How Should We Meta-Learn Reinforcement Learning Algorithms?

Anonymous authors
Paper under double-blind review

Keywords: Meta-Reinforcement Learning, Algorithm Discovery.

Summary

The process of meta-learning algorithms from data, instead of relying on manual design, is growing in popularity as a paradigm for improving the performance of machine learning systems. Meta-learning shows particular promise for reinforcement learning (RL), where algorithms are often adapted from supervised or unsupervised learning despite their suboptimality for RL. However, until now there has been a severe lack of comparison between different meta-learning algorithms, such as using evolution to optimise over black-box functions or LLMs to propose code. In this paper, we carry out this empirical comparison of the different approaches when applied to a range of meta-learned algorithms, which each target different parts of the RL pipeline. In addition to meta-train and meta-test performance, we also investigate factors including the interpretability, sample cost and train time for each meta-learning algorithm. Based on these findings, we propose several guidelines for meta-learning new RL algorithms which will help ensure that future learned algorithms are as performant as possible.

Contribution(s)

1. We provide a large scale empirical study comparing different meta-learning algorithms when applied to a range of meta-learned algorithms for reinforcement learning. This study considers meta-train and meta-test performance of learned algorithms in addition to how sample-efficient, time-consuming and interpretable different meta-learning algorithms are.
Context: Prior work has introduced a number of different meta-learning algorithms, such as using evolution to optimise black-box algorithms (Goldie et al., 2024; Lu et al., 2022), prompting LLMs to propose new functions (Lu et al., 2024), or using symbolic distillation of black-box functions to discover interpretable symbolic algorithms (Zheng et al., 2022). However, there has not been a direct comparison of these different meta-learning algorithms, limiting our understanding.

2. Based on our experimental results, we produce a set of practical design principles for meta-learning pipelines in the future. These can be used to ensure that meta-learned algorithms are as performant as possible while satisfying the needs of a researcher.
Context: Meta-learning experiments are very time-consuming and costly. For instance, Goldie et al. (2024) uses over 2 GPU-years of compute for meta-learning optimisers in small-scale RL environments, and Metz et al. (2022b) use over 4000 TPU-months to meta-learn a large versatile optimisation algorithm. Separately, in our results, we find that the performance of a meta-learned algorithm is directly linked to how it is learned. Our design principles help to reduce redundant experimentation, while at the same time ensuring that performance of learned algorithms is maximised.

How Should We Meta-Learn Reinforcement Learning Algorithms?

Anonymous authors
Paper under double-blind review

Abstract

The process of meta-learning algorithms from data, instead of relying on manual design, is growing in popularity as a paradigm for improving the performance of machine learning systems. Meta-learning shows particular promise for reinforcement learning (RL), where algorithms are often adapted from supervised or unsupervised learning despite their suboptimality for RL. However, until now there has been a severe lack of comparison between different meta-learning algorithms, such as using evolution to optimise over black-box functions or LLMs to propose code. In this paper, we carry out this empirical comparison of the different approaches when applied to a range of meta-learned algorithms, which each target different parts of the RL pipeline.[1] In addition to meta-train and meta-test performance, we also investigate factors including the interpretability, sample cost and train time for each meta-learning algorithm. Based on these findings, we propose several guidelines for meta-learning new RL algorithms which will help ensure that future learned algorithms are as performant as possible.

1 Introduction

The improvement of machine learning algorithms typically relies on manual design, a cumbersome process that is limited by human intuition and yields breakthroughs only rarely. An alternative, recent paradigm instead involves *meta-learning* learning algorithms from data. In this setting, algorithms are discovered computationally, with only limited need for human intervention in the design of the meta-learning process. This has particular potential for reinforcement learning (Sutton & Barto, 2020, RL), which is prone to instability (Van Hasselt et al., 2018; Achiam et al., 2019; Tang & Berseth, 2024) and often borrows algorithms from supervised and unsupervised learning that require adaptation to RL (e.g., (Parisotto et al., 2020; Obando Ceron et al., 2023; Ellis et al., 2024)).

There are numerous meta-learning algorithms, such as using evolution to optimise over neural networks for black-box algorithms, prompting a language model to propose algorithms in code or distilling from a pretrained black-box algorithm into a symbolic function. However, while many papers compare their meta-learned algorithms with handcrafted baselines, there have been few direct comparisons between methods for learning the algorithm itself. Consequently, there is little clarity on the pros and cons of different meta-learning algorithms, and to which settings they are most suited.

In this paper, we aim to address this deficit with an empirical analysis of different meta-learning algorithms. We consider a number of *meta-learned algorithms* – learned algorithms which replace certain components in RL training – and find the best *meta-learning algorithms* for each – ways for training the learned algorithm. This distinction is visualised in Figure 1, which is based on a figure from Goldie et al. (2024). We select meta-learned algorithms that exhibit different qualities, such as using recurrence or a large number of inputs, to provide coverage for a range of different possible algorithm features. These include learned optimisers and a learned drift function (Kuba et al., 2024).

[1] Code will be open-sourced upon acceptance.

Our analysis focuses on the trade-offs between the different meta-learning algorithms. Principally, we consider the performance of each approach, both within its meta-training domain and in generalisation to new environments. In reinforcement learning, this is particularly important since algorithms often show limited ability to transfer (e.g., (Jackson et al., 2023)). In addition, due to the significant cost incurred by meta-learning experiments, which can require thousands of TPU-months of compute (Metz et al., 2022b), and the need for environment simulation in RL that is not present in supervised and unsupervised learning, we also consider the time and compute cost for training. Finally, we discuss the interpretability of the learned algorithms, which is useful for analysing the behaviour of an algorithm and its corresponding safety implications.

In our results, we find that: prompting a language model is a sample-efficient way to find effective RL algorithms, but only when there is a good algorithm from which to kickstart meta-training; distillation of learned algorithms into other networks sometimes improves performance without increasing the sample cost; and symbolic representations do not scale to recurrent algorithms or those with many inputs. Based on these findings, we propose several guidelines for better ways to meