tasks, in sections 5.2.4 and 5.2.4, we also observed improved meta-generalization to deeper and wider networks. This discovery is entirely empirical as we did not use a parameterization that has depth transfer properties (e.g. μDepth (Yang et al., 2024)). With Figure 9 as evidence, we hypothesize that the reason for improved transfer to deeper models and longer training is μLOs' ability to maintain stable logits in the optimizee throughout training in contrast to SP LOs. For instance, in subfigure (a), we observe that the first layer pre-activations of depth 8 and depth 16 MLPs trained with LO_M grow rapidly at the beginning of training, while those of deeper MLPs optimized by μLO_M vary similarly to the depth-3 MLP (same depth as meta-training). In subfigure (b), we observe a similar but less drastic change in logit L1 norm as training progresses. While the L1 norm of the MLP trained by μLO_M consistently grows at a stable rate throughout training, for LO_M the MLP's logits undergo a change in slope after 8000 steps of training and a near discontinuity at 13000 steps. With the evidence we have so far, it is not possible to be certain whether the observed activation stability is the cause of the improved generalization or merely a symptom of it. That being said, these results can still help inform on favorable properties for the generalization of LOs.

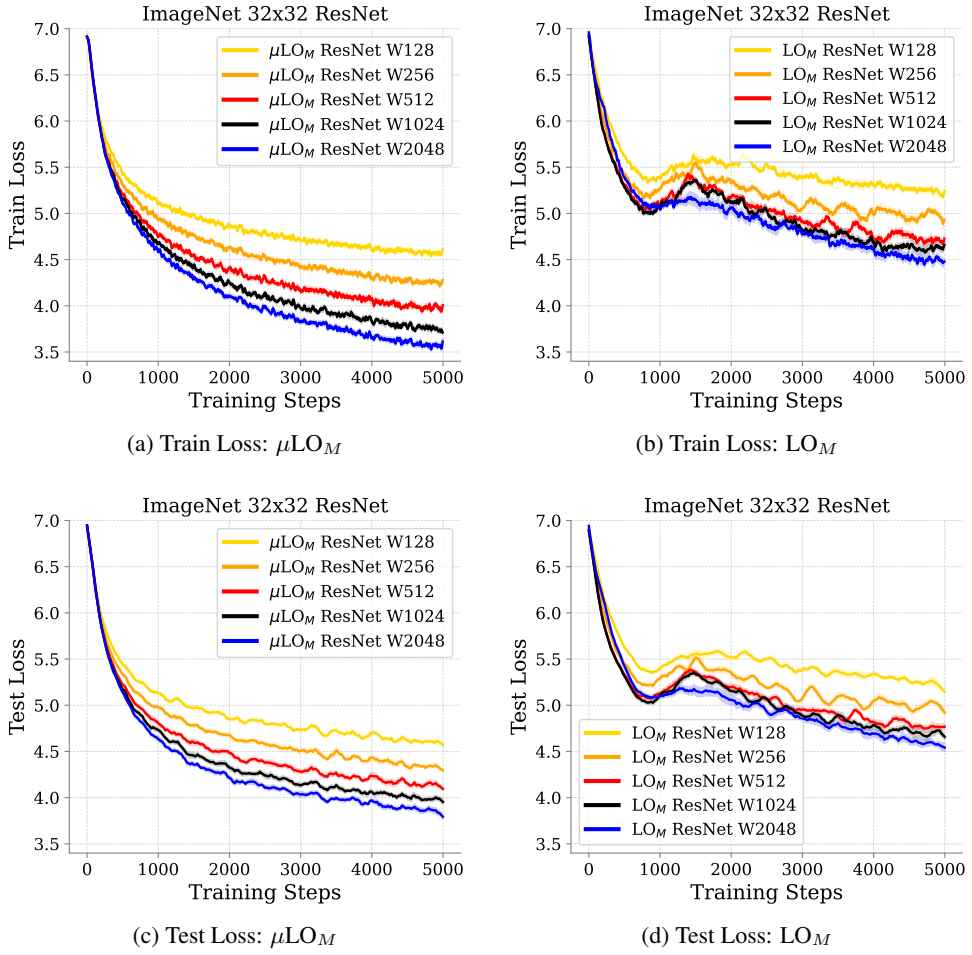


(a) Increasing Depth (b) Longer Training

Figure 9: **Activation stability for deeper and longer training.** Each curve reports the five-seed average L1 norm of first-layer pre-activation and logits for (a) and (b), respectively.

F.1.2 EVALUATING THE PERFORMANCE OF μLO_M ON RESNET TASKS

In this section, we compare the meta-generalization performance of μLO_M to LO_M on ResNet tasks. Figure 10 reports train and test loss during training, while Table 11 reports the final train and test loss across different width ResNet tasks for each optimizer. We observe that each width μLO_M outperforms LO_M and that the final evaluation loss for each closely tracks the training loss.



(a) Train Loss: μLO$_M$

(b) Train Loss: LO$_M$

(c) Test Loss: μLO$_M$

(d) Test Loss: LO$_M$

Figure 10: **Train and Test loss across width on a ResNet task.** Each curve reports the five-seed average. We compare the meta-generalization of μLO$_M$ to LO$_M$ when optimizing ResNet tasks. We find that μLO$_M$ generalizes significantly better.

Table 11: **Final train and test losses on ResNet tasks.** Each value reports the five-seed average and is rounded to two decimals.

Optimizer	Width	Final Train Loss	Final Test Loss
μLO$_M$	128	4.56 ± 0.03	4.58 ± 0.03
μLO$_M$	256	4.25 ± 0.02	4.32 ± 0.03
μLO$_M$	512	3.95 ± 0.02	4.11 ± 0.02
μLO$_M$	1024	3.72 ± 0.03	3.98 ± 0.05
μLO$_M$	2048	3.56 ± 0.02	3.83 ± 0.03
LO$_M$	128	5.20 ± 0.01	5.18 ± 0.02
LO$_M$	256	4.92 ± 0.04	4.93 ± 0.03
LO$_M$	512	4.70 ± 0.02	4.76 ± 0.02
LO$_M$	1024	4.63 ± 0.02	4.70 ± 0.05
LO$_M$	2048	4.47 ± 0.03	4.59 ± 0.05

G Coordinate evolution of MLP layers in μP for Adam and Learned Optimizers

The following section presents the continuation of our experiments comparing pre-activation growth during training for SP LOs and μLOs with different meta-training recipes, SP adam, and μAdam.

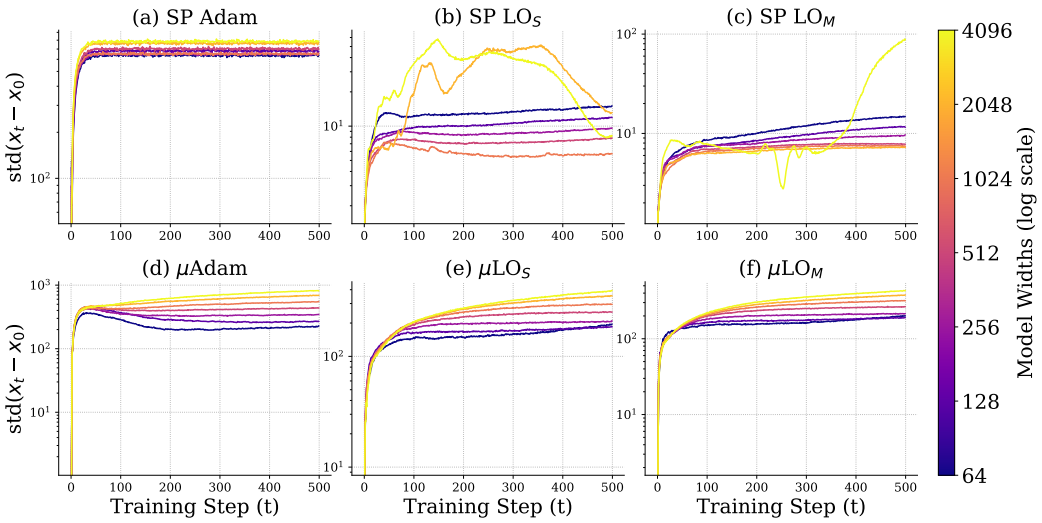


Figure 11: **Layer 0 pre-activations behave harmoniously in μP for LOs and Adam alike.** We report the evolution of coordinate-wise standard deviation between the difference of initial and current second-layer pre-activations. We observe that all models parameterized in μP enjoy stable coordinates across widths, while the pre-activations of larger-width models in SP blow up after a number of training steps. All plots report these metrics for the first 500 steps of a single training run.

