# Continually Adapting Optimizers Improve Meta-Generalization

**Wenyi Wang**
AI Initiative, King Abdullah University of Science and Technology
`wenyi.wang@kaust.edu.sa`

**Louis Kirsch**
The Swiss AI Lab IDSIA, USI, SUPSI
`louis@idsia.ch`

**Francesco Faccio**
The Swiss AI Lab IDSIA, USI, SUPSI
AI Initiative, King Abdullah University of Science and Technology
`francesco@idsia.ch`

**Mingchen Zhuge**
AI Initiative, King Abdullah University of Science and Technology
`mingchen.zhuge@kaust.edu.sa`

**Jürgen Schmidhuber**
The Swiss AI Lab IDSIA, USI, SUPSI
AI Initiative, King Abdullah University of Science and Technology
`juergen.schmidhuber@kaust.edu.sa`

## Abstract

Meta-learned optimizers increasingly outperform analytical hand-crafted optimizers such as SGD and Adam. However, on some tasks, they fail to generalize strongly, underperforming hand-crafted methods. Then one can fall back on hand-crafted methods through a *guard*, to combine the efficiency benefits of learned optimizers and the guarantees of analytical methods. At some point in the iterative optimization process, however, such guards may make the learned optimizer incompatible with the remaining optimization, and thus useless for further progress. Our novel method *Meta Guard* continues to adapt the learned optimizer to the target optimization problem. It experimentally outperforms other baselines, adapting to new tasks during training.

## 1 Introduction

Neural networks can be used as function approximators to solve a wide range of machine learning problems, including regression, classification, and sequence modelling [24]. Instead of optimizing the parameters of the neural network using hand-crafted algorithms such as stochastic gradient descent (SGD) [1], these neural networks can be trained to implement the learning algorithms in their activation updates to learn how to learn [21, 8], potentially outperforming hand-crafted learning algorithms [12]. Recently, such meta-learning approaches have been successful in Large Language Models (LLMs) and related Transformer-based models under the name of in-context learning [3, 11].

Given a sufficiently universal architecture, such as LSTMs [7], standard Transformers [26] that scale quadratically with sequence length, or Linear Transformers / Fast Weight Programmers [22, 10, 20], these neural networks can, in principle, not only implement new learning algorithms but also meta-learn, meta-meta-learn and so on [23, 9, 13]. Unfortunately, we have no guarantees that such networks are sufficiently robust to generalize to unseen problems that were not part of meta-training. A method that ensures that these universal models continue to improve without divergence is desired.

In this paper, we focus on the setting of learned optimization (L2O) [2, 16, 15]. A learned optimizer implemented by a neural network $g_\theta$ with parameters $\theta$ incrementally updates (optimizes) the optimizee with parameters $\eta$. The optimizee is a neural network to solve any given task, for example, an MNIST [4] classifier. L2O usually occurs in two phases: First, during meta-training, the optimizer $g_\theta$ is trained to optimize well on a distribution of tasks, i.e., for each task, producing $\eta_T$ after $T$ steps of optimization with low loss values. Second, during meta-testing, the optimizer parameters $\theta$ are frozen, and the optimizer is applied to a new test-task. Doing well on any given test task requires that either the test task is very similar to previously trained-on tasks or that the optimizer generalizes well at the meta-test time. In practice, strong generalization is difficult to achieve [17, 12]. An approach to address this issue is to undo underperforming self-modifications using the success story algorithm [25] or to allocate more computational resources to better-performing solutions [13]. Instead, we propose to continue adjusting the meta-learner during meta-testing when its performance falls below a known hand-crafted optimizer such as SGD. We refer to this as the 'Meta Guard'. This leverages the efficiency gains of a learned optimizer while ensuring that it is continuously adapted to the meta-test task when it fails to generalize.

Inspired by Loss-Guarded L2O [18] (LGL2O), we propose two guard mechanisms (Alg. 1 and Alg. 2) that select between analytical updates and learned updates to both the optimizer and optimizee parameters with respect to a meta objective that evaluates how well the learned optimizer performs in optimizing the current optimizee. We theoretically justify their convergence and empirically show their adaptation to novel tasks.

This paper is organized as follows. We discuss learned optimizers and loss guards in the background section 2. Section 3 introduces our *Meta Guard*, a learned optimizer that continually adapts at meta-test time. Then, in section 4, we empirically demonstrate that our proposed method outperforms other baselines, including non-adaptive guards, in various problem settings. Finally, we conclude the paper in Section 4 with a summary.

## 2 Background

**Learning to Optimize** Let $g_\theta$ be a learned optimizer parameterized by $\theta \in \Theta \subset \mathbb{R}^{n_\theta}$. Let $\eta \in \mathbb{R}^m$ represent the optimizee parameters, for example, the parameters of a convolutional neural network (CNN) [5, 14] that classifies handwritten digit images. Given a loss function $l : \mathbb{R}^m \to \mathbb{R}$ that measures the performance of the optimizee on specific data, our objective is to meta-learn optimizers $g_\theta$ that minimize the loss $l(\eta_t)$ of the optimizee parameters $\eta_t$ at each iteration $t$ of the optimization process. The learned optimizer is expressed as a function

$$\eta_t, h_t \leftarrow g_\theta(\eta_{t-1}, h_{t-1}, \sigma(\eta_{t-1})), \tag{1}$$

and is applied at every iteration $t$ for $t \in 0, \dots, T$. The function $\sigma(\eta_{t-1})$ includes auxiliary variables regarding the previous optimizee parameters, such as the loss and gradient, denoted as $\sigma(\eta_{t-1}) = (l(\eta_{t-1}), \nabla_\eta l(\eta)|_{\eta=\eta_{t-1}})$. Here, $h_t$ represents a memory component, e.g., the hidden state of a long short-term memory (LSTM) neural network [7, 6]. In principle, this allows the network to adjust its optimization behavior to the problem, performing meta-meta-learning [13]. For notational simplicity, the initial memory state $h_0$ is included in the parameter vector $\theta$, with $\theta_t$ containing the updated $h_t$ instead of $h_0$ after $t$ iterations of $g_\theta$. We then further simplify the notation to $\eta_t, \theta_t \leftarrow g(\eta_{t-1}, \theta_{t-1})$, with the implicit input $\sigma(\eta_{t-1})$. Unrolling the learned optimizer for $T$ steps leads to $\eta_T, \theta_T \leftarrow g^T(\eta_0, \theta_0)$, where $g^T$ denotes $T$ recurrent applications of the learned optimizer $g$.

We parameterize the learned optimizer $g_\theta$ using an LSTM, which processes each element of the optimizee parameter vector $\eta$ coordinate-wise (separately for each parameter scalar $\eta_i$), as shown in Andrychowicz et al.[2]. Starting with a unique $\theta_0$, this results in multiple variants $\theta_T^i$, where $i$ ranges from 1 to $m$, producing a variant for each coordinate of $\eta$. To reduce them back into a single $\theta_T$, we calculate the average as $\theta_T = \frac{1}{m}\sum_{i=1}^{m} \theta_T^i$. This averaging is done after $T$ steps as part of $g^T$.

**Meta-Training and Meta-Testing** In the context of learned optimizers, meta-training typically involves minimizing a meta-objective $L$ with respect to $\theta$:

$$L(\theta) = \mathbb{E}_{l \sim p(l), \eta_0 \sim p(\eta_0)} \left[ \frac{1}{T} \sum_{t=1}^{T} l(\eta_t) \right], \tag{2}$$

where the expectation is taken over a particular distribution of tasks $p(l)$ and initial optimizee parameters $p(\eta_0)$. During meta-testing, the optimizer parameters $\theta$ remain frozen, and the optimizer is applied to a new test-task.

**Loss-Guarded L2O** A method related to ours is the Loss-Guarded Learned Optimizer (LGL2O) [18]. LGL2O employs a learned optimizer during meta-testing. However, if the performance of the learned optimizer falls short when compared to a hand-crafted optimizer such as SGD, it switches to using SGD. Unlike our approach, LGL2O does not continue adapting the learned optimizer; its parameters remain fixed during meta-testing. Our paper illustrates that continuing to adapt the learned optimizer during its meta-test time allows for exploiting regularities of the target problem that further accelerates learning on the problem. This is related to online meta-gradient approaches, which always adapt hyper-parameters [29] or learned loss functions [28] continually throughout meta-testing but do not employ a guard.

## 3 Learned Optimizers That Continually Adapt

In the following, we propose learned optimizers that continually adapt. We first meta-train a learned optimizer with a regular meta-training phase as in Equation 2 on a task distribution. Next, we select an optimization task different from this training distribution, and run our learned optimizer in meta-test mode. Unlike previous work, we continue to adapt the learned optimizer by introducing a *Meta Guard*. This guard operates the learned optimizer as usual, but switches to adapting both the optimizer and the optimizee if the performance drops below the gradient descent baseline. To adapt the optimizer and optimizee, we introduce a meta-objective which is optimized during meta-testing:

$$L^{\text{Adapt}}(\eta, \theta) = l(\eta_T), \tag{3}$$

$$\text{with} \quad \eta_T, \theta_T = g^T(\eta, \theta). \tag{4}$$

Note that unlike in Equation 2, here we are only optimizing on for the current meta-test task $l$ starting from the current optimizee parameter $\eta$ instead of using a meta-training distribution. Given the current $\eta$ and $\theta$, the fallback adaptation step then becomes

$$\eta^{\text{fallback}}, \theta^{\text{fallback}} \leftarrow (\eta, \theta) - \lambda \nabla_{(\eta,\theta)} L^{\text{Adapt}}(\eta, \theta), \tag{5}$$

for a specified learning rate $\lambda$.

**Simple Meta Guard (SMG)** We begin by introducing a simple implementation of this Meta Guard mechanism in Algorithm 1. The learned optimizer $g_\theta$ and the fallback from Equation 5 are applied to the current optimizer $\theta$ and optimizee $\eta$. This results in the learned optimizer solution $\eta^{\text{learned}}, \theta^{\text{learned}}$ and the adapted fallback solution $\eta^{\text{fallback}}, \theta^{\text{fallback}}$. The new optimizers and optimizees are then evaluated through two other unrolls using the meta-objectives $L^{\text{Adapt}}(\eta^{\text{learned}}, \theta^{\text{learned}})$ and $L^{\text{Adapt}}(\eta^{\text{fallback}}, \theta^{\text{fallback}})$. Based on the solution with the lowest meta-objective, we select as optimizer and optimizee either the learned optimizer solution $\eta^{\text{learned}}, \theta^{\text{learned}}$ or the adapted fallback $\eta^{\text{fallback}}, \theta^{\text{fallback}}$.

**Accelerated Meta Guard (AMG)** The meta-objective in SMG uses the performance of the learned optimizer after $T$ steps to assign credit to the current optimizer $g_\theta$ and the optimizee $\eta$. Unfortunately, using information about the future can be computationally expensive, as it requires unrolling the learned optimizer when evaluating either the fallback update or the learned optimizer itself. However, such an evaluation is necessary to ensure that the chosen update is not worse than the fallback update in order to preserve the convergence property of the fallback algorithm. Indeed, if we use the current performance of the optimizee, setting $L^{\text{Adapt}}(\eta, \theta) = l(\eta)$, the parameters of the learned optimizer are only updated through self-modifications, e.g., by updates in the memory of the LSTM, and only when the performance of the learned optimizer is better than that of the hand-crafted learning algorithm. This case reverts to the standard LGL2O, where the learned optimizer cannot be optimized at test time. On the other hand, LGL2O benefits from immediate evaluation of the optimizee, which can save computational resources.

In order to get the best of both methods, we introduce Accelerated Meta Guard (AMG), a hybrid method that combines the meta-objectives of LGL2O and SMG. The new meta-objective for AMG (replacing $L^{\text{Adapt}}$) is defined as:

$$L^{\text{AMG}}(\eta, \theta) = \min(l(\eta), l(\eta_T)), \tag{6}$$

$$\text{where} \quad \eta_T, \theta_T = g^T(\eta, \theta). \tag{7}$$

Taking the minimum of the two objectives guarantees that the resulting update should have a lower optimizee loss than that of LGL2O and SMG. This results in Algorithm 2. Like in SMG, the learned optimizer $g_\theta$ and the fallback from Equation 5 are applied to the current optimizer $\theta$ and optimizee $\eta$. The learned optimizer solution $\eta^{\text{learned}}$ and $\theta^{\text{learned}}$ is computed as in SMG. Unlike SMG, the fallback for the new objective $L^{\text{AMG}}$ assumes two expressions, depending on the minimum inside the meta-objective. When $l(\eta) \leq l(\eta_T)$, then the fallback is the one of LGL2O with $\eta^{\text{fallback}} \leftarrow \eta - \lambda \nabla_\eta l(\eta)$ and $\theta^{\text{fallback}} \leftarrow \theta$ remaining constant. When $l(\eta) \geq l(\eta_T)$, the fallback is the same as in SMG. We compute another unroll of the learned optimizer to evaluate the new optimizers and optimizees using the meta-objective of AMG. However, for the fallback, we must evaluate $L^{\text{AMG}}(\eta^{\text{fallback}}, \theta^{\text{fallback}}) = \min(l(\eta^{\text{fallback}}), l({\eta_T}^{\text{fallback}}))$. Formally, we compute ${\eta_T}^{\text{fallback}}, {\theta_T}^{\text{fallback}} \leftarrow g^T(\eta^{\text{fallback}}, \theta^{\text{fallback}})$. Similarly, for the learned optimizer solutions $L^{\text{AMG}}(\eta^{\text{learned}}, \theta^{\text{learned}})$, we would need to compute ${\eta_T}^{\text{learned}}, {\theta_T}^{\text{learned}} \leftarrow g^T(\eta^{\text{learned}}, \theta^{\text{learned}})$.

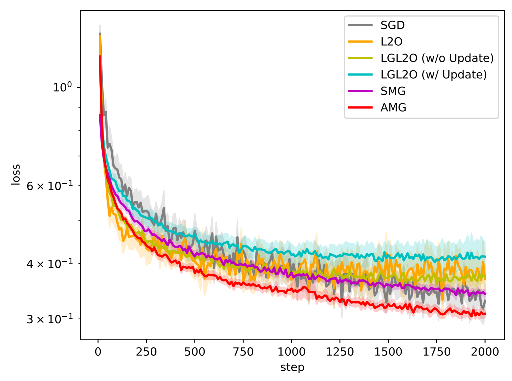
To trade off computational cost versus accurate evaluation, we evaluate this quantity using an upper bound of $L^{\text{AMG}}(\eta^{\text{learned}}, \theta^{\text{learned}})$. Since $L^{\text{AMG}}(\eta^{\text{learned}}, \theta^{\text{learned}}) \leq l(\eta^{\text{learned}})$, we use $l(\eta^{\text{learned}})$ as a proxy to evaluate the meta-objective after applying the learned optimizer. Note that since we compute $L^{\text{AMG}}(\eta^{\text{fallback}}, \theta^{\text{fallback}})$ to evaluate the fallback, we now have access to $l({\eta_T}^{\text{fallback}})$, which is an upper bound for $L^{\text{AMG}}({\eta_T}^{\text{fallback}}, {\theta_T}^{\text{fallback}})$. Following the same argument as above, we can implement a guard check consisting of four elements: $L^{\text{AMG}}(\eta^{\text{fallback}}, \theta^{\text{fallback}})$, $L^{\text{AMG}}(\eta, \theta)$, $L^{\text{AMG}}(\eta^{\text{learned}}, \theta^{\text{learned}})$, and $L^{\text{AMG}}({\eta_T}^{\text{fallback}}, {\theta_T}^{\text{fallback}})$, with the last two approximated using their corresponding upper bounds. Therefore, we determine the minimum of $\min(L^{\text{AMG}}(\eta^{\text{fallback}}, \theta^{\text{fallback}}), L^{\text{AMG}}(\eta, \theta), L^{\text{AMG}}(\eta^{\text{learned}}, \theta^{\text{learned}}), L^{\text{AMG}}({\eta_T}^{\text{fallback}}, {\theta_T}^{\text{fallback}}))$ $\leq \min(l(\eta^{\text{fallback}}), l({\eta_T}^{\text{fallback}}), l(\eta), l(\eta^{\text{learned}}))$ , to decide which update to take.

**Convergence Results** Our algorithm AMG inherits the convergence properties of LGL2O [18]. Under the assumptions that the loss function $l(\cdot)$ is continuous, $\mu$-strongly convex, and L-smooth, given a constant learning rate $0 < \lambda < \min(\frac{2}{L}, 2\mu)$, we can guarantee that the sequence of optimizees $\eta_i$ generated by AMG will converge to the global optimum: $\lim_{i \to \infty} l(\eta_i) = \min_\eta l(\eta) = \min_{\eta,\theta} L^{\text{AMG}}(\eta, \theta)$. The proof is straightforward and closely follows Theorem 1 in LGL2O [18]. In particular, in their Proposition 2, when $\eta_{i+1}$ is not chosen by the gradient descent update with respect to $l(\eta_i)$, we still have that $l(\eta_{i+1}) < l(\eta_i - \lambda \nabla_{(\eta)} l(\eta_i))$. In SMG, we assume that the loss $L^{\text{Adapt}}(\eta, \theta)$ is continuous, $\mu$-strongly convex, and L-smooth, with respect to the vector $[\eta, \theta]$. Given a constant learning rate $0 < \lambda < \min(\frac{2}{L}, 2\mu)$, one can apply Theorem 1 from LGL2O [18]. This implies that $\lim_{i \to \infty} L^{\text{Adapt}}(\eta_i, \theta_i) = \min_{(\eta,\theta)} L^{\text{Adapt}}(\eta, \theta)$.

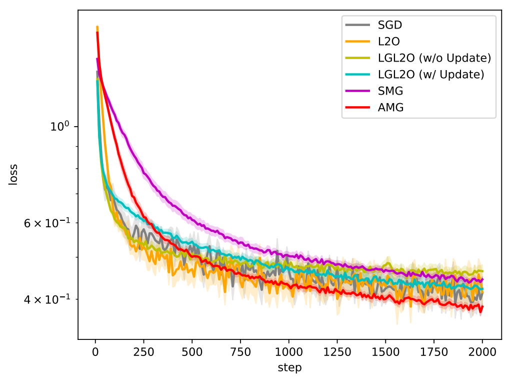
## 4 Experiments

**Tasks.** We compare our proposed SMG and AMG on various different meta-train and meta-test settings. For each of the experiments, the optimizee is either a multilayer perceptron (MLP) [19] with one hidden layer and 20 hidden neurons or a 3-layer CNN [5]. The target problem is either the MNIST [4] or the FashionMNIST [27]. To test the generalization capabilities of our learned optimizers, we test four different settings in Figure 1: (1) Seen data and optimizee, (2) seen data and unseen optimizee, (3) unseen data seen optimizee, and (4) unseen data and unseen optimizee. We use mini-batches for learned optimizer and fallback gradient descent updates.

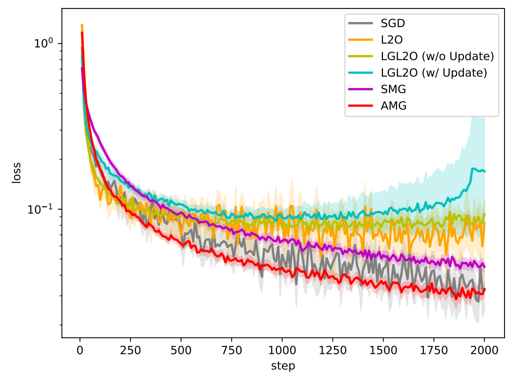
**Baselines.** For the baselines, we use (1) only SGD, (2) only the learned optimizer (L2O), and (3) the LGL2O loss guard. Two versions of LGL2O are tested (LGL20 (w/ Update) and LGL2O (w/o



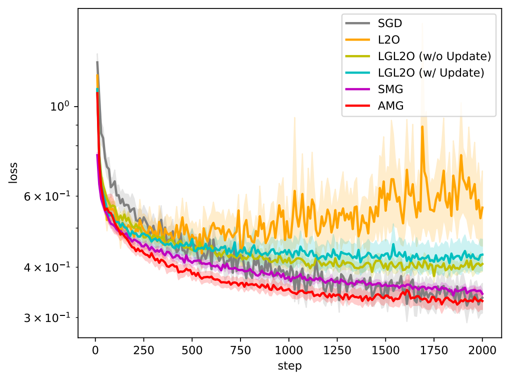
(a) Train: CNN & FashionMNIST, Test: CNN & FashionMNIST



(b) Train: CNN & FashionMNIST, Test: MLP & FashionMNIST



(c) Train: CNN & FashionMNIST, Test: CNN & MNIST



(d) Train: MLP & MNIST, Test: CNN & FashionMNIST

Figure 1: **Our adapting learned optimizers (SMG & AMG) generalize to various problem settings.** We observe that the baselines L2O and two versions of LGL2O either worsen or improve very slowly after 1000 optimization steps. SGD keeps improving, however, initially significantly slower than the learned optimizer. SMG does not prematurely converge, but is usually slower than SGD. AMG outperforms the baselines in most cases. Each loss is averaged over 25 seeds with 95% confidence intervals.

Update)). The former carries over the hidden state between iterations, and the latter reverts back to the initial hidden state.

