Is Scaling Learned Optimizers Worth It? Evaluating The Value of VeLO's 4000 TPU Months

Fady Rezk
School of Informatics
University of Edinburgh
F.R.G.Rezk@sms.ed.ac.uk

Antreas Antoniou
School of Informatics
University of Edinburgh
a.antoniou@ed.ac.uk

Henry Gouk
School of Informatics
University of Edinburgh
henry.gouk@ed.ac.uk

Timothy M. Hospedales
University of Edinburgh
Samsung AI Research, Cambridge
t.hospedales@ed.ac.uk

Abstract

We analyze VeLO (versatile learned optimizer [17]), the largest scale attempt to train a general purpose "foundational" optimizer to date. VeLO was trained on thousands of machine learning tasks using over 4000 TPU months with the goal of producing an optimizer capable of generalizing to new problems while being hyperparameter free, and outperforming industry standards such as Adam. We independently evaluate VeLO on the MLCommons optimizer benchmark suite. We find that, contrary to initial claims: (1) VeLO has a critical hyperparameter that needs problem-specific tuning, (2) VeLO does not necessarily outperform competitors in quality of solution found, and (3) VeLO is not faster than competing optimizers at reducing the training loss. These observations call into question VeLO's generality and the value of the investment in training it.

1 Introduction

Meta-learning, or learning to learn, refers to the appealing vision of learning the learning algorithm itself, similarly to how deep learning replaced the tradition of handcrafted feature engineering [11]. Meta-learning has found compelling applications in various facets of AI. In particular, one notable application of meta-learning is to learn improved optimization strategies [1] that provide better or faster optimization than hand-crafted optimizers [4]. After initial successes in relatively small scale problems, researchers have recently focused on scaling learned optimizers [5, 17].

A noteworthy example is VeLO [17]. Trained on a huge array of tasks with over 4000 TPU months, it aspires to be a 'foundational' optimizer capable of solving any new problems more rapidly than hand-designed optimizers such as Adam. VeLO claimed multiple remarkable abilities, such as being at least $4\times$ faster than Adam on 50% of tasks in the VeLOdrome suite. If true, VeLO would eventually pay for its up-front training cost by accelerating learning across the community. Nevertheless, evaluating optimizers – especially learned optimizers – is itself a very difficult problem with multiple facets including iteration and time-efficiency, quality and generalisation of minima discovered, hyperparameter sensitivity [8, 4, 7], and generalisation of the learned optimiser itself.

In this work, we critically analyse VeLO's performance to understand if it is as effective as claimed. Our evaluation casts doubt on its claimed efficacy, and whether scaling-up training is the silver bullet for optimizer learning that it has been in other areas of AI. Our contributions are: (1) Validation of VeLO: We conduct a rigorous, independent evaluation of VeLO's performance using an extended analysis based on the MLCommons benchmark.(2) Claims Reassessment: Our empirical results challenge

I Can Not Believe It Is Not Better 2023. NeurIPS 2023 workshop.

several key claims made in the original VeLO paper, specifically that of being hyperparameter-free, outperforming baselines in minimizing training objectives, and offering optimization speedups. (3) Introduction of Explicit Metrics: We introduce a set of carefully selected metrics that directly align with the fundamental objectives an optimizer should fulfil. These metrics serve as a standardized framework for comparing VeLO against other optimizers.

2 Preliminaries

Let f_θ be a function parameterized by θ where θ is defined over some domain $\theta \in \Theta$. We refer to f_θ as the *optimizee*; the function being optimized. Performance of f_θ on a task $\mathcal{T}_i$ sampled from a distribution of tasks $p(\mathcal{T})$ can be measured by a loss function $L_i(f_\theta, \mathcal{T}_i)$. The goal of learning is to find the minimizer $\theta^* = \arg\min_{\theta\in\Theta} L_i(f_\theta, \mathcal{T}_i)$. Gradient descent minimizes the loss function by producing a sequence of updates in the form:

$$\theta_{t+1} = \theta_t - \alpha_t \nabla_\theta L_i(f_\theta, \mathcal{T}_i) \tag{1}$$

Learning to optimize strategies reformulate gradient descent as

$$\theta_{t+1} = \theta_t + g(L_i(f_\theta, \mathcal{T}_i)), \tag{2}$$

which recovers standard gradient descent when $g(\cdot)$ is a simple scaling $g(L_i(f_\theta, \mathcal{T}_i)) = -\alpha_t \nabla_\theta L_i(f_\theta, \mathcal{T}_i)$. These approaches assume that performance can be improved by paramaterizing the function g with some learnable parameters λ, e.g., defining a small MLP.

Learning-to-optimize is usually formulated as a bi-level optimization problem where the goal is to the learn *optimizer* g so that the *optimizee* f achieves low loss on some task distribution after learning. More specifically:

$$\lambda^* = \arg\min_\lambda \sum_{i=1}^{M} \mathcal{L}(\theta_i^*(\lambda), \lambda, \mathcal{T}_i) \tag{3}$$

$$\text{s.t. } \theta_i^*(\lambda) = \arg\min_\theta L_i(\theta, \lambda, \mathcal{T}_i) \tag{4}$$

where Eq. 4 is solved with the learnable optimizer Eq. 2, and the optimizer learning objective is in Eq. 3. Compactly, the gradient of the loss, after t steps, on a given sampled task would then be [16]:

$$\frac{dL_t}{d\lambda} = \frac{\partial L_t}{\partial \lambda} + \sum_{k=1}^{T} \frac{\partial L_t}{\partial \theta_t} \left(\prod_{i=k}^{T} \frac{\partial \theta_i}{\partial \theta_{i-1}} \right) \frac{\partial \theta_k}{\partial \lambda}, \tag{5}$$

which allows the optimizer to be learned with gradient descent.

3 Background and Motivation

Learning Optimizers Searching for simple and symbolic update rules for training neural networks dates back to the 90's [18, 3]. More recently, [1] parameterized the optimization algorithm as an LSTM which acts coordinate-wise on the inner-loop problem. Various work has since explored the design space of learning optimizers. The space spans a) the parameterization of the learned optimizer including it's IO representation, b) the meta-training task distribution, c) meta-optimizers (optimizer used to update the learned optimizer) and d) the outer-loop objective function for estimating the learned optimizer performance.

Parameterizations included LSTMs [1, 14], hierarchical RNNs [22, 5], MLPs [15], transformers [10] and hyper-networks [17]. Tree structured search spaces [6], domain-specific languages [2] and evolutionary strategies [12] were also explored. Search spaces and black-box parameterizations can be learned using various techniques such as gradient-based meta-learning [1], or evolutionary strategies [6]. Meta-loss functions also vary between inner-loop training [1, 14], validation loss [15, 23], or more complex objectives that measure resource-efficiency [13] and speed [24].

Benchmarking optimizers Benchmarking optimizers – especially in deep learning – is extremely challenging, as there are many facets to optimizer quality including iterations and clock-time to convergence, quality of the solution found in non-convex problems, generalisation of the final

solution to a validation or testing set, hyperparameter sensitivity, consistency of performance across different workloads, etc [21, 7, 20]. As discussed in [8], this is the reason behind multiple apparently contradictory claims in the literature, and the lack of consensus on benchmarks and metrics compared to other areas of machine learning and AI. All this makes it challenging to compare optimizers as they may excel at one facet while falling down in another.

For the reasons discussed above, apples-to-oranges comparisons are common in the literature, and can lead to misleading conclusions. For example, comparing optimizer performance without controlling hyperparameter tuning or HPO objective [21]. This has led to a few attempts to establish common evaluation frameworks for optimizers, notably MLCommons [8], which can control for HPO.

Benchmarking learned optimizers This challenge of optimizer evaluation is further exacerbated when considering benchmarking of learned optimizers, as the cost of optimizer learning, and the robustness of the learned optimizer to diverse and out-of-distribution tasks open up additional important criteria. As optimizer learning is a costly process, most learned optimizers justify themselves with amortization arguments: The idea that the up-front cost of optimizer learning can be paid off by the learned optimizer's improved solution to multiple subsequent tasks. However, the learned optimizer needs to be applied on new tasks for this justification to hold, as good solutions to the training tasks have already been found during optimizer training. Thus, the practical value of a learned optimizer is intrinsically intertwined with both its efficacy and how well it generalizes to new tasks. All this makes fairly benchmarking learned optimizers even harder than handcrafted optimizers.