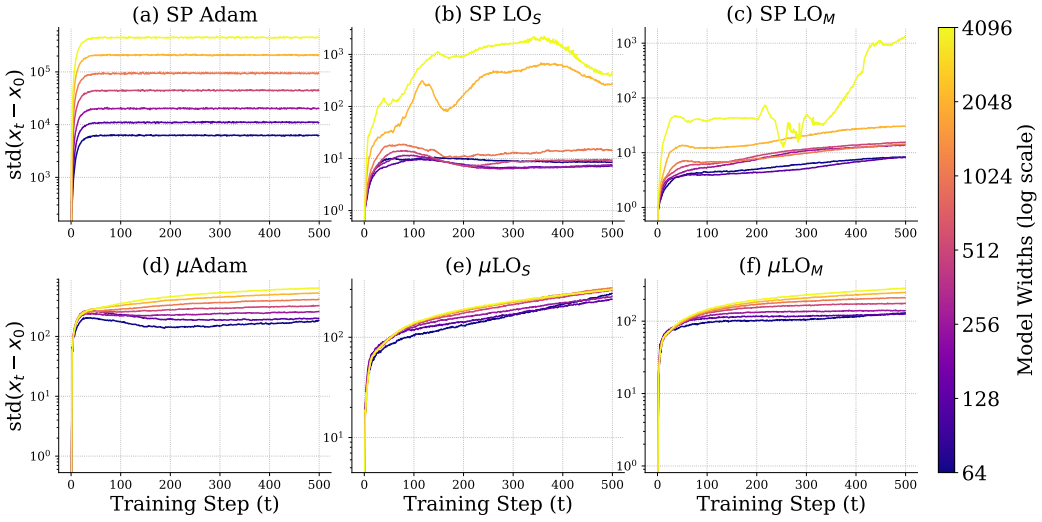


Figure 12: **Layer 1 pre-activations behave harmoniously in μP for LOs and Adam alike.** We report the evolution of coordinate-wise standard deviation between the difference of initial and current second-layer pre-activations. We observe that all models parameterized in μP enjoy stable coordinates across widths, while the pre-activations of larger-width models in SP blow up after a number of training steps. All plots report these metrics for the first 500 steps of a single training run.

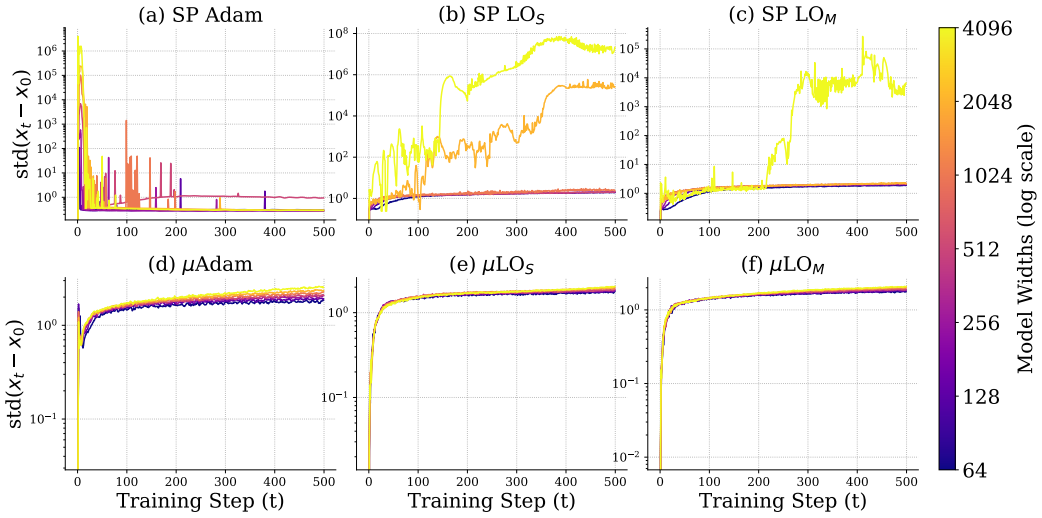


Figure 13: **Logits behave harmoniously in μP for LOs and Adam alike.** We report the evolution of coordinate-wise standard deviation between the difference of initial and current second-layer pre-activations. We observe that all models parameterized in μP enjoy stable logits across widths, while the pre-activations of larger-width models in SP blow up after a number of training steps. All plots report these metrics for the first 500 steps of a single training run.