**Observations.** From Figure 1 we observe that standard L2O initially outperforms SGD, particularly in the first 200 steps, but then diverges or stops optimizing as training progresses, which is a typical issue for L2O. The LGL2O baselines are initially efficient, but the simple loss guard seems to be ineffective after 1000 optimization steps. In contrast, our SMG continuously improves the loss but is usually slower than SGD. Our AMG further improves on these results, outperforming all baselines in most cases.

**Conclusion** Learned optimizers do not always generalize to meta-test tasks significantly different from those in the meta-training distribution. To address this problem, our continually adapting learned optimizer *Meta Guard* tests whether the self-updating learned optimizer falls below the performance of a hand-crafted learning algorithm. If so, it explicitly adapts both the learned optimizer as well as the inner optimizer via gradient descent. We demonstrate that this combines the efficiency benefits of learned optimizers with the robustness of a hand-crafted ones. Our Accelerated Meta Guard performs favourably compared to SGD, standard learned optimizers, and non-adaptive loss guards.

## Acknowledgement

We thanks Piotr Piękos for his review and suggestions for our paper. This work was partially supported by the European Research Council (ERC, Advanced Grant Number 742870) and the Swiss National Science Foundation (SNF, Grant Number 200021 192356).

## References

[1] S. Amari. A theory of adaptive pattern classifiers. *IEEE Transactions on Electronic Computers*, (3):299–307, 1967.

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

[3] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[4] L. Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[5] K. Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological cybernetics*, 36(4):193–202, 1980.

[6] F. A. Gers, J. Schmidhuber, and F. Cummins. Learning to forget: Continual prediction with lstm. *Neural computation*, 12(10):2451–2471, 2000.

[7] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[8] S. Hochreiter, A. S. Younger, and P. R. Conwell. Learning to learn using gradient descent. In *International Conference on Artificial Neural Networks*, pages 87–94. Springer, 2001.

[9] K. Irie, I. Schlag, R. Csordás, and J. Schmidhuber. A modern self-referential weight matrix that learns to modify itself. In *International Conference on Machine Learning*, pages 9660–9677. PMLR, 2022.

[10] A. Katharopoulos, A. Vyas, N. Pappas, and F. Fleuret. Transformers are rnns: Fast autoregressive transformers with linear attention. In *International conference on machine learning*, pages 5156–5165. PMLR, 2020.

[11] L. Kirsch, J. Harrison, J. Sohl-Dickstein, and L. Metz. General-purpose in-context learning by meta-learning transformers. *arXiv preprint arXiv:2212.04458*, 2022.

[12] L. Kirsch and J. Schmidhuber. Meta learning backpropagation and improving it. *Advances in Neural Information Processing Systems*, 34:14122–14134, 2021.

[13] L. Kirsch and J. Schmidhuber. Eliminating meta optimization through self-referential meta learning. *arXiv preprint arXiv:2212.14392*, 2022.

[14] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[15] L. Metz, C. D. Freeman, N. Maheswaranathan, and J. Sohl-Dickstein. Training learned optimizers with randomly initialized learned optimizers. *arXiv preprint arXiv:2101.07367*, 2021.

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

[17] L. Metz, N. Maheswaranathan, R. Sun, C. D. Freeman, B. Poole, and J. Sohl-Dickstein. Using a thousand optimization tasks to learn hyperparameter search strategies. *arXiv preprint arXiv:2002.11887*, 2020.

[18] I. Prémont-Schwarz, J. Vítků, and J. Feyereisl. A simple guard for learned optimizers. In K. Chaudhuri, S. Jegelka, L. Song, C. Szepesvari, G. Niu, and S. Sabato, editors, *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pages 17910–17925. PMLR, 17–23 Jul 2022.

[19] F. Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[20] I. Schlag, K. Irie, and J. Schmidhuber. Linear transformers are secretly fast weight programmers. In *International Conference on Machine Learning*, pages 9355–9366. PMLR, 2021.

[21] J. Schmidhuber. *Evolutionary principles in self-referential learning, or on learning how to learn: the meta-meta-... hook.* PhD thesis, Technische Universität München, 1987.

[22] J. Schmidhuber. Reducing the ratio between learning complexity and number of time varying variables in fully recurrent nets. In *International Conference on Artificial Neural Networks*, pages 460–463. Springer, 1993.

[23] J. Schmidhuber. A 'self-referential'weight matrix. In *International conference on artificial neural networks*, pages 446–450. Springer, 1993.

