Learning to Optimize for Reinforcement Learning

Qingfeng Lan [*]
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada
qlan3@ualberta.ca

A. Rupam Mahmood
Department of Computing Science
University of Alberta
CIFAR AI Chair, Amii
armahmood@ualberta.ca

Shuicheng Yan
Sea AI Lab, Skywork AI
shuicheng.yan@gmail.com

Zhongwen Xu
Sea AI Lab, Tencent AI Lab
zhongwen.s.xu@gmail.com

Abstract

In recent years, by leveraging more data, computation, and diverse tasks, learned optimizers have achieved remarkable success in supervised learning, outperforming classical hand-designed optimizers. Reinforcement learning (RL) is essentially different from supervised learning, and in practice, these learned optimizers do not work well even in simple RL tasks. We investigate this phenomenon and identify two issues. First, the agent-gradient distribution is non-independent and identically distributed, leading to inefficient meta-training. Moreover, due to highly stochastic agent-environment interactions, the agent-gradients have high bias and variance, which increases the difficulty of learning an optimizer for RL. We propose pipeline training and a novel optimizer structure with a good inductive bias to address these issues, making it possible to learn an optimizer for reinforcement learning from scratch. We show that, although only trained in toy tasks, our learned optimizer can generalize to unseen complex tasks in Brax. [1]

1 Introduction

Deep learning has achieved great success in many areas (LeCun et al., 2015), which is largely attributed to the automatically learned features that surpass handcrafted expert features. The use of gradient descent enables automatic adjustments of parameters within a model, yielding highly effective features. Despite these advancements, as another important component in deep learning, optimizers are still largely hand-designed and heavily reliant on expert knowledge. To reduce the burden of hand-designing optimizers, researchers propose to learn to optimize with the help of meta-learning (Sutton, 1992; Andrychowicz et al., 2016; Chen et al., 2017; Wichrowska et al., 2017; Maheswaranathan et al., 2021). Compared to designing optimizers with human expert knowledge, learning an optimizer is a data-driven approach, reducing the reliance on expert knowledge. During training, a learned optimizer can be optimized to speed learning and help achieve better performance.

Despite the significant progress in learning optimizers, previous works only present learned optimizers for supervised learning (SL). These learned optimizers usually have complex neural network structures and incorporate numerous human-designed input features, requiring a large amount of computation and human effort to design and train them. Moreover, they have been shown to perform poorly in reinforcement learning (RL) tasks (Metz et al., 2020b; 2022b). *Learning to optimize for RL remains an open and challenging problem.*

---

[*]Work was done during an internship at Sea AI Lab, Singapore.

[1]The code is available at https://github.com/sail-sg/optim4rl/.

Classical optimizers are typically designed for optimization in SL tasks and then applied to RL tasks. However, RL tasks possess unique properties that are largely overlooked by classical optimizers. For example, unlike SL, the input distribution of an RL agent is non-stationary and non-independent and identically distributed (non-iid) due to locally correlated transition dynamics (Alt et al., 2019). Additionally, due to policy and value iterations, the target function and the loss landscapes in RL are constantly changing throughout the learning process, resulting in a much more unstable and complex optimization process. In some cases, these properties also make it inappropriate to apply optimization algorithms designed for SL to RL directly, such as stale accumulated gradients (Bengio et al., 2020a) or unique interference-generalization phenomenon (Bengio et al., 2020b). *We still lack optimizers specifically designed for RL tasks.*

In this work, we aim to learn optimizers for RL. Instead of manually designing optimizers by studying RL optimization, we apply meta-learning to learn optimizers from data generated in the agent-environment interactions. We first investigate the problem and find that the complicated agent-gradient distribution impedes the training of learned optimizers for RL. Furthermore, the non-iid nature of the agent-gradient distribution also hinders meta-training. Lastly, the highly stochastic agent-environment interactions can lead to agent-gradients with high bias and variance, exacerbating the difficulty of learning an optimizer for RL. In response to these challenges, we propose a novel approach, *Optim4RL*, a learned optimizer for RL that involves pipeline training and a specialized optimizer structure of good inductive bias. Compared with previous methods, Optim4RL is more stable to train and more effective in optimizing RL tasks, without complex optimizer structures or numerous human-designed input features. We demonstrate that Optim4RL can learn to optimize RL tasks from scratch and generalize to unseen tasks. Our work is the first to propose a learned optimizer for deep RL tasks that works well in practice.

2 Background

2.1 Reinforcement Learning

The process of reinforcement learning (RL) can be formalized as a Markov decision process (MDP). Formally, let $M = (\mathcal{S}, \mathcal{A}, P, r, \gamma)$ be an MDP which includes a state space $\mathcal{S}$, an action space $\mathcal{A}$, a state transition probability function $P : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$, a reward function $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, and a discount factor $\gamma \in [0, 1)$. At each time-step t, the agent observes a state $S_t \in \mathcal{S}$ and samples an action A_t from the policy $\pi(\cdot|S_t)$. Then it observes the next state $S_{t+1} \in \mathcal{S}$ according to P and receives a scalar reward $R_{t+1} = r(S_t, A_t)$. The return is defined as the weighted sum of rewards, i.e., $G_t = \sum_{k=t}^{\infty} \gamma^{k-t} R_{k+1}$. The state-value function $v_\pi(s)$ is defined as the expected return starting from a state s. The agent aims to find an optimal policy π^* to maximize the expected return.

Proximal policy optimization (PPO) (Schulman et al., 2017) and advantage actor-critic (A2C) (Mnih et al., 2016) are two widely used RL algorithms for continuous control. PPO improves training stability by using a clipped surrogate objective to prevent the policy from changing too much at each time step. A2C is a variant of actor-critic method (Sutton & Barto, 2018), featured with multiple-actor parallel training and synchronized gradient update. In both algorithms, v_π is approximated by v, which is usually parameterized as a neural network. In practice, temporal difference (TD) learning is applied to approximate v_π:

$$v(S_t) \leftarrow v(S_t) + \alpha(R_{t+1} + \gamma v(S_{t+1}) - v(S_t)), \tag{1}$$

where α is the learning rate, S_t and S_{t+1} are two successive states, and $R_{t+1} + \gamma v(S_{t+1})$ is named the TD target. TD targets are usually biased, non-stationary, and noisy due to changing state-values, complex state transitions, and noisy reward signals (Schulman et al., 2016). They usually induce a changing loss landscape that evolves during training. As a result, the agent-gradients [2] usually have high bias and variance (Lan et al., 2022) which can lead to sub-optimal performance or even a failure of convergence.

---

[2]In this work, we use the term agent-gradients to refer to gradients of all parameters in a learning agent, which may include policy gradient, gradient of value functions, gradient of other hyper-parameters.

2.2 Learning to Optimize with Meta-Learning

We aim to learn an optimizer using meta-learning. Let θ be the agent-parameters of an RL agent that we aim to optimize. A (learned) optimizer is defined as an update function U that maps input gradients to parameter updates, implemented as a meta-network, parameterized by the meta-parameters ϕ. Let z be the input of this meta-network which may include gradients g, losses L, exponential moving average of gradients, etc. Let h be an optimizer state which stores historical values. We can then compute agent-parameters updates $\Delta\theta$ and the updated agent-parameters θ':

$$\Delta\theta, h' = U_\phi(z, h) \text{ and } \theta' = \theta + \Delta\theta.$$

Note that all classical first-order optimizers can be written in this form with $\phi = \emptyset$. As an illustration, for SGD, $h' = h = \emptyset$, $z = g$, and $U_{\text{SGD}}(g, \emptyset) = (-\alpha g, \emptyset)$, where α is the learning rate. For RMSProp (Tieleman & Hinton, 2012), set $z = g$; h is used to store the average of squared gradients. Then $U_{\text{RMSProp}}(g, h) = (-\frac{\alpha g}{\sqrt{h'} + \epsilon}, h')$, where $h' = \beta h + (1 - \beta)g^2$, $\beta \in [0, 1]$, and ϵ is a tiny positive number for numerical stability.

Similar to Xu et al. (2020), we apply bilevel optimization to optimize θ and ϕ. First, we collect $M + 1$ trajectories $\mathcal{T} = \{\tau_i, \tau_{i+1}, \cdots, \tau_{i+M-1}, \tau_{i+M}\}$. For the inner update, we fix ϕ and apply multiple steps of gradient descent updates to θ by minimizing an inner loss L^{inner}. Specifically, for each trajectory $\tau_i \in \mathcal{T}$, we have

$$\Delta\theta_i \propto \nabla_\theta L^{\text{inner}}(\tau_i; \theta_i, \phi) \text{ and } \theta_{i+1} = \theta_i + \Delta\theta_i,$$

where $\nabla_\theta L^{\text{inner}}$ are agent-gradients of θ. By repeating the above process for M times, we get $\theta_i \xrightarrow{\phi} \theta_{i+1} \cdots \xrightarrow{\phi} \theta_{i+M}$. Here, θ_{i+M} are functions of ϕ. For simplicity, we abuse the notation and still use θ_{i+M}. Next, we use τ_{i+M} as a validation trajectory to optimize ϕ with an outer loss L^{outer}:

$$\Delta\phi \propto \nabla_\phi L^{\text{outer}}(\tau_{i+M}; \theta_{i+M}, \phi) \text{ and } \phi' = \phi + \Delta\phi,$$

where $\nabla_\phi L^{\text{outer}}$ are meta-gradients of ϕ. Since θ_{i+M} are functions of ϕ, we can apply the chain rule to compute meta-gradients $\nabla_\phi L^{\text{outer}}$, with the help of automatic differentiation packages.

3 Related Work

Our work is closely related to three areas: optimization in RL, discovering general RL algorithms, and learning to optimize in SL.

3.1 Optimization in Reinforcement Learning

Henderson et al. (2018) tested different optimizers in RL and pointed out that classical adaptive optimizers may not always consider the complex interactions between RL algorithms and environments. Sarigül & Avci (2018) benchmarked different momentum strategies in deep RL and found that Nesterov momentum is better at generalization. Bengio et al. (2020a) took one step further and showed that unlike SL, momentum in TD learning becomes doubly stale due to changing parameter updates and bootstrapping. By correcting momentum in TD learning, the sample efficiency can be improved. *These works together indicate that it may not always be appropriate to bring optimization methods in SL directly to RL without considering the unique properties in RL.* Unlike these works which hand-design new optimizers for RL, we adopt a data-driven approach and apply meta-learning to learn an RL optimizer from data generated in the agent-environment interactions.

3.2 Discovering General Reinforcement Learning Algorithms

The data-driven approach is also explored in discovering general RL algorithms. For example, Houthooft et al. (2018) proposed to meta-learn a loss function that takes the agent's history into account and greatly improves learning efficiency. Similarly, Kirsch et al. (2020) proposed MetaGenRL,

which learns the objective function for deterministic policies using off-policy second-order gradients. Oh et al. (2020) applied meta-learning and discovered an entire update rule for RL by interacting with a set of environments. Instead of discovering the entire update rule, Lu et al. (2022) focused on exploring the mirror learning space with evolution strategies and demonstrated the generalization ability in unseen settings. Bechtle et al. (2021) incorporated additional information at meta-train time into parametric loss functions and applied this method to image classification, behavior cloning, and model-based RL. Kirsch et al. (2022) explored the role of symmetries in discovering new RL algorithms and showed that symmetries improve generalization. Jackson et al. (2023) examined the impact of environment design in meta-learning update rules in RL and developed an automatic adversarial environment design approach to improve in-distribution robustness and generalization performance of learned RL algorithms. Following these works, our work adheres to the data-driven spirit, aiming to learn an optimizer instead of general RL algorithms. Our training and evaluation procedures are largely inspired by them as well.

3.3 Learning to Optimize in Supervised Learning

Initially, learning to optimize is only applied to tune the learning rate (Jacobs, 1988; Sutton, 1992; Mahmood et al., 2012). Recently, researchers started to learn an optimizer completely from scratch. Andrychowicz et al. (2016) implemented learned optimizers with long short-term memory networks (Hochreiter & Schmidhuber, 1997) and showed that learned optimizers could generalize to unseen tasks. Li & Malik (2017) applied a guided policy search method to find a good optimizer. Wichrowska et al. (2017) introduced a hierarchical recurrent neural network (RNN) (Medsker & Jain, 2001) architecture, which greatly reduces memory and computation, and was shown to generalize to different network structures. Metz et al. (2022a) developed learned optimizers with multi-layer perceptions, which achieve a better balance among memory, computation, and performance.

Learned optimizers are known to be hard to train. Part of the reason is that they are usually trained by truncated backpropagation through time, which leads to strongly biased gradients or exploding gradients. To overcome these issues, Metz et al. (2019) presented a method to dynamically weigh a reparameterization gradient estimator and an evolutionary strategy style gradient estimator, stabilizing the training of learned optimizers. Vicol et al. (2021) resolved the issues by dividing the computation graph into truncated unrolls and computing unbiased gradients with evolution strategies and gradient bias corrections. Harrison et al. (2022) investigated the training stability of optimization algorithms and proposed to improve the stability of learned optimizers by adding adaptive nominal terms from Adam (Kingma & Ba, 2015) and AggMo (Lucas et al., 2019). Metz et al. (2020a) trained a general-purpose optimizer by training optimizers on thousands of tasks with a large amount of computation. Following the same spirit, Metz et al. (2022b) continued to perform large-scale optimizer training, leveraging more computation ($4,000$ TPU-months) and more diverse SL tasks. The learned optimizer, VeLO, requires no hyperparameter tuning and works well on a wide range of SL tasks. VeLO is the precious outcome of long-time research in the area of learning to optimize, building on the wisdom and effort of many generations. Although marking a milestone for the success of learned optimizers in SL tasks, VeLO still performs poorly in RL tasks, as shown in Section 4.4.4 in Metz et al. (2022b).

The failure of VeLO in RL tasks suggests that designing learned optimizers for RL is still a challenging problem. Unlike previous works that focus on learning optimizers for SL, we aim to learn to optimize for RL. As we will show next, our method is simple, stable, and effective, without using complex neural network structures or incorporating numerous human-designed features. *As far as we know, our work is the first to demonstrate the success of learned optimizers in deep RL tasks.*

4 Issues in Learning to Optimize for Reinforcement Learning

Learned optimizers for SL are infamously hard to train, suffering from high training instability (Wichrowska et al., 2017; Metz et al., 2019; 2020a). Learning an optimizer for RL is even harder (Metz et al., 2022b). In the following, we identify two issues in learning to optimize for RL.

(a) At the beginning of training   (b) In the middle of training   (c) At the end of training
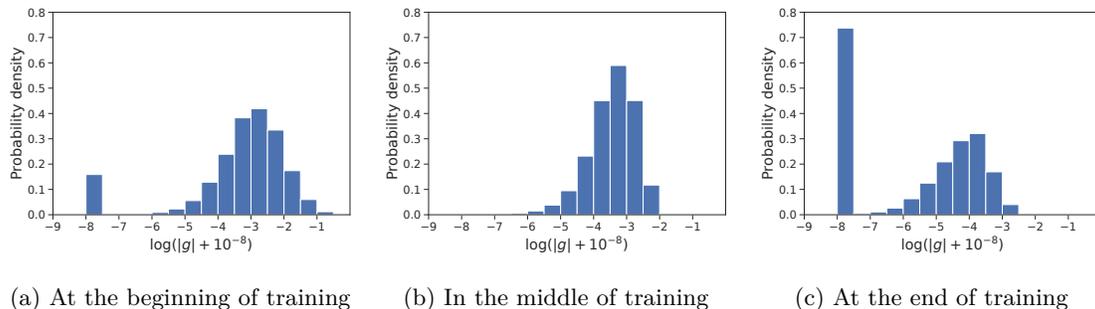
Figure 1: Visualizations of agent-gradient distributions (a) at the beginning of training, (b) in the middle of training, and (c) at the end of training. All agent-gradients are collected during training A2C in big_dense_long, optimized by RMSProp. We compute $\log(|g| + 10^{-8})$ to avoid the error of applying log function to non-positive agent-gradients.

4.1 The Agent-Gradient Distribution is Non-IID

In RL, a learned optimizer takes the agent-gradient g as an input and outputs the agent-parameter update $\Delta\theta$. To investigate the hardness of learning an optimizer for RL, we train an A2C agent in a gridworld (i.e., big_dense_long, see Appendix B for details) with RMSProp (Tieleman & Hinton, 2012) and collect the agent-gradients at different training stages. We plot these agent-gradients with logarithmic x-axis in Figure 1. The y-axis shows the probability density. Clearly, the agent-gradient distribution is non-iid, changing throughout the training process. Specifically, at the beginning of training, there are two peaks in the agent-gradient distribution. In the middle of training, most agent-gradients are non-zero, concentrated around 10^{-3}. At the end of the training, a large portion of the agent-gradients are zeros. It is well-known that a non-iid input distribution makes training more unstable and reduces learning performance in many settings (Ma et al., 2022; Wang et al., 2023; Khetarpal et al., 2022). Similarly, the violation of the iid assumption would also increase learning instability and decrease efficiency for training learned optimizers. Note that this issue exists in both learning to optimize for SL and RL. However, the agent-gradient distribution from RL is generally more non-iid than the gradient distribution from SL, since RL tasks are inherently more non-stationary. For more details, please check Appendix D.

4.2 A Vicious Spiral of Bilevel Optimization

Learning an optimizer while optimizing parameters of a model is a bilevel optimization, suffering from high training instability (Wichrowska et al., 2017; Metz et al., 2020a; Harrison et al., 2022). In RL, due to highly stochastic agent-environment interactions, the agent-gradients have high bias and variance, which make the bilevel optimization even more unstable.

Specifically, in SL, it is often assumed that the training set consists of iid samples. However, the input data distribution in RL is non-iid, which makes the whole training process much more unstable and complex, especially when learning to optimize is involved. In most SL settings, true labels are noiseless and time-invariant. For example, the true label of a written digit 2 in MNIST (Deng, 2012) is $y = 2$, which does not change during training. In RL, TD learning (see Equation (1)) is widely used, and TD targets play a similar role as labels in SL. Unlike labels in SL, TD targets are biased, non-stationary, and noisy, due to highly stochastic agent-environment interactions. This leads to a loss landscape that evolves during training and potentially results in the deadly triad (Van Hasselt et al., 2018) and capacity loss (Lyle et al., 2021). Moreover, in SL, a lower loss usually indicates better performance (e.g., higher classification accuracy). But in RL, a lower outer loss is not necessarily a good indicator of better performance (i.e., higher return) due to a changing loss landscape. Together with biased TD targets, the randomness from state transitions, reward signals, and agent-environment interactions, make the bias and variance of agent-gradients relatively high. In learning

to optimize for RL, meta-gradients are afflicted with large noise induced by the high bias and variance of agent-gradients. With noisy and inaccurate meta-gradients, the improvement of the learned optimizer is unstable and slow. Using a poorly performed optimizer, policy improvement is no longer guaranteed. A poorly performed agent is unlikely to collect "high-quality" data to boost the performance of the agent and the learned optimizer. In the end, this bilevel optimization gets stuck in a vicious spiral: a poor optimizer → a poor agent policy → collected data of low-quality → a poor optimizer → ⋯⋯.

5 Optim4RL: A Learned Optimizer for Reinforcement Learning

To overcome the issues in Section 4, we propose a learned optimizer for RL, named *Optim4RL*, which incorporates pipeline training and a novel optimizer structure. As we will show next, Optim4RL is more robust and efficient to train than previous methods.

5.1 Pipeline Training

In Figure 1, we show that the agent-gradient distribution is non-iid during training. Generally, a good optimizer should be well-functioned under different agent-gradient distributions in the whole training process. To make the agent-gradient distribution more iid, we propose *pipeline training*.

Instead of training only one agent, we train n agents in parallel, each with its own task and optimizer state. Together, the three elements form a *training unit* (agent, task, optimizer state); and we have n training units in total. Let m be a positive integer we call the *reset interval*. A complete *training interval* lasts for m training iterations. In Figure 2 (a), we show an example of pipeline training with $m = n = 3$. To train an optimizer effectively, the input of the learned optimizer includes agent-gradients from all n training units. Before training, we choose n integers $\{r_1, \cdots, r_n\}$ such that they are evenly spaced over the interval $[0, m-1]$. Then we assign r_i to training unit i for $i \in \{1, \cdots, n\}$. At training iteration t, we reset training unit i if $r_i \equiv t \pmod{m}$. By resetting training units at regular intervals, it is guaranteed that at iteration t, we can access training data across one training interval. For instance, at $t = 3$, the input consists of agent-gradients from unit 1 at the beginning of an interval, agent-gradients from unit 2 at the end of an interval, and agent-gradients from unit 3 in the middle of an interval, indicated by the dashed line in Figure 2 (a). With pipeline training, the input agent-gradients are more diverse and spread across a whole training interval, making the input distribution more iid. Ideally, we expect $m \leq n$ so that the input consists of agent-gradients from all training stages. In our experiments, n is the number of training environments; m depends on the training steps of each task, and it has a similar magnitude as n.

5.2 Improving the Inductive Bias of Learned Optimizers

Recently, Harrison et al. (2022) proved that adding adaptive terms to learned optimizers improves the training stability of optimizing a noisy quadratic model. Experimentally, Harrison et al. (2022) showed that adding terms from Adam (Kingma & Ba, 2015) and AggMo (Lucas et al., 2019) improves the stability of learned optimizers as well. However, including human-designed features not only makes an optimizer more complex but is also against the spirit of learning to optimize — ideal learned optimizers should be able to automatically learn useful features, reducing the reliance on human expert knowledge as much as possible. Instead of incorporating terms from adaptive optimizers directly, we design the parameter update function in a similar form to adaptive optimizers:

$$\Delta\theta = -\alpha \frac{m}{\sqrt{v + \epsilon}}, \tag{2}$$

where α is the learning rate, ϵ is a small positive number, and m and v are the processed outputs of dual-RNNs, as shown in Figure 2 (b). Specifically, for each input gradient g, we generate two scalars o_1 and o_2. We then set $m = g_{sign} \exp(o_1)$ and $v = \exp(o_2)$, where $g_{sign} \in \{-1, 1\}$ is the sign of g. More details are included in Algorithm 1.

(a) Pipeline training   (b) The optimizer structure of Optim4RL
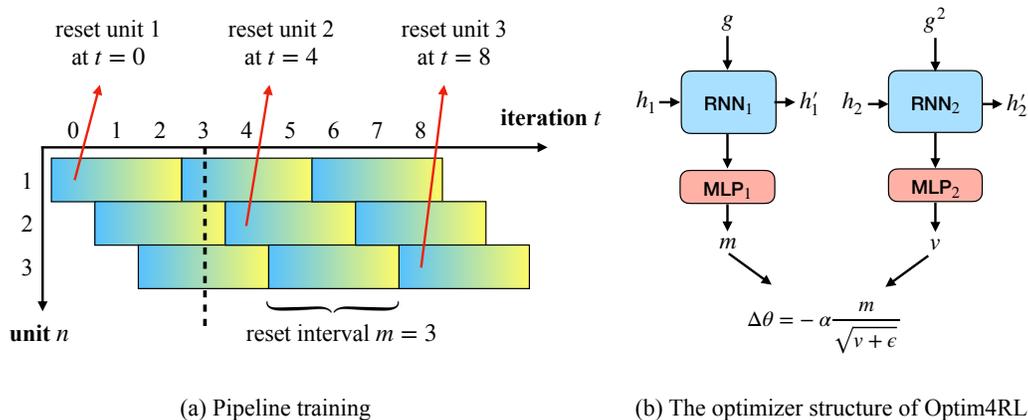
Figure 2: (a) An example of pipeline training where the reset interval $m = 3$ and the number of units $n = 3$. All training units are reset at regular intervals to diversify training data. (b) The network structure of Optim4RL. g is the input agent-gradient, h_i and h'_i are hidden states, α is the learning rate, ϵ is a small positive constant, and $\Delta\theta$ is the parameter update.

By parameterizing the parameter update function as Equation (2), *we improve the inductive bias of learned optimizers by choosing a suitable hypothesis space for learned optimizers and reducing the burden of approximating square root and division for neural networks.* In general, we want to learn a good optimizer in a reasonable hypothesis space. It should be large enough to include as many good optimizers as possible, such as Adam (Kingma & Ba, 2015) and RMSProp (Tieleman & Hinton, 2012). Meanwhile, it should also rule out bad choices so that a suitable candidate can be found efficiently. An optimizer in the form of Equation (2) meets the two requirements exactly. Moreover, it is generally hard for neural networks to approximate mathematical operations accurately (Telgarsky, 2017; Yarotsky, 2017; Boullé et al., 2020; Lu et al., 2021). With Equation (2), a neural network can spend all its expressivity and capacity learning m and v, reducing the burden of approximating square root and division.

Finally, we combine the two techniques and propose our method — a learned optimizer for RL (Optim4RL). Following Andrychowicz et al. (2016), our optimizer also operates coordinatewisely on agent-parameters so that all agent-parameters share the same optimizer. Besides gradients, many previously learned optimizers for SL include human-designed features as inputs, such as moving average of gradient values at multiple timescales, moving average of squared gradients, and Adafactor-style accumulators (Shazeer & Stern, 2018). In theory, these features can be learned and stored in the hidden states of RNNs in Optim4RL. So for simplicity, we only consider agent-gradients as inputs. As we will show next, despite its simplicity, our learned optimizer Optim4RL achieves satisfactory performance in many RL tasks, outperforming several learned optimizers.

6 Experiment

In this section, we first verify that Optim4RL can learn to optimize for RL from scratch. Then, we show how to train a general-purpose learned optimizer for RL. More experimental results are included in Appendix E.

Following Oh et al. (2020), we design several gridworlds with various properties, such as different horizons, reward functions, or state-action spaces. More details are described in Appendix B. Besides gridworlds, we also test our method in Catch (Osband et al., 2020) and Brax tasks (Freeman et al., 2021). We mainly consider two RL algorithms — A2C (Mnih et al., 2016) and PPO (Schulman et al., 2017). For all experiments, we train A2C in gridworlds and train PPO in Brax tasks. For Optim4RL, due to resource constraints, we choose a small network with two GRUs (Cho et al., 2014)
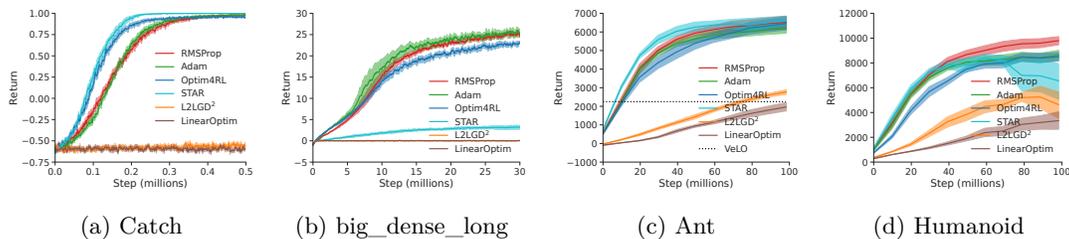
Figure 3: The optimization performance of different optimizers in four RL tasks. Note that the performance of VeLO is estimated based on Figure 11 (a) in Metz et al. (2022b). All other results are averaged over 10 runs, and the shaded areas represent 90% confidence intervals. Optim4RL is the only learned optimizer that achieves satisfactory performance in all tasks.

of hidden size 8; both multi-layer perceptrons (MLPs) have two hidden layers of size 16. We use Adam to optimize learned optimizers. More implementation details are included in Appendix C.

6.1 Learning an Optimizer for RL from Scratch

We first show that it is feasible to train Optim4RL in RL tasks from scratch, while learned optimizers for SL do not work well consistently in RL tasks. We consider both classical (Adam and RMSProp) and learned optimizers (L2LGD^2 (Andrychowicz et al., 2016), STAR (Harrison et al., 2022), and VeLO (Metz et al., 2022b)) as baselines. Except for VeLO, we meta-learn optimizers in one task and then test the *fixed* learned optimizers in this specific task. The optimization performance of optimizers is measured by returns averaging over 10 runs, as shown in Figure 3. In general, L2LGD^2 fails in all four tasks. In Catch, both STAR and Optim4RL perform better than classical optimizers (Adam and RMSProp), achieving a faster convergence rate. In Ant, Optim4RL and STAR perform pretty well, on par with Adam and RMSProp, while significantly outperforming the state-of-the-art optimizer — VeLO. However, STAR fails to optimize effectively in big_dense_long; in Humanoid, STAR's performance is unstable and crashes in the end. *Optim4RL is the only learned optimizer that achieves stable and satisfactory performance in all tasks, which is a significant accomplishment in its own right, as it demonstrates the efficacy of our approach and its potential for practical applications.*

The advantage of the inductive bias of Optim4RL As an ablation study, we demonstrate the advantage of the inductive bias of OptimRL by comparing it with *LinearOptim*, which has a "linear" parameter update function: $\Delta\theta = -\alpha(a * g + b)$, where α is the learning rate, a and b are the outputs of an RNN model. The only difference between LinearOptim and Optim4RL is the inductive bias — the parameter update function of LinearOptim is in the form of a linear function. In contrast, the parameter update function of Optim4RL is inspired by adaptive optimizers (see Equation (2)). As shown in Figure 3, LinearOptim fails to optimize in all tasks, verifying the advantage of the inductive bias of Optim4RL.

The effectiveness of pipeline training By making the input agent-gradient distribution more iid and less time-dependent, pipeline training could improve the training stability and efficiency. To verify this claim, we compare the optimization performance of Optim4RL with and without pipeline training in Table 1. We observe minor performance improvement in two gridworlds (small_dense_long and big_dense_long) and more significant improvement in two Brax tasks (Ant and Humanoid), confirming the effectiveness of pipeline training.

6.2 Toward a General-Purpose Learned Optimizer for RL

A general-purpose optimizer should perform well even when the input gradients are at various scales. To meta-train a learned general-purpose optimizer, first we design six gridworlds such that the generated agent-gradients in these tasks vary across a wide range. To demonstrate the generalization

(a) Ant (b) Humanoid (c) Pendulum (d) Walker2D
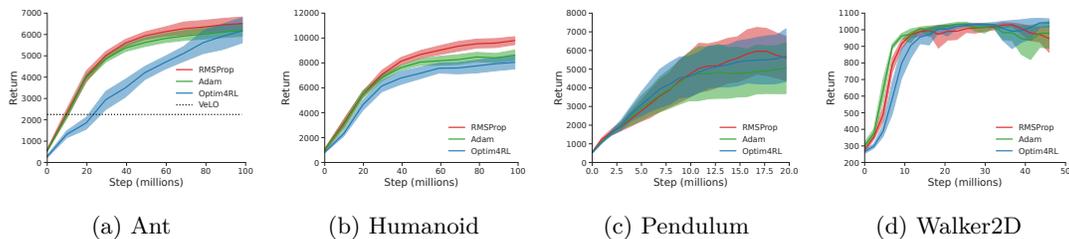
Figure 4: Optim4RL shows strong generalization ability and achieves good performance in Brax tasks, although it is only trained in six simple gridworlds from scratch. For comparison, VeLO (Metz et al., 2022b) is trained for 4,000 TPU-months with thousands of tasks but only achieves sub-optimal performance in Ant. These results demonstrate the generalization ability of Optim4RL in complex unseen tasks, which is a significant achievement in itself, proving the effectiveness of our approach.

Method \ Task	small_dense_long	big_dense_long	Ant	Humanoid
With Pipeline Training	32.22±0.52	23.10±0.31	6421±355	8440±364
W.o. Pipeline Training	30.64±0.69	22.47±0.51	5038±235	6557±1055

Table 1: The performance of Optim4RL with and without pipeline training. All results are averaged over 10 runs, reported with 90% confidence intervals.

ability of Optim4RL, we then meta-train Optim4RL in these gridworlds with A2C and test it in Brax tasks with PPO. As shown in Figure 4, Optim4RL achieves satisfactory performance in these tasks, showing a strong generalization ability. Note that Optim4RL surpasses VeLO (the state-of-the-art learned optimizer) significantly in Ant. This is a great success since VeLO is trained for 4,000 TPU-months on thousands of tasks while Optim4RL is only trained in six toy tasks for a few GPU-hours. Finally, Optim4RL is also competitive compared with classical human-designed optimizers (Adam and RMSProp), even though it is entirely trained from scratch. *Training a universally applicable learned optimizer for RL tasks is an inherently formidable challenge. Our results demonstrate the generalization ability of Optim4RL in complex unseen tasks, which is a great achievement in itself, proving the effectiveness of our approach.*

7 Conclusion and Future Work

In this work, we analyzed the hardness of learning to optimize for RL and studied the failures of learned optimizers in RL. Our investigation reveals that agent-gradients in RL are non-iid and have high bias and variance. To mitigate these problems, we introduced pipeline training and a novel optimizer structure. Combining these techniques, we proposed a learned optimizer for RL, Optim4RL, which can be meta-learned to optimize RL tasks entirely from scratch. Although only trained in toy tasks, Optim4RL showed its strong generalization ability to unseen complex tasks.

Learning to optimize for RL is a challenging problem. Due to memory and computation constraints, our current result is limited since we can only train Optim4RL in a small number of toy tasks. In the future, by leveraging more computation and memory, we expect to extend our approach to a larger scale and improve the performance of Optim4RL by training in more tasks with diverse RL agents. Moreover, theoretically analyzing the convergence of learned optimizers is also an interesting topic. We hope our analysis and proposed method can inspire and benefit future research, paving the way for better learned optimizers for RL.

References

Bastian Alt, Adrian Šošić, and Heinz Koeppl. Correlation priors for reinforcement learning. *Advances in Neural Information Processing Systems*, 32, 2019.