The VeLO optimizer [17] aspired to achieve both efficient optimization and cross-task generalization by large scale optimizer training on a huge problem suite. It then evaluated the resulting optimizer on the VeLOdrome task suite [17] and an early version of the MLCommons optimizer benchmark suite [8] where it claimed to provide decisive efficiency improvements over competitors, thus justifying its huge up-front training cost. This paper critically evaluates these claims.

4 VeLO: Versatile Learned Optimizer

VeLO Architecture VeLO [17] is a learned optimizer trained with the outer-objective (Eq 3) of minimizing the training loss. The learned optimizer is parameterized as a hierarchical hypernetwork; a per-tensor LSTM that generates the parameters for a per-parameter MLP. The per-tensor hypernetwork operates on features aggregated from each parameter tensor, i.e: neural network layer. VeLO optimizer states and inputs include current iteration number, momentum at different timescales, squared gradients, adafactor-style accumulator, loss exponentially-moving average features, and tensors rank.

VeLO Training The meta-training task distribution included MLPs, CNNs, ResNets, ViTs, auto-encoders, variational auto-encoders, RNNs, and vanilla Transformers of various sizes. The architectures included dynamic configurations such as initialization and activation functions. Standard training datasets for image and language domains such as 16×16 ImageNet, CIFAR 10 and 100, Fashion MNIST, LM1B, and Wikipedia English among others. The meta-optimizer used was standard evolutionary strategy with antithetic-samples [19]. Meta-training spanned a total of 4000 TPU months with an online HPO procedure divided across 4 phases. Problem sizes and training unroll lengths were gradually increased over a curriculum which was found to improve meta-generalization.

VeLO Claims Some key VeLO claims are (a) achieving a $4\times$ speedup over learning rate-tuned Adam on 50% of tasks while being $16\times$ times faster on 14% of VeLOdrome suite of tasks ([17], Fig. 1). (b) out-performing hyperparameter tuned Adam on a suite of tasks from the MLCommons algorithms track in terms of the training loss ([17], Sec. 4.2), (c) out-performing hyperparameter tuned Adam's generalization (validation loss) on the same benchmark ([17], App. G.7).

It can be seen that VeLO's claims span learned optimizer benchmarking practical objectives of (a) training speedups and (b) absolute performance gains on both train and validation metrics while (c) meta-generalizing to new tasks distributions including VeLOdrome and MLCommons benchmark, which is a key justification behind amoritizing VeLO's meta-training cost.

Caveats Besides the inputs discussed in the architecture paragraph above, VeLO needs one special input: It must be prompted with the total training steps it is expected to run for in order to initialize its states. This is then used to estimate the fraction of training remaining online during learning. For an explanation of how to control for this factor of variation fairly, we refer the reader to appendix B.

5 Benchmark Design

To examine VeLO's claims, our point of departure is the most recent time-to-result benchmark by MLCommons [8]. Comparing training curves to measure speedups is ill-posed. Therefore, the MLCommons [8] protocol measures learning speed by fixing a performance target (e.g., loss), and measuring time/steps taken for an optimizer to reach this target. Since VeLO also reported improved solution quality, we extend this protocol to the complementary perspective of fixing an optimization time/step budget and measuring the loss achieved at this point.

Baselines and Workloads The original VeLO paper mainly compared with Adam. For more thorough evaluation, we train several GD variants, namely SGD with Heavy Ball Momentum, SGD with Nesterov Momentum, Adam, NAdam (Adam with Nesterov Momentum) and NAdamW. We train all baselines with default hyperparameters as reported in appendix C. All algorithms are trained for a maximum allowed budget, either runtime or steps, on 4 workloads from the MLCommons benchmark, namely ResNet-50 on Imagenet, GNN on OGBG, DLRM on Criteo-1TB and U-Net on FastMRI. A workload is a fixed architecture, dataset and training objective. Please refer to [8] for details.

Measuring Training Speedups The key evaluation hyperparameter in MLCommons is the notion of a *performance target* (e.g., in units of loss) that defines a successful optimization. We can then measure speedups in terms of the wall-clock time or number of iterations taken to reach the target.

Establishing performance targets is somewhat involved in the MLCommons methodology. First one sets a maximum allowed runtime in wall-clock or step count for each workload, runs multiple trials of all algorithms for the full budget, and then measures the performance of all algorithms trials at 75% of the maximum allowed budget. Then, for each algorithm on each workload, the median performance is selected, and the best performing algorithm defines the target for the workload. This translates to $\text{target}_w = \max_a\{\text{median}_s\{L_{a,s,w}\}\}$ for all $w \in \mathcal{W}$ where $L_{a,s,w}$ is the performance metric of interest achieved by algorithm a on trial s and workload w.

Subsequently we can measure the time/steps $t_{a,s,w}$ that the sth trial of any given algorithm a takes to reach the target performance level $\textbf{target}_w$ on workload w. To aggregate results, we can employ performance profiles [9, 8]. Denote algorithms by $\mathcal{A} = \{a_1, a_2, .., a_k\}$ and workloads as $\mathcal{W} = \{w_1, w_2, .., w_n\}$. Then, given a workload w, we record the median time/steps taken for algorithm a to achieve the performance target across all trials/seeds as $t_{a,w} = \textbf{median}_s\{t_{a,s,w}\}$. Then, to score an algorithm $\hat{a}$ on a workload w, the performance ratio is defined as:

$$r_{\hat{a},w} = \frac{t_{\hat{a},w}}{\min_{a\in\mathcal{A}} t_{a,w}} \tag{6}$$

The performance profile $\rho_{\hat{a}}(\tau)$ for an algorithm $\hat{a}$ on a random workload w drawn uniformly from $\mathcal{W}$ is the probability of having a performance ratio $r_{\hat{a},w}$ of at most τ:

$$\rho_{\hat{a}}(\tau) = \left(\frac{1}{n}\right) \times |\{w : r_{\hat{a},w} \leq \tau\}| \tag{7}$$

Following [8], the final score B_a for each algorithm integrates the performance profile over a predefined range $r_{\textbf{max}}$ resembling a space of τs and normalized by $r_{\textbf{max}} - 1$. This means that an algorithm that is consistently the fastest across all workload would have a score of 1.

In summary, for individual benchmarks w and algorithm a, we report time-to-target $t_{a,w}$. We measure both wall-clock-time to target (denoted time-control condition), and steps to target (step-control condition). To aggregate across benchmarks we report the aggregate MLCommons score B_a.

Measuring Training Quality While MLCommons mainly focuses on training speedup, the complementary metric to is quality of solution found within a certain time/step budget. To this end, we also assess training and validation performance $p_{a,w}$ (e.g., loss, accuracy) after algorithm a reaching a certain time/step quota for workload w (denoted performance-control).

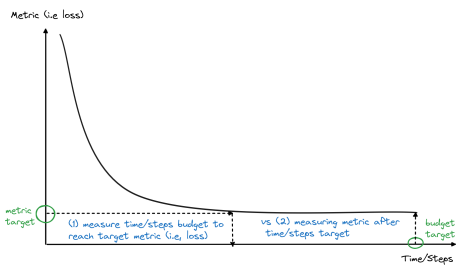


Figure 1: Illustration of optimizer learning metrics: Time/steps to performance target vs performance achieved at time budget.

For the specific workload budgets and targets found, please see Appendix A. The training speedup vs training-quality metrics are illustrated schematically in Figure 1. Finally, see appendix F for a comparison of similarities and extensions between our evaluation and VeLO's original assessment.

6 Experiments

We now set out to assess whether VeLO's claims are justified, and hence whether its large up-front training cost can be justified from an amortization perspective.