[24] J. Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[25] J. Schmidhuber, J. Zhao, and M. Wiering. Shifting inductive bias with success-story algorithm, adaptive levin search, and incremental self-improvement. *Machine Learning*, 28(1):105–130, 1997.

[26] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[27] H. Xiao, K. Rasul, and R. Vollgraf. Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms. *CoRR*, abs/1708.0, 2017.

[28] Z. Xu, H. P. van Hasselt, M. Hessel, J. Oh, S. Singh, and D. Silver. Meta-gradient reinforcement learning with an objective discovered online. *Advances in Neural Information Processing Systems*, 33:15254–15264, 2020.

[29] Z. Xu, H. P. van Hasselt, and D. Silver. Meta-gradient reinforcement learning. *Advances in neural information processing systems*, 31, 2018.

## A Algorithms

**Algorithm 1** Simple Meta Guard

```
Require: Initial optimizee parameters $\eta$, learned optimizer parameters $\theta$
  while termination condition not satisfied do
    $\eta^{\text{learned}}, \theta^{\text{learned}} \leftarrow g^T(\eta, \theta)$ {Apply learned optimizer}
    $\eta^{\text{fallback}}, \theta^{\text{fallback}} \leftarrow (\eta, \theta) - \lambda \nabla_{(\eta,\theta)} L^{\text{Adapt}}(\eta, \theta)$ {Fallback adaptation}
    if $L^{\text{Adapt}}(\eta^{\text{learned}}, \theta^{\text{learned}}) > L^{\text{Adapt}}(\eta^{\text{fallback}}, \theta^{\text{fallback}})$ then
      $\eta, \theta \leftarrow \eta^{\text{fallback}}, \theta^{\text{fallback}}$
    else
      $\eta, \theta \leftarrow \eta^{\text{learned}}, \theta^{\text{learned}}$
    end if
  end while
```

**Algorithm 2** Accelerated Meta Guard

```
Require: Initial optimizee parameters $\eta$, learned optimizer parameters $\theta$
  while termination condition not satisfied do
    $\eta^{\text{learned}}, \theta^{\text{learned}} \leftarrow g^T(\eta, \theta)$ {Apply learned optimizer}
    $\eta^{\text{fallback}}, \theta^{\text{fallback}} \leftarrow \eta - \lambda \nabla_{(\eta,\theta)} \min\{l(\eta), L^{\text{Adapt}}(\eta, \theta)\}$ {Fallback adaptation}
    ${\eta_T}^{\text{fallback}}, {\theta_T}^{\text{fallback}} \leftarrow g^T(\eta^{\text{fallback}}, \theta^{\text{fallback}})$
                                {Apply the fallback updated learned optimizer}
    if $l(\eta^{\text{learned}}) = \min\{l(\eta), l(\eta^{\text{learned}}), l(\eta^{\text{fallback}}), l({\eta_T}^{\text{fallback}})\}$ then
      $\eta, \theta \leftarrow \eta^{\text{learned}}, \theta^{\text{learned}}$
    else if $l(\eta^{\text{fallback}}) = \min\{l(\eta), l(\eta^{\text{learned}}), l(\eta^{\text{fallback}}), l({\eta_T}^{\text{fallback}})\}$ then
      $\eta, \theta \leftarrow \eta^{\text{fallback}}, \theta^{\text{fallback}}$
    else if $l({\eta_T}^{\text{fallback}}) = \min\{l(\eta), l(\eta^{\text{learned}}), l(\eta^{\text{fallback}}), l({\eta_T}^{\text{fallback}})\}$ then
      $\eta, \theta \leftarrow {\eta_T}^{\text{fallback}}, {\theta_T}^{\text{fallback}}$
    else
      $\eta, \theta \leftarrow \eta, \theta$
    end if
  end while
```

## B Experimental details

### B.1 Hyper-parameters

Our fallback SGD uses two different learning rates when applied to learned optimizers and optimizees. The learning rate on the optimizee is set to $\lambda_\eta = 3$, (following [18]). The learned optimizer adaptation learning rate is set to $\lambda_\theta = 0.1$. The learned optimizers are meta-trained using Adam with its default parameters in PyTorch 2.0.0, e.g., learning rate equal to $0.001$. The CNN architecture is adopted from [18] so that the "number of channels = (8, 16, 32), kernel sizes = (5, 3, 3) and strides = (2, 2, 2), with a fully connected final layer". We use mini-batches of size 128. $T$ is set to be 10 for Algorithm 1 and Algorithm 2.