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*, 2016.

Sarah Bechtle, Artem Molchanov, Yevgen Chebotar, Edward Grefenstette, Ludovic Righetti, Gaurav Sukhatme, and Franziska Meier. Meta learning via learned loss. In *2020 25th International Conference on Pattern Recognition (ICPR)*, 2021.

Emmanuel Bengio, Joelle Pineau, and Doina Precup. Correcting momentum in temporal difference learning. *NeurIPS Workshop on Deep RL*, 2020a.

Emmanuel Bengio, Joelle Pineau, and Doina Precup. Interference and generalization in temporal difference learning. In *International Conference on Machine Learning*, 2020b.

Nicolas Boullé, Yuji Nakatsukasa, and Alex Townsend. Rational neural networks. *Advances in Neural Information Processing Systems*, 33:14243–14253, 2020.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/google/jax.

Yutian Chen, Matthew W Hoffman, Sergio Gomez Colmenarejo, Misha Denil, Timothy P Lillicrap, Matt Botvinick, and Nando de Freitas. Learning to learn without gradient descent by gradient descent. *International Conference on Machine Learning*, 2017.

Kyunghyun Cho, Bart van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder–decoder approaches. In *Proceedings of SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation*, 2014.

Li Deng. The MNIST database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 2012.

C. Daniel Freeman, Erik Frey, Anton Raichuk, Sertan Girgin, Igor Mordatch, and Olivier Bachem. Brax - a differentiable physics engine for large scale rigid body simulation, 2021. URL http://github.com/google/brax.

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

Peter Henderson, Joshua Romoff, and Joelle Pineau. Where did my optimum go?: An empirical analysis of gradient descent optimization in policy gradient methods. In *The 14th European Workshop on Reinforcement Learning*, 2018.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 1997.

Rein Houthooft, Yuhua Chen, Phillip Isola, Bradly Stadie, Filip Wolski, OpenAI Jonathan Ho, and Pieter Abbeel. Evolved policy gradients. *Advances in Neural Information Processing Systems*, 2018.

Matthew Thomas Jackson, Minqi Jiang, Jack Parker-Holder, Risto Vuorio, Chris Lu, Gregory Farquhar, Shimon Whiteson, and Jakob Nicolaus Foerster. Discovering general reinforcement learning algorithms with adversarial environment design. *Advances in Neural Information Processing Systems*, 2023.

Robert A Jacobs. Increased rates of convergence through learning rate adaptation. *Neural Networks*, 1988.

Khimya Khetarpal, Matthew Riemer, Irina Rish, and Doina Precup. Towards continual reinforcement learning: A review and perspectives. *Journal of Artificial Intelligence Research*, 2022.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations*, 2015.

Louis Kirsch, Sjoerd van Steenkiste, and Juergen Schmidhuber. Improving generalization in meta reinforcement learning using learned objectives. In *International Conference on Learning Representations*, 2020.

Louis Kirsch, Sebastian Flennerhag, Hado van Hasselt, Abram Friesen, Junhyuk Oh, and Yutian Chen. Introducing symmetries to black box meta reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2022.

Qingfeng Lan, Samuele Tosatto, Homayoon Farrahi, and Rupam Mahmood. Model-free policy learning with reward gradients. In *International Conference on Artificial Intelligence and Statistics*, 2022.

Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 2015.

Ke Li and Jitendra Malik. Learning to optimize. In *International Conference on Learning Representations*, 2017.

Chris Lu, Jakub Kuba, Alistair Letcher, Luke Metz, Christian Schroeder de Witt, and Jakob Foerster. Discovered policy optimisation. *Advances in Neural Information Processing Systems*, 2022.

Lu Lu, Pengzhan Jin, Guofei Pang, Zhongqiang Zhang, and George Em Karniadakis. Learning nonlinear operators via DeepONet based on the universal approximation theorem of operators. *Nature Machine Intelligence*, 2021.

James Lucas, Shengyang Sun, Richard Zemel, and Roger Grosse. Aggregated momentum: Stability through passive damping. In *International Conference on Learning Representations*, 2019.

Clare Lyle, Mark Rowland, and Will Dabney. Understanding and preventing capacity loss in reinforcement learning. In *International Conference on Learning Representations*, 2021.

Xiaodong Ma, Jia Zhu, Zhihao Lin, Shanxuan Chen, and Yangjie Qin. A state-of-the-art survey on solving non-iid data in federated learning. *Future Generation Computer Systems*, 2022.

Niru Maheswaranathan, David Sussillo, Luke Metz, Ruoxi Sun, and Jascha Sohl-Dickstein. Reverse engineering learned optimizers reveals known and novel mechanisms. *Advances in Neural Information Processing Systems*, 2021.

Ashique Rupam Mahmood, Richard S Sutton, Thomas Degris, and Patrick M Pilarski. Tuning-free step-size adaptation. In *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2012.

Larry R Medsker and LC Jain. Recurrent neural networks. *Design and Applications*, 2001.

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*, 2019.

Luke Metz, Niru Maheswaranathan, C Daniel Freeman, Ben Poole, and Jascha Sohl-Dickstein. Tasks, stability, architecture, and compute: Training more effective learned optimizers, and using them to train themselves. *arXiv preprint arXiv:2009.11243*, 2020a.

Luke Metz, Niru Maheswaranathan, Ruoxi Sun, C Daniel Freeman, Ben Poole, and Jascha Sohl-Dickstein. Using a thousand optimization tasks to learn hyperparameter search strategies. *arXiv preprint arXiv:2002.11887*, 2020b.

Luke Metz, C Daniel Freeman, James Harrison, Niru Maheswaranathan, and Jascha Sohl-Dickstein. Practical tradeoffs between memory, compute, and performance in learned optimizers. In *Conference on Lifelong Learning Agents*, 2022a.

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.

Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, 2016.

Junhyuk Oh, Matteo Hessel, Wojciech M Czarnecki, Zhongwen Xu, Hado P van Hasselt, Satinder Singh, and David Silver. Discovering reinforcement learning algorithms. *Advances in Neural Information Processing Systems*, 2020.

Ian Osband, Yotam Doron, Matteo Hessel, John Aslanides, Eren Sezener, Andre Saraiva, Katrina McKinney, Tor Lattimore, Csaba Szepesvari, Satinder Singh, et al. Behaviour suite for reinforcement learning. In *International Conference on Learning Representations*, 2020.

Mehmet Sarigül and Mutlu Avci. Performance comparison of different momentum techniques on deep reinforcement learning. *Journal of Information and Telecommunication*, 2018.

John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. In *International Conference on Learning Representations*, 2016.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Noam Shazeer and Mitchell Stern. Adafactor: Adaptive learning rates with sublinear memory cost. In *International Conference on Machine Learning*, 2018.

Richard S Sutton. Adapting bias by gradient descent: An incremental version of delta-bar-delta. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 1992.

Richard S Sutton and Andrew G Barto. *Reinforcement Learning: An Introduction*. MIT Press, second edition, 2018.

Matus Telgarsky. Neural networks and rational functions. In *International Conference on Machine Learning*, 2017.

Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-RMSProp: Divide the gradient by a running average of its recent magnitude. *COURSERA Neural Networks Neural Networks for Machine Learning*, 2012.

Hado Van Hasselt, Yotam Doron, Florian Strub, Matteo Hessel, Nicolas Sonnerat, and Joseph Modayil. Deep reinforcement learning and the deadly triad. *arXiv preprint arXiv:1812.02648*, 2018.

Paul Vicol, Luke Metz, and Jascha Sohl-Dickstein. Unbiased gradient estimation in unrolled computation graphs with persistent evolution strategies. In *International Conference on Machine Learning*, 2021.

Liyuan Wang, Xingxing Zhang, Hang Su, and Jun Zhu. A comprehensive survey of continual learning: Theory, method and application. *arXiv preprint arXiv:2302.00487*, 2023.

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*, 2017.

Zhongwen Xu, Hado P van Hasselt, Matteo Hessel, Junhyuk Oh, Satinder Singh, and David Silver. Meta-gradient reinforcement learning with an objective discovered online. *Advances in Neural Information Processing Systems*, 2020.

Dmitry Yarotsky. Error bounds for approximations with deep ReLU networks. *Neural Networks*, 2017.

A   Pseudocode of Optim4RL

The pseudocode of Optim4RL is presented in Algorithm 1. Specifically, in all our experiments, we use GRUs (Cho et al., 2014) with hidden size 8, MLPs with hidden sizes $[16, 16]$.

Algorithm 1 Optim4RL: A Learned Optimizer for Reinforcement Learning

---

Require: RNN_1 and RNN_2, MLP_1 and MLP_2, hidden states h_1 and h_2, input gradient g, $\epsilon = 10^{-8}$, learning rate α.

$g \leftarrow \perp g$ $\triangleright \perp$ denotes the stop-gradient operation

$h_1, x_1 \leftarrow \text{RNN}_1(h_1, g)$ and $o_1 = \text{MLP}_1(x_1)$

$m = \text{sign}(g) \exp(o_1)$ $\triangleright$ Compute m: 1st pseudo moment estimate

$h_2, x_2 \leftarrow \text{RNN}_2(h_2, g^2)$ and $o_2 = \text{MLP}_2(x_2)$

$v = \exp(o_2)$ $\triangleright$ Compute v: 2nd pseudo moment estimate

$\Delta\theta \leftarrow -\alpha \frac{m}{\sqrt{v+\epsilon}}$ $\triangleright$ Compute the parameter update

---

B   Gridworlds

We follow Oh et al. (2020) and design 6 gridwolds. In each gridworld, there are N objects. Each object is described as $[r, \epsilon_{\text{term}}, \epsilon_{\text{respawn}}]$. Object locations are randomly determined at the beginning of each episode, and an object reappears at a random location after being collected, with a probability of $\epsilon_{\text{respawn}}$ for each time-step. The observation consists of a tensor $\{0, 1\}^{N \times H \times W}$, where N is the number of objects, and $H \times W$ is the size of the grid. An agent has 9 movement actions for adjacent positions, including staying in the same position. When the agent collects an object, it receives the corresponding reward ($r \times$ reward scale), and the episode terminates with a probability of ϵ_{term} associated with the object. The default reward scale is 1. In Table 2, we describe the setting of each gridworld in detail.

Task \ Setting	Size ($H \times W$)	Objects	Horizon
big_sparse_short	10×12	$2 \times [1.0, 0.0, 0.05]$, $2 \times [-1.0, 0.5, 0.05]$	50
big_sparse_long	12×10	$2 \times [1.0, 0.0, 0.05]$, $2 \times [-1.0, 0.5, 0.05]$	500
big_dense_short	9×13	$2 \times [1.0, 0.0, 0.5]$, $2 \times [-1.0, 0.5, 0.5]$	50
big_dense_long	13×9	$2 \times [1.0, 0.0, 0.5]$, $2 \times [-1.0, 0.5, 0.5]$	500
small_dense_long	6×4	$[1.0, 0.0, 0.5]$, $[-1.0, 0.5, 0.5]$	500
small_dense_short	4×6	$[1.0, 0.0, 0.5]$, $[-1.0, 0.5, 0.5]$	50

Table 2: The detailed settings of gridworlds.

C   Experimental Details

In this work, we apply Jax (Bradbury et al., 2018) to do automatic differentiation. For A2C training in gridworlds, the feature net is an MLP with hidden size 32 for the "small" gridworlds. For the "big" gridworlds, the feature net is a convolution neural network (CNN) with 16 features and kernel size 2, followed by an MLP with output size 32. Unless mentioned explicitly, we use ReLU as the activation function. We set $\lambda = 0.95$ to compute λ-returns. The discount factor $\gamma = 0.995$. One rollout has 20 steps. The critic loss weight is 0.5, and the entropy weight is 0.01. For PPO training in Brax games, we use the same settings in Brax examples [3]. To meta-learn optimizers, we set $M = 4$ in all experiments; that is, for every outer update, we do 4 inner updates. Potentially, a larger M could lead to more farsighted learning but results in increasing memory and computation

---

[3] https://github.com/google/brax/blob/main/notebooks/training.ipynb

requirements. We set $M = 4$ as a trade-off, which also works well in practice. Following common practice (Lu et al., 2022), we report results averaged over 10 runs. Other details are presented in the following sections.

C.1 Computation Resource

All our experiments can be trained with V100 GPUs. For some experiments, we use 4 V100 GPUs due to a large GPU memory requirement. The computation to repeat all experimental results in this work should be less than 1 GPU-year, while the exact computation used is hard to estimate.

C.2 Implementation Details for Section 4

We collect agent-gradients by training A2C in big_dense_long for $30M$ steps with learning rate $3e - 3$, optimized by RMSProp. All collected agent-gradients are divided into 30 parts by time-steps. We then plot the agent-gradients in the first, sixteenth, and last parts as the agent-gradient distributions at the beginning, middle, and end of training, respectively.

C.3 Implementation Details for Section 6.1

For both LinearOptim and L2LGD^2, the model consists of a GRU with hidden size 8, followed by an MLP with hidden sizes $[16, 16]$. For STAR, we use the official implementation from learned_optimization [4]. Unlike the supervised learning setting, we set weight decay to 0 since a positive weight decay in STAR leads to much worse performance. For a fair comparison, we apply pipeline training to train all learned optimizers.

For Catch, the agent learning rate is $1e - 3$; the number of environments / training units n is 64; the reset interval m is chosen from $\{32, 64\}$. For big_dense_long, the agent learning rate is $3e - 3$; the number of environments / training units n is 512; the reset interval m is chosen from $\{72, 144, 288, 576\}$. For Ant and Humanoid, the agent learning rate is $3e - 4$; the number of environments / training units n is 2048; the reset interval m is chosen from $\{32, 64, 128, 256, 512\}$. Furthermore, in order to reduce memory requirement, we set the number of mini-batches to 8; and change the hidden sizes of the value network from $[256, 256, 256, 256, 256]$ to $[64, 64, 64, 64, 64]$.

We use Adam as the meta optimizer and choose the meta learning rate from $\{1e - 5, 3e - 5, 1e - 4, 3e - 4, 1e - 3, 3e - 3, 1e - 2\}$.

C.4 Implementation Details for Section 6.2

Optim4RL is meta-trained in 6 gridworlds and then tested in Brax tasks. We use Adam as the meta optimizer and choose the meta learning rate from $\{1e - 5, 3e - 5, 1e - 4, 3e - 4, 1e - 3, 3e - 3\}$. The number of environments/training units n is 512. The reset interval m is chosen from $\{72, 144, 288, 576\}$. The reward scales of all gridworlds are in Table 3.

Gridworld	Reward Scale
small_dense_long	1000
small_dense_short	100
big_sparse_short	100
big_dense_short	10
big_sparse_long	10
big_dense_long	1

Table 3: The reward scales of gridworlds used for learning a general-purpose optimizer.

---

[4]https://github.com/google/learned_optimization/blob/main/learned_optimization/learned_optimizers/adafac_nominal.py

D Experiments for the Gradient Distribution in Supervised Learning

In this section, we show that the gradient distribution in supervised learning is also non-iid, but it is more iid than the agent-gradient distribution in RL. Specifically, we train a neural network on MNIST (Deng, 2012) for 10 epochs with RMSProp and collect gradients at different training stages. Note that the network is the same as the actor network used in training A2C in big_dense_long, except for the output layer. We plot these gradients with logarithmic x-axis in Figure 5. Similar to Figure 1, the gradient distribution is also non-iid, changing throughout the training process.

(a) At the beginning of training (b) In the middle of training (c) At the end of training
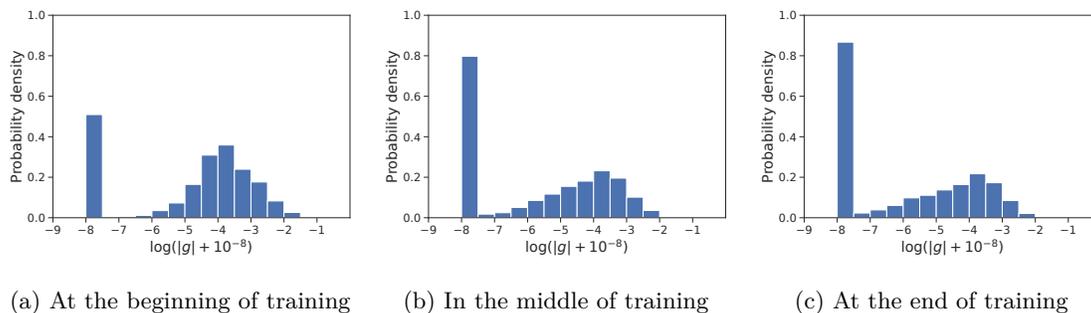
Figure 5: Visualizations of gradient distributions (a) at the beginning of training, (b) in the middle of training, and (3) at the end of training. All gradients are collected during training in MNIST, optimized by RMSProp.

To show the gradient distribution from training on MNIST is more iid than the agent-gradient distribution from training in big_dense_long, we compute the Wasserstein distance (WD) between the (agent-)gradient distribution at different training stages and the distribution of all (agent-)gradients during training in Table 4. Note that a smaller distance indicates a higher iid degree. Thus these results support the above claim.

Task \ WD Value	WD(beginning, all)	WD(middle, all)	WD(end, all)
MNIST	4.0642×10^{-4}	0.7205×10^{-4}	1.4204×10^{-4}
big_dense_long	7.1583×10^{-4}	1.1726×10^{-4}	6.7967×10^{-4}

Table 4: The Wasserstein distance between the (agent-)gradient distribution at different training stages and the distribution of all (agent-)gradients during training.

E Robust Training and Strong Generalization Under Different Hyper-Parameter Settings

Generally, we find it hard to train learned optimizers partly due to Not a Number (NaN) errors during training, even when gradient clipping or gradient normalization is applied. For example, among all meta-training hyper-parameter settings, we fail to train STAR due to NaN errors in more than 80% and 50% settings in Humanoid and Ant, respectively. However, NaN errors are seldom encountered when we meta-train Optim4RL, LinearOptim, and L2LGD^2 in Humanoid and Ant; and Optim4RL is the only one that achieves satisfactory performance among them.

Next, we show that Optim4RL not only generalizes to unseen tasks, but also transfers to different hyper-parameter settings. To be specific, we train our learned optimizer Optim4RL under the default hyper-parameter setting and then test it under different hyper-parameter settings in two gridworlds – small_dense_short and big_dense_long. We report the returns at the end of training, averaged

over 10 runs. As shown in Table 5, Table 6, and Table 7, Optim4RL is robust under different hyper-parameter settings, such as GAE λ, entropy weight, and discount factor.

Task	Parameter Value	Return
small_dense_short	0.9	11.51±0.19
small_dense_short	0.95	11.25±0.16
small_dense_short	0.99	10.81±0.17
small_dense_short	0.995	10.66±0.17
big_dense_long	0.9	23.35±0.76
big_dense_long	0.95	23.57±0.60
big_dense_long	0.99	21.31±0.64
big_dense_long	0.995	20.55±0.54

Table 5: The performance of Optim4RL with different *GAE λ values* in two gridworlds. All results are averaged over 10 runs, reported with 90% confidence intervals.

Task	Parameter Value	Return
small_dense_short	0.005	11.01±0.16
small_dense_short	0.01	11.13±0.09
small_dense_short	0.02	11.29±0.12
small_dense_short	0.04	11.25±0.16
big_dense_long	0.005	22.41±0.59
big_dense_long	0.01	22.45±0.79
big_dense_long	0.02	22.59±0.43
big_dense_long	0.04	19.96±1.30

Table 6: The performance of Optim4RL with different *entropy weights* in two gridworlds. All results are averaged over 10 runs, reported with 90% confidence intervals.

Task	Parameter Value	Return
small_dense_short	0.9	12.47±0.14
small_dense_short	0.95	12.32±0.09
small_dense_short	0.99	11.48±0.09
small_dense_short	0.995	11.01±0.17
big_dense_long	0.9	18.13±3.27
big_dense_long	0.95	25.01±1.42
big_dense_long	0.99	25.45±0.47
big_dense_long	0.995	22.07±0.81

Table 7: The performance of Optim4RL with different *discount factors* in two gridworlds. All results are averaged over 10 runs, reported with 90% confidence intervals.