Specifically, we ask the following questions:

(Q1) *Is VeLO hyperparameter free as claimed?*

(Q2) *Does VeLO indeed outperform existing hand-crafted optimizers on training and validation loss minimization as claimed?*

(Q3) *Does VeLO indeed provide dramatically faster optimization than standard baselines?*

Conclusion 1: VeLO is Not Hyperparameter Free but Hyperparameter Sensitive. Recall that the VeLO has one user-defined input: It requires prompting with the total number of steps (Sec 4 and [17]). It will accelerate (attempt to converge faster, but possibly reach a worse minima) if prompted with fewer steps. We study the MLCommons time-to-performance target protocol for different values of this hyperparameter. We consider prompting with steps corresponding to either 100% or 75% of MLCommons wall-clock max runtime. From the results in Table 1 we see that the prompt is actually a key hyperparameter. For example, the 75% prompt reaches the Criteo training target before timeout, while the 100% prompt doesn't succeed in time. Meanwhile the 100% prompt reaches the OGBG training target before timeout, while the 75% prompt does not (it behaves too greedily and converges to a poor optimum worse than the required performance target). Overall VeLO is in fact sensitive to the number of steps hyperparameter, often crucially so.

Table 1: Influence of VeLO's step prompt hyperparameter. Time (↓, sec) taken to reach MLCommons training and validation performance targets. The prompt is given as steps corresponding to 100% or 75% of MLCommons wall-clock budget for the task. (-) indicates timeout.

	Workload	Criteo	FastMRI	ImageNet	OGBG
Train	VeLO (100%)	-	-	-	**13215**
	VeLO (75%)	**6088**	-	-	-
Val	VeLO (100%)	-	**5728**	62587	8779
	VeLO (75%)	-	5913	**58713**	**7238**

Conclusion 2: VeLO Does Not Outperform Baselines in Minimizing Both Training and Validation Losses. VeLO reported outperforming hand-crafted optimizers in terms of achieving lower training and validation losses, both on VeLOdrome and MLCommons (algorithm) benchmark suites. But the associated experiments on MLCommons compared against Adam alone [17]. Meanwhile, [8] observe that different optimizers often 'win' on different benchmarks. So we directly compare VeLO against a range of off-the-shelf optimizers with default hyperparameters in terms of optimization quality after a fixed step-budget on MLCommons. From the results in Table 2 (see full details in Appendix E), we see a different picture: VeLO is not a consistent winner in either train or validation loss achieved, despite that we conducted no HPO at all on the baselines. We attribute this discrepancy two factors: (1) VeLO [17] evaluating insufficient competitors in their original comparison – since as we see different competitors win on different benchmarks/metrics. (2) VeLO's evaluation primarily focused on VeLOdrome benchmarks, which were reportedly more similar to VeLO's training distribution [17], and focused less on the MLCommons suite which was reportedly more different. To the extent that this is the explanation, it suggests that VeLO is not as general purpose as claimed, and thus undermines the amortization argument used to justify its up-front training cost.

Conclusion 3: VeLO Does Not Provide Faster Training. VeLO claims substantially faster training. It was trained for the objective of fast training loss minimisation, and empirically observed to also provide fast validation loss minimisation. However, again these original claims were largely based on the VeLOdrome benchmark (which may be unrealistically easy, as discussed in the previous

Table 2: Training and validation losses (↓) across workloads after training each algorithm for a fixed number of steps. VeLO is not a consistent winner.

	Workload / Optimizer	Criteo-1TB	FastMRI	ImageNet	OGBG
Train	Adam	0.1225 ± 0.00015	$\mathbf{0.2702 \pm 0.00693}$	0.0470 ± 0.00587	0.0165 ± 0.00033
	Heavy Ball	0.1293 ± 0.00045	0.2809 ± 0.00282	0.2582 ± 0.02502	0.0341 ± 0.00026
	NAdam	0.1239 ± 0.00302	0.2692 ± 0.00396	0.0571 ± 0.00060	0.0173 ± 0.00033
	NAdamW	$\mathbf{0.1220 \pm 0.00018}$	0.2750 ± 0.00102	$\mathbf{0.0470 \pm 0.00106}$	0.0197 ± 0.00196
	Nesterov	0.1301 ± 0.00096	0.2811 ± 0.00505	0.2512 ± 0.02606	0.0331 ± 0.00008
	VeLO	0.1229 ± 0.00034	0.2804 ± 0.00008	0.1046 ± 0.00342	$\mathbf{0.0153 \pm 0.00088}$
Validation	Adam	$\mathbf{0.1237 \pm 0.00005}$	$\mathbf{0.2850 \pm 0.00003}$	1.9438 ± 0.00717	0.0515 ± 0.00020
	Heavy Ball	0.1299 ± 0.00062	0.2899 ± 0.00008	1.6870 ± 0.02763	0.0466 ± 0.00038
	NAdam	0.1256 ± 0.00315	0.2851 ± 0.00022	1.9528 ± 0.00120	0.0509 ± 0.00027
	NAdamW	$\mathbf{0.1237 \pm 0.00005}$	0.2851 ± 0.00016	1.6345 ± 0.01098	0.0483 ± 0.00157
	Nesterov	0.1305 ± 0.00067	0.2899 ± 0.00004	1.7022 ± 0.03822	$\mathbf{0.0462 \pm 0.00027}$
	VeLO	0.1240 ± 0.00008	0.2851 ± 0.00022	$\mathbf{1.5017 \pm 0.02693}$	0.0522 ± 0.00126

section), and in terms of MLCommons they were based on comparison to Adam alone. We now compare VeLO to a range of off-the-shelf optimizers with default hyperparameters on our four MLCommons tasks using the time/steps to performance target protocol of MLCommons. The MLCommons benchmark results presented in Table 3 in terms of the aggregate MLCommons score B_a, which integrates over the algorithms' performance profiles (see Appendix D and E for details).

Surprisingly, VeLO is far from best in training speed (which might be expected given it is optimised for training efficacy), although it surpasses some baselines in speed of minimising the validation loss. VeLO's loss to Adam in training efficiency we attribute to (1) weak generalisation to the MLCommons task suite, and (2) Adam's default learning rate decay schedule potentially being more effective than the outcome of the amount of HPO applied with Adamin in [17]. VeLO's comparative success in validation is potentially attributable to several MLcommons workloads being in the overfitting regime[1], so VeLO's less effective minimisation of the train loss can lead to better validation than competitors. (Note that while we measure validation performance, all optimizers are run with default parameters and not tuned on validation metrics.). This is particularly the case in the time-control condition because since Velo is slower per-iteration than the baselines, it runs fewer iterations than baselines when using a wall clock-time budget, and thus effectively benefits from early stopping compared to the baselines. Finally, returning to the hyperparameter sensitivity issue from Experiment 1, we also compare VeLO with 75% of the total step-prompt and see a noticeable impact in the score distribution.

7 Conclusion

Learned optimizers have shown substantial success on narrowly defined task distributions. VeLO scaled up optimizer learning to train a foundational optimizer on a vast task distribution at huge cost. The vision was that it would then generalize to arbitrary machine learning workloads, and outperform hand-crafted optimizers, thus justifying its up-front training cost. We were initially optimistic and excited to see this in action. However ultimately, our independent evaluation on the MLCommons optimizer benchmark called into question most of VeLO's big claims of being hyperparameter free, and providing improved and faster optimization.

Table 3: Optimizer speed evaluation (MLCommons score B_a, (↑), Eq. 7). Time-To-Result and Steps-To-Results are reported when fixing wall-clock time and steps respectively across train and validation targets.

Optimizer	**Train Scores**		**Validation Scores**	
	Time	**Step**	**Time**	**Step**
NAdam	0.24	0.25	0.00	0.23
NAdamW	0.36	0.25	0.49	0.36
Adam	**1.00**	**1.00**	0.24	**0.50**
Nesterov	0.00	0.00	0.00	0.00
HeavyBall	0.00	0.00	0.00	0.00
VeLO	0.19	0.00	**0.71**	0.39
VeLO Short	0.16	0.03	0.74	0.59

[1]A regime where fully minimizing the training loss ultimately worsens validation performance

Acknowledgments and Disclosure of Funding

We extend sincere gratitude to Frank Schneider, Zach Nado, and George Dahl for their support. During the course of this research, they provided clarifications on conceptual ideas regarding the design and implementation of the MLCommons benchmark.

The project was supported by the Royal Academy of Engineering under the Research Fellowship programme. This work was also supported by the Edinburgh International Data Facility (EIDF) and the Data-Driven Innovation Programme at the University of Edinburgh.

References

[1] M. Andrychowicz, M. Denil, S. G. Colmenarejo, M. W. Hoffman, D. Pfau, T. Schaul, B. Shillingford, and N. de Freitas. Learning to learn by gradient descent by gradient descent. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, NIPS'16, page 3988–3996, Red Hook, NY, USA, 2016. Curran Associates Inc. ISBN 9781510838819.

[2] I. Bello, B. Zoph, V. Vasudevan, and Q. V. Le. Neural optimizer search with reinforcement learning. In D. Precup and Y. W. Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 459–468. PMLR, 06–11 Aug 2017. URL https://proceedings.mlr.press/v70/bello17a.html.

[3] S. Bengio, Y. Bengio, J. Cloutier, , and J. Gecsei. On the optimization of a synaptic learning rule. In *Conference on Optimality in Artificial and Biological Neural Networks*.

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

[5] X. Chen, T. Chen, Y. Cheng, W. Chen, A. Awadallah, and Z. Wang. Scalable learning to optimize: A learned optimizer can train big models. In *Computer Vision – ECCV 2022: 17th European Conference, Tel Aviv, Israel, October 23–27, 2022, Proceedings, Part XXIII*, page 389–405, Berlin, Heidelberg, 2022. Springer-Verlag. ISBN 978-3-031-20049-6. doi: 10.1007/978-3-031-20050-2_23. URL https://doi.org/10.1007/978-3-031-20050-2_23.

[6] X. Chen, C. Liang, D. Huang, E. Real, K. Wang, Y. Liu, H. Pham, X. Dong, T. Luong, C.-J. Hsieh, Y. Lu, and Q. V. Le. Symbolic discovery of optimization algorithms, 2023.

[7] D. Choi, C. J. Shallue, Z. Nado, J. Lee, C. J. Maddison, and G. E. Dahl. On empirical comparisons of optimizers for deep learning, 2020. URL https://openreview.net/forum?id=HygrAR4tPS.

[8] G. E. Dahl, F. Schneider, Z. Nado, N. Agarwal, C. S. Sastry, P. Hennig, S. Medapati, R. Eschenhagen, P. Kasimbeg, D. Suo, J. Bae, J. Gilmer, A. L. Peirson, B. Khan, R. Anil, M. Rabbat, S. Krishnan, D. Snider, E. Amid, K. Chen, C. J. Maddison, R. Vasudev, M. Badura, A. Garg, and P. Mattson. Benchmarking neural network training algorithms, 2023.

[9] E. D. Dolan and J. J. Moré. Benchmarking optimization software with performance profiles. *Mathematical Programming*, 91(2):201–213, Jan. 2002. doi: 10.1007/s101070100263. URL https://doi.org/10.1007/s101070100263.

[10] E. Gärtner, L. Metz, M. Andriluka, C. D. Freeman, and C. Sminchisescu. Transformer-based learned optimization, 2023.

[11] T. Hospedales, A. Antoniou, P. Micaelli, and A. Storkey. Meta-learning in neural networks: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(09):5149–5169, sep 2022. ISSN 1939-3539. doi: 10.1109/TPAMI.2021.3079209.

[12] R. T. Lange, T. Schaul, Y. Chen, T. Zahavy, V. Dalibard, C. Lu, S. Singh, and S. Flennerhag. Discovering evolution strategies via meta-black-box optimization. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=mFDU0fP3EQH.

[13] C. Li, T. Chen, H. You, Z. Wang, and Y. Lin. Halo: Hardware-aware learning to optimize. In *Computer Vision – ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part IX*, page 500–518, Berlin, Heidelberg, 2020. Springer-Verlag. ISBN 978-3-030-58544-0. doi: 10.1007/978-3-030-58545-7_29. URL https://doi.org/10.1007/978-3-030-58545-7_29.

[14] K. Lv, S. Jiang, and J. Li. Learning gradient descent: Better generalization and longer horizons. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, page 2247–2255. JMLR.org, 2017.

[15] L. Metz, N. Maheswaranathan, J. Nixon, D. Freeman, and J. Sohl-Dickstein. Understanding and correcting pathologies in the training of learned optimizers. In K. Chaudhuri and R. Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 4556–4565. PMLR, 09–15 Jun 2019. URL https://proceedings.mlr.press/v97/metz19a.html.

[16] L. Metz, C. D. Freeman, S. S. Schoenholz, and T. Kachman. Gradients are not all you need, 2022.

[17] L. Metz, J. Harrison, C. D. Freeman, A. Merchant, L. Beyer, J. Bradbury, N. Agrawal, B. Poole, I. Mordatch, A. Roberts, and J. Sohl-Dickstein. Velo: Training versatile learned optimizers by scaling up, 2022.

[18] T. Runarsson and M. Jonsson. Evolution and design of distributed learning rules. In *2000 IEEE Symposium on Combinations of Evolutionary Computation and Neural Networks. Proceedings of the First IEEE Symposium on Combinations of Evolutionary Computation and Neural Networks (Cat. No.00EX448)*. IEEE, 2000. doi: 10.1109/ecnn.2000.886220. URL https://doi.org/10.1109/ecnn.2000.886220.

[19] T. Salimans, J. Ho, X. Chen, S. Sidor, and I. Sutskever. Evolution strategies as a scalable alternative to reinforcement learning, 2017.

[20] R. M. Schmidt, F. Schneider, and P. Hennig. Descending through a crowded valley - benchmarking deep learning optimizers. In M. Meila and T. Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 9367–9376. PMLR, 18–24 Jul 2021. URL https://proceedings.mlr.press/v139/schmidt21a.html.

[21] P. T. Sivaprasad, F. Mai, T. Vogels, M. Jaggi, and F. Fleuret. Optimizer benchmarking needs to account for hyperparameter tuning. In H. D. III and A. Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 9036–9045. PMLR, 13–18 Jul 2020. URL https://proceedings.mlr.press/v119/sivaprasad20a.html.

[22] O. Wichrowska, N. Maheswaranathan, M. W. Hoffman, S. G. Colmenarejo, M. Denil, N. de Freitas, and J. Sohl-Dickstein. Learned optimizers that scale and generalize. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, page 3751–3760. JMLR.org, 2017.

[23] Y. Xiong and C.-J. Hsieh. Improved adversarial training via learned optimizer, 2020.

[24] J. Yang, X. Chen, T. Chen, Z. Wang, and Y. Liang. M-l2o: Towards generalizable learning-to-optimize by test-time fast self-adaptation. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=s7oOe6cNRT8.

A Benchmark Details

Setting Maximum Allowed Wall-Clock Time Across Hardware The MLCommons benchmark is based on fixing a maximum allowed wall-clock time for each workload, denoted time-control condition. To transfer this maximum allowed wall-clock runtime across hardware, we compute the ratio between time per step of algorithms on their hardware (8×V100) and ours (1×A100-80GB or 2×A100-80GB). Then, this ratio is used as a multiplier factor of the maximum allowed wall-clock time for each workload.

To get time per step for the original V100 hardware, we use total number of steps each algorithm runs for and the equivalent wall-clock time for those steps as supplemented by the authors in table 28 in [8]. For a given workload, the bold time entry is the maximum allowed wall-clock runtime. For the algorithm with this bold-entry, the steps it runs for in the wall-clock time can be found in the *Steps* row. For reference, we copy the numbers here as in table 4.

Table 4: Steps and corresponding runtime for the target-setting algorithms from [8] table 28 on (8 x V100 16GB) Hardware.

Workload Datasheet for (8 x V100 16GB) Hardware				
	Criteo 1TB DLRMsmall	**FastMRI U-Net**	**ImageNet Resnet-50**	**OGBG GNN**
Optimizer	NAdamW	Nesterov	Heavy Ball	Nesterov
Steps	10,667	38,189	186,667	80,000
Runtime (sec)	7703	8859	63,008	18,477
Time Per Step (sec)	0.7221336833	0.2319777947	0.3375422544	0.2309625

To transfer the wall-clock time, we execute the implementation of the algorithm with this bold time entry as found here on our hardware for 5% of it's steps. Then, we compute our time-per-step on both 1× and 2× A100 GPUs. Finally, we use the ratio between the V100 and A100 time-per-step as a factor multiplication of the wall-clock time. The time per step hardware benchmarking results for A100 GPUs are shown in table 5. To maximally utilize our infrastructure, we use 2 GPUs for FastMRI and ImageNet workloads and 1 GPU for Criteo and OGBG experiments. The final maximum allowed wall-clock runtime are also reported in table 5.

Table 5: Steps and corresponding runtime for the target-setting algorithms on A100-SXM4-80GB Hardware. The steps FastMRI workload is benchmarked for are 1101 steps less than the 5% ratio. This is because FastMRI data loading introduces initial noise in the algorithm and hence time per step are evaluated over the final 808 steps.

	Criteo 1TB DLRMsmall	**FastMRI U-Net**	**ImageNet Resnet-50**	**OGBG GNN**
Steps (∼5% of table 4)	533	808	9,000	4,000
1×A100-SXM4-80GB Hardware				
Runtime (sec)	569	367	8,387	743
Time Per Step (sec)	1.067542214	0.4542079208	0.9318888889	0.185745
2×A100-SXM4-80GB Hardware				
Runtime (sec)	513	220	3,951	666
Time Per Step (sec)	0.9624765478	0.2699386503	0.4381237525	0.1665
Final Maximum Allowed Wall-Clock Times				
Number of GPUs	1	2	2	1
Maximum Allowed Time	11,387	10,308	81,783	14,859

Established Targets in Maximum Allowed Time/Steps We set targets and measure time/steps to reach those targets as standardized in the MLCommons benchmark. To set the self-tuning regime targets, we use the methodology introduced in section 5 as done originally by [8]. We set separate

targets for the time-control and step-control conditions. Targets and maximum allowed runtimes for both time-control and step-control conditions in table 6. The tables also include the maximum allowed wall-clock time or maximum allowed steps to run for.

Table 6: Total allowed runtimes and targets for all workloads. The target metrics are cross-entropy loss (CE), structural similarity index measure (SSIM), accuracy and mean average precision (mAP)

	Criteo 1TB DLRMsmall	**FastMRI U-Net**	**ImageNet Resnet-50**	**OGBG GNN**
Wall-Clock Time-Control Condition				
Maximum Allowed Time	11,387	10,308	81,783	14,859
Metric	CE	SSIM	Accuracy	mAP
Training Target	0.12215	74.543%	98.709%	75.946%
Validation Target	0.12367	72.671%	71.012%	27.867%
Step-Control Condition				
Maximum Allowed Steps	7,000	36,000	150,000	80,000
Metric	CE	SSIM	Accuracy	mAP
Training Target	0.1225	74.535%	98.147%	76.547%
Validation Target	0.12408	72.659%	70.600%	27.849%

Measuring Training Quality For measuring training quality, we measure the final performance $p_{a,w}$ achieved within a certain time/step budget. The time and steps budgets used are the maximum allowed wall-clock time and maximum allowed steps used for the time-to-result benchmark. These maximum runtime are presented in table 6 as maximum allowed steps and maximum allowed time for step-control and time-control conditions respectively.

B Managing VeLO Inputs

VeLO requires total steps at input to intialize optimizer states. This is used to compute percentage of remaining training, a feature input to the LSTM hypernetwork. We can follow two different approaches to provide this input to VeLO. First, we could refactor the benchmark and VeLO implementation to provide the percentage of time remaining directly as input while the experiment is running. This would be computed as the ratio of time remaining and total allowed runtime. The remaining time can be computed directly from the benchmark accumulated_submission_time variable from the MLCommons benchmark which is updated every step by the profiler. A simpler approach is estimating the steps VeLO can take within the maximum allowed wall-clock time. We opt for the latter. In table 7, we provide the estimates over two runs of VeLO for 5% of the step hints discussed in appendix A.

The implementation is written in jax. Jax compiles the computational graphs using XLA. We ommit the compilation times from the estimates since they take insignificant ratio from the whole training runtime but can potentially influence the estimate over 5% of the runtime. To explain the rows in table 7, we first run VeLO for a fixed number of steps corresponding to row **Steps Run**. These are the same steps used earlier in appendix A. Then, we measure the total runtime as reported in **Observed Runtime (sec)**. The **Time Per Step (sec)**, the ratio of the first and second row are used to estimate the hyperparameter, **Estimated Total Steps**. Subsequently, we average the total steps VeLO can fit in the runtime over the two estimates. We train the workloads using VeLO from start to finish once and then update the total steps for each workload given the actual observed steps and run for two more trials. Since we take median over trials, the evaluation of VeLO is insensitive to any outliers produced by the estimates.

For the performance-control condition, where we run for a total fixed number of steps, we run VeLO from start to finish given the maximum allowed steps in table 6. Meanwhile, for the VeLO Short run, denoted also as VeLO (75%) in table 1, which is prompted with 75% with the steps VeLO can run in the maximum allowed wall-clock time, we use 75% of the steps reported in the final row of table 7.

Table 7: VeLO main hyperparameter estimate; total steps it will run for.

	Criteo 1TB DLRMsmall	**FastMRI U-Net**	**ImageNet Resnet-50**	**OGBG GNN**
Number GPUs	1	2	2	1
VeLO Total Steps Estimate 1				
Steps Run	533	44,799	9,018	4,000
Observed Runtime (sec)	791	12080	4651	750
Time Per Step (sec)	1.484052533	0.2696488761	0.5157462852	0.1875
Estimated Total Steps	7672	38227	158572	79248
VeLO Total Steps Estimate 2				
Steps Run	533	65,576	9,018	4,000
Observed Runtime (sec)	791	17863	4635	746
Time Per Step (sec)	1.484052533	0.2724014883	0.5139720559	0.1865
Estimated Total Steps	7672	37841	159119	79672
Trial 1 Estimated Steps	7672	38034	158845	79460
Trial 1 Actual Steps (used in trials 2/3)	7545	37160	156960	74972

C Default Hyperparameters

For all default hyperparameters used for Adam and SGD variants, please refer to table 8. The learning rate schedule consists of a linear warmup followed by cosine decay as illustrated in figure 2. The schedule requires a total number of steps to operate on. We set the total steps of the schedule to 75% of the step hint provided by the MLCommons benchmark for each workload. The step hint is approximately the total steps the SGD variants can run for given the maximum allowed wall-clock time of the benchmark. We set the warmup and cosine decay steps to the first 5% of the schedule steps and the remaining 95% respectively.

Table 8: Default Hyperparameters for all baseline optimizers.

Hyperparameter	Adam	NAdamW	NAdam	Heavy Ball	Nesterov
Base LR	5e-3	1e-2	1e-2	5e-2	5e-2
L_2 Regularization	-	4e-3	-	-	-
β_1	0.9	0.9	0.9	0.9	0.9
β_2	0.999	0.999	0.999	-	-
Schedule	linear warmup	linear warmup	linear warmup	-	-
	+ cosine decay	+ cosine decay	+ cosine decay	-	-
Warmup Steps	5%	5%	5%	-	-
Decay Steps	95%	95%	95%	-	-
Minimum LR	1e-4	1e-4	1e-4	-	-

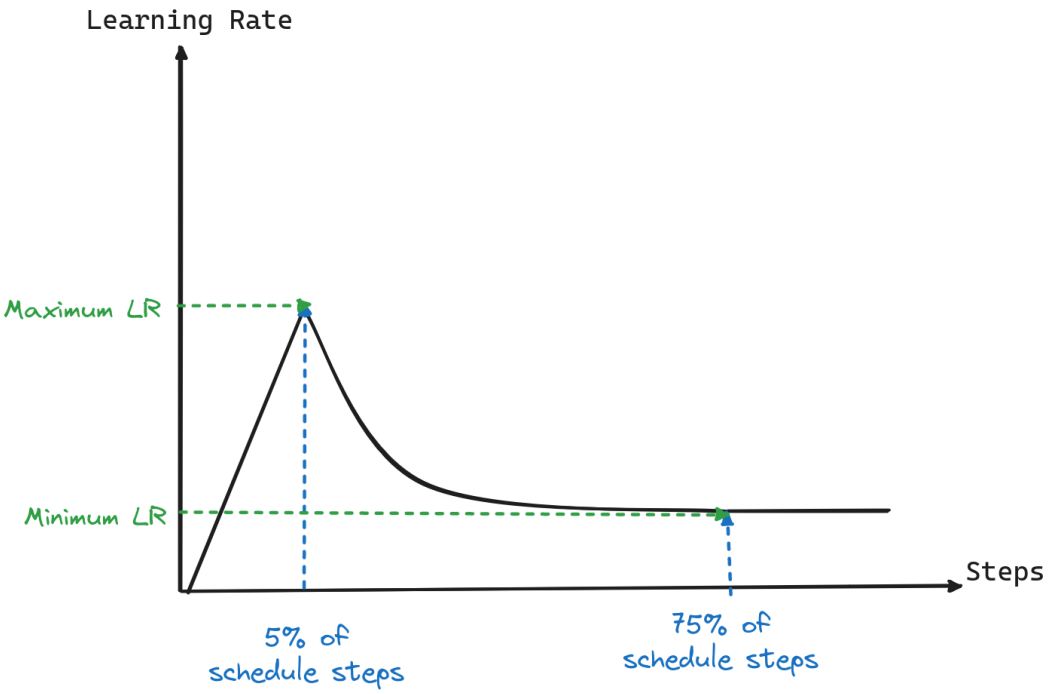


Figure 2: The Learning Rate schedule used for Adam variants from table 8. The first 5% of steps are the linear warmup to the maximum LR, which is followed by 75% of cosine decay to a minimum LR.

D Time-Controlled Experiments Results

D.1 Time-To-Result Measurements

Table 9: Total wall-clock time (sec) to achieve the train target performance. These numbers are used to plot the performance profiles in figure 3. A value of inf indicates that that baseline was unable to achieve the target within the maximum allowed runtime.

Workload Optimizer	Criteo	FastMRI	ImageNet	OGBG
Adam	**3746**	**5817**	**48341**	**9277**
HeavyBall	inf	inf	inf	inf
NAdam	4096	inf	inf	inf
NAdamW	6232	inf	70831	inf
Nesterov	inf	inf	inf	inf
VeLO	inf	inf	inf	13215
VeLO Short	6088	inf	inf	inf

Figure 3: Time-to-Target performance profiles of baselines vs VeLO and VeLO with 75% (VeLO Short) prompt on training targets.

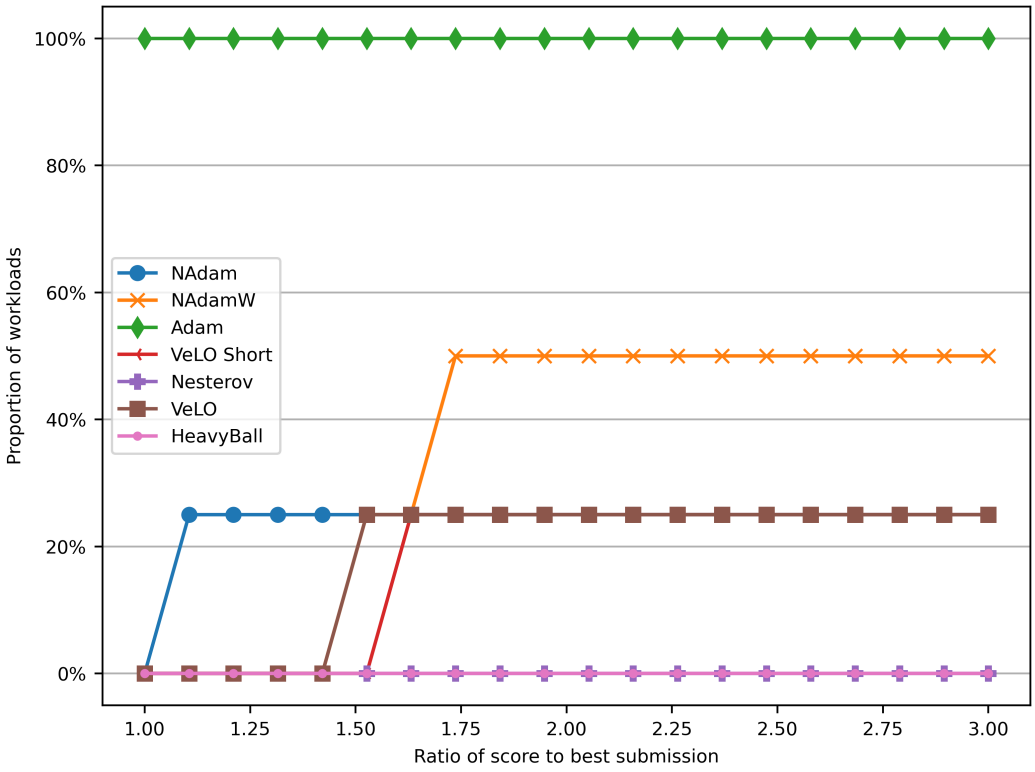
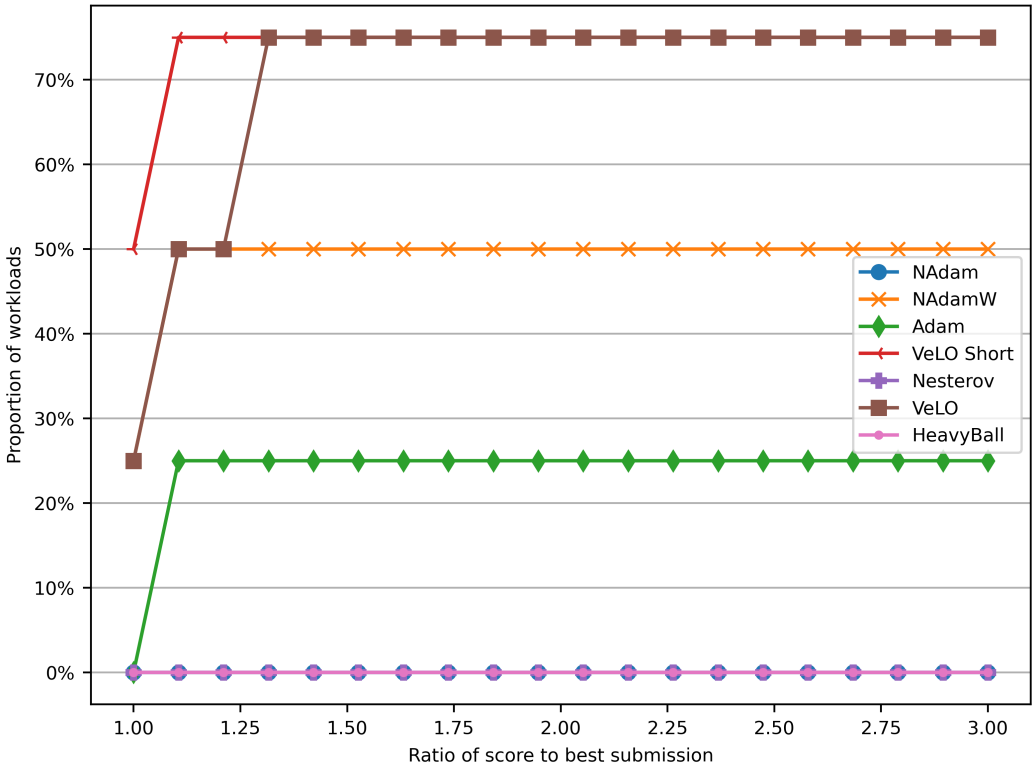


Table 10: Total wall-clock time (sec) to achieve the validation target performance. These numbers are used to plot the performance profiles in figure 4. A value of inf indicates that that baseline was unable to achieve the target within the maximum allowed runtime.

Workload Optimizer	Criteo	FastMRI	ImageNet	OGBG
Adam	7644	inf	inf	inf
HeavyBall	inf	inf	inf	inf
NAdam	inf	inf	inf	inf
NAdamW	**6939**	inf	63502	inf
Nesterov	inf	inf	inf	inf
VeLO	inf	**5728**	**62587**	**8779**
VeLO Short	inf	5913	58713	7238

Figure 4: Time-to-Target performance profiles of baselines vs VeLO and VeLO with 75% (VeLO Short) prompt on validation targets.



D.2 ImageNet

Name	train/loss	train/accuracy	validation/loss	validation/accuracy
Adam	0.0481 ± 0.00531	98.6401 ± 0.16370	1.9405 ± 0.00915	69.8780 ± 0.03027
Heavy Ball	0.1961 ± 0.06906	94.1552 ± 2.11355	1.7592 ± 0.05631	66.1827 ± 0.22902
NAdam	0.0558 ± 0.00040	98.3976 ± 0.05396	1.9516 ± 0.00162	70.0233 ± 0.13590
NAdamW	0.0479 ± 0.00101	98.7139 ± 0.01168	1.6328 ± 0.00854	71.4420 ± 0.06245
Nesterov	0.1572 ± 0.04028	95.4427 ± 1.31832	1.7840 ± 0.00779	66.2933 ± 0.34269
VeLO	0.0862 ± 0.00715	97.4058 ± 0.27273	1.5358 ± 0.02996	72.9073 ± 0.09617
VeLO Short	0.1445 ± 0.00158	95.6785 ± 0.04413	1.3775 ± 0.00333	73.2160 ± 0.10806
HPO	0.5380 ± 0.00695	92.0088 ± 0.23923	1.1170 ± 0.00383	77.4887 ± 0.11420

D.3 FastMRI

Name	train/loss	train/ssim	validation/loss	validation/ssim
Adam	0.2702 ± 0.00693	74.2444 ± 0.51236	0.2850 ± 0.00002	72.6139 ± 0.01243
Heavy Ball	0.2806 ± 0.00298	72.8935 ± 0.20797	0.2897 ± 0.00014	71.9518 ± 0.05269
NAdam	0.2692 ± 0.00396	74.3033 ± 0.50080	0.2851 ± 0.00022	72.6006 ± 0.03608
NAdamW	0.2750 ± 0.00102	73.4651 ± 0.09084	0.2851 ± 0.00015	72.5916 ± 0.04572
Nesterov	0.2809 ± 0.00508	72.9652 ± 0.48978	0.2898 ± 0.00005	71.9132 ± 0.02163
VeLO	0.2737 ± 0.00324	74.0819 ± 0.41933	0.2851 ± 0.00008	72.6663 ± 0.00503
VeLO Short	0.2763 ± 0.00128	73.6923 ± 0.09914	0.2850 ± 0.00026	72.6646 ± 0.02622
HPO	0.2716 ± 0.00371	74.2704 ± 0.30959	0.2851 ± 0.00067	72.6110 ± 0.13964

D.4 Criteo-1TB

Name	train/loss	validation/loss
Adam	0.1222 ± 0.00098	0.1237 ± 0.00005
Heavy Ball	0.1268 ± 0.00110	0.1279 ± 0.00094
NAdam	0.1237 ± 0.00280	0.1255 ± 0.00317
NAdamW	0.1226 ± 0.00059	0.1237 ± 0.00003
Nesterov	0.1296 ± 0.00173	0.1298 ± 0.00153
VeLO	0.1232 ± 0.00039	0.1240 ± 0.00005
VeLO Short	0.1236 ± 0.00024	0.1242 ± 0.00003
HPO	0.1219 ± 0.00085	0.1237 ± 0.00012

D.5 OGBG

Please note that on HPO results, 2 trials out of 3 were unstable, hence, the missing standard deviations.

Name	train/loss	train/mAP	validation/loss	validation/mAP
Adam	0.0165 ± 0.00045	76.3886 ± 1.51418	0.0515 ± 0.00021	27.3651 ± 0.18156
Heavy Ball	0.0329 ± 0.00012	31.9587 ± 0.42343	0.0461 ± 0.00022	23.0116 ± 0.01945
NAdam	0.0174 ± 0.00026	74.3218 ± 1.07672	0.0509 ± 0.00027	27.2395 ± 0.63613
NAdamW	0.0196 ± 0.00181	68.3208 ± 5.45994	0.0483 ± 0.00157	27.6925 ± 0.23413
Nesterov	0.0323 ± 0.00040	33.1559 ± 0.62190	0.0458 ± 0.00024	23.6963 ± 0.38710
VeLO	0.0164 ± 0.00037	76.6425 ± 0.92113	0.0510 ± 0.00017	27.4374 ± 0.30907
VeLO Short	0.0180 ± 0.00058	72.5836 ± 0.96198	0.0491 ± 0.00064	28.2645 ± 0.35253
HPO	0.0205 ± nan	58.8443 ± nan	0.0463 ± nan	28.9687 ± nan

E Step-Controlled Experiments Results

E.1 Steps-To-Result Measurements

Table 11: Number of steps to achieve the train target performance. These numbers are used to plot the performance profiles in figure 5. A value of inf indicates that that baseline was unable to achieve the target within the maximum allowed runtime.

Workload Optimizer	Criteo	FastMRI	ImageNet	OGBG
Adam	**3497**	**21262**	**102029**	**47419**
HeavyBall	inf	inf	inf	inf
NAdam	4589	inf	121571	inf
NAdamW	4694	inf	117460	inf
Nesterov	inf	inf	inf	inf
VeLO	5252	inf	inf	71649
VeLO Short	4057	inf	inf	inf

Figure 5: Steps-to-Target performance profiles of baselines vs VeLO and VeLO with 75% (VeLO Short) prompt on training targets.

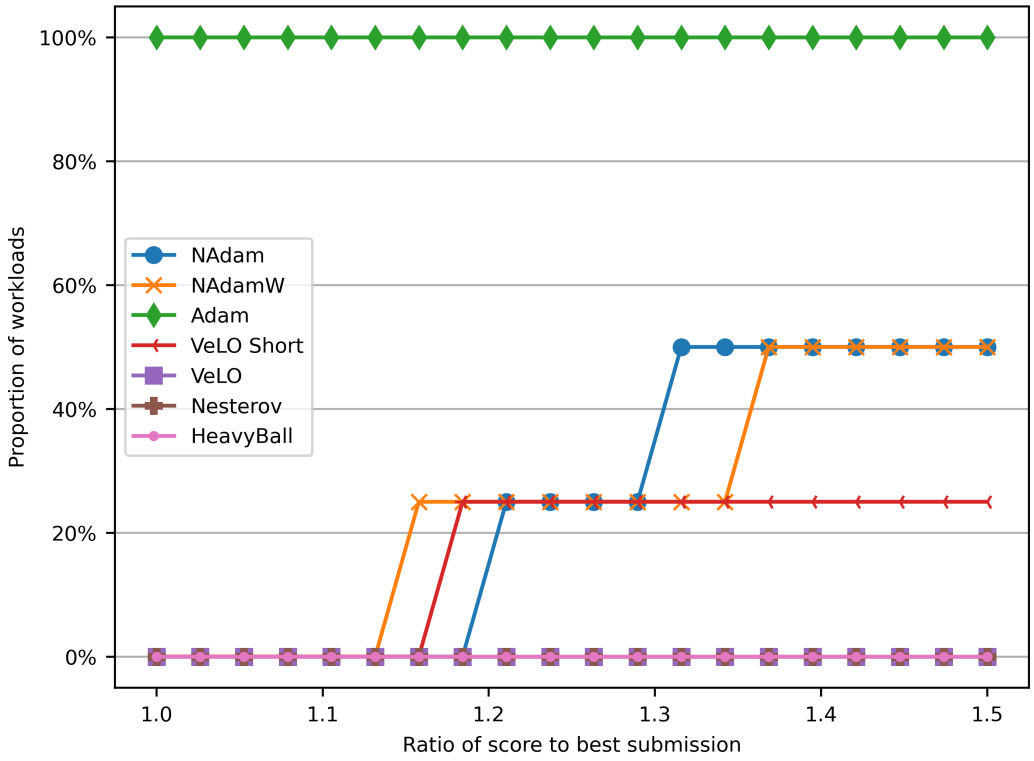


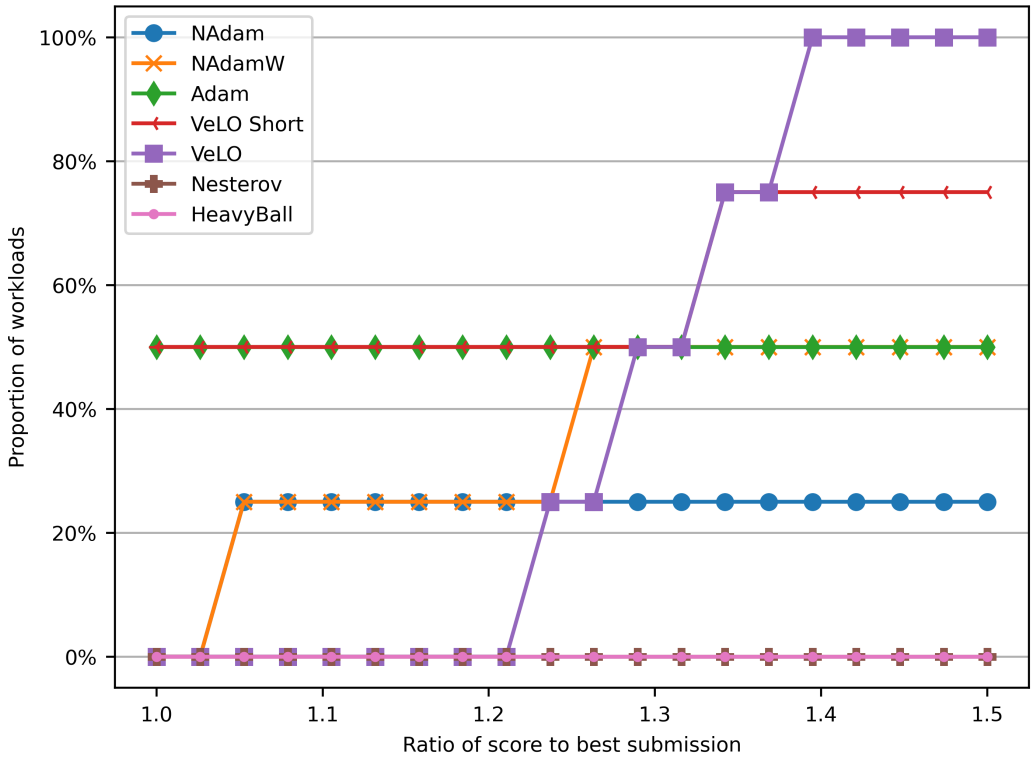
Table 12: Number of steps to achieve the validation target performance. These numbers are used to plot the performance profiles in figure 6. A value of inf indicates that that baseline was unable to achieve the target within the maximum allowed runtime.

Workload Optimizer	Criteo	FastMRI	ImageNet	OGBG
Adam	**5025**	**13344**	inf	inf
HeavyBall	inf	inf	inf	inf
NAdam	5177	inf	inf	inf
NAdamW	5277	inf	**112463**	inf
Nesterov	inf	inf	inf	inf
VeLO	6105	17649	114586	46979
VeLO Short	inf	17616	89419	34078

E.2 ImageNet

Name	train/loss	train/accuracy	validation/loss	validation/accuracy
Adam	0.0470 ± 0.00587	98.6554 ± 0.22522	1.9438 ± 0.00717	69.8487 ± 0.05294
Heavy Ball	0.2582 ± 0.02502	92.2350 ± 0.80171	1.6870 ± 0.02763	66.2940 ± 0.31674
NAdam	0.0571 ± 0.00060	98.3279 ± 0.04514	1.9528 ± 0.00120	69.9993 ± 0.11780
NAdamW	0.0470 ± 0.00106	98.7786 ± 0.04481	1.6345 ± 0.01098	71.4537 ± 0.07961
Nesterov	0.2512 ± 0.02606	92.4685 ± 0.86639	1.7022 ± 0.03822	66.3390 ± 0.35794
VeLO	0.1046 ± 0.00342	96.8478 ± 0.11937	1.5017 ± 0.02693	72.9120 ± 0.09714
HPO	0.5542 ± 0.00117	91.5016 ± 0.06893	1.1154 ± 0.00098	77.4423 ± 0.08693

Figure 6: Steps-to-Target performance profiles of baselines vs VeLO and VeLO with 75% (VeLO Short) prompt on validation targets.



E.3 FastMRI

Name	train/loss	train/ssim	validation/loss	validation/ssim
Adam	0.2702 ± 0.00693	74.2438 ± 0.51237	0.2850 ± 0.00003	72.6131 ± 0.01279
Heavy Ball	0.2809 ± 0.00282	72.8451 ± 0.14562	0.2899 ± 0.00008	71.9194 ± 0.02081
NAdam	0.2692 ± 0.00396	74.3011 ± 0.49641	0.2851 ± 0.00022	72.5980 ± 0.04064
NAdamW	0.2750 ± 0.00102	73.4640 ± 0.09133	0.2851 ± 0.00016	72.5903 ± 0.04674
Nesterov	0.2811 ± 0.00505	72.9825 ± 0.45841	0.2899 ± 0.00004	71.9274 ± 0.00075
VeLO	0.2804 ± 0.00008	73.5760 ± 0.01010	0.2851 ± 0.00022	72.6630 ± 0.02184
HPO	0.2716 ± 0.00381	74.2566 ± 0.29555	0.2850 ± 0.00080	72.6028 ± 0.13814

E.4 Criteo

Name	train/loss	validation/loss
Adam	0.1225 ± 0.00015	0.1237 ± 0.00005
Heavy Ball	0.1293 ± 0.00045	0.1299 ± 0.00062
NAdam	0.1239 ± 0.00302	0.1256 ± 0.00315
NAdamW	0.1220 ± 0.00018	0.1237 ± 0.00005
Nesterov	0.1301 ± 0.00096	0.1305 ± 0.00067
VeLO	0.1229 ± 0.00034	0.1240 ± 0.00008
HPO	0.1222 ± 0.00058	0.1238 ± 0.00022

E.5 OGBG

Name	train/loss	train/mAP	validation/loss	validation/mAP
Adam	0.0165 ± 0.00033	76.2472 ± 0.74703	0.0515 ± 0.00020	27.3603 ± 0.18697
Heavy Ball	0.0341 ± 0.00026	29.8791 ± 0.24316	0.0466 ± 0.00038	22.3993 ± 0.03845
NAdam	0.0173 ± 0.00033	74.3650 ± 0.94344	0.0509 ± 0.00027	27.2383 ± 0.63573
NAdamW	0.0197 ± 0.00196	68.1042 ± 5.87819	0.0483 ± 0.00157	27.6927 ± 0.23119
Nesterov	0.0331 ± 0.00008	31.8606 ± 0.30560	0.0462 ± 0.00027	22.9570 ± 0.22694
VeLO	0.0153 ± 0.00088	79.4321 ± 1.70572	0.0522 ± 0.00126	27.5886 ± 0.47789
HPO	4.1961 ± 3.61664	$61.3348 \pm \text{nan}$	4.3632 ± 3.73861	$28.9854 \pm \text{nan}$

F Current vs Original VeLO Evaluation Protocol

The original VeLO evaluation codebase was not released. Nevertheless, the authors do evaluate on a specific version of the MLCommons benchmark that precedes the latest one used in this study [8]. The workloads they chose were:

1. Resnet-50 on ImageNet
2. ViT on Imagenet
3. Transformer on WMT
4. Deepspeech on Librespeech
5. Conformer on Librespeech
6. GNN on OGBG

Meanwhile, we choose the workloads to be as close as possible to the training distribution:

1. Resnet-50 on ImageNet
2. U-Net on FastMRI
3. GNN on OGBG
4. DLRM on Criteo-1TB

Please note that ResNets and Autoencoders (U-Net like architectures) were seen heavily during training. Moreover, GNNs and DLRMs are implemented as MLPs which are also seen heavily during training. ImageNet was part of the training datasets employed too. It is unclear whether the magnitude, sparsity and other properties of the gradients on the above workloads deviate heavily from the totality of meta-training tasks sampled.

Beside the choice of workloads, the original paper have trained (1) VeLO and (2) 20 hyperparameter trials of Adam for a fixed number of steps. Results are presented as figures of training curves which paints comparisons imprecise. Meanwhile, our evaluation uses the MLCommons recommended total wall-clock time to run for and an approximately equivalent total number of steps across a wider set of optimizers. We also fix the optimizers hyerparameters to default values chosen one-shot by the authors to stay faithful to the self-tuning regime instead of sampling for a hyperparameter search space tuned to target validation performance. To paint a more detailed picture and stronger conclusions, we report all final results across step and wall-clock time quotas, and detailed raw data for performance profiles. All in all, our protocol shows that VeLO's efficacy is invalidated once the results are stress-tested in a fair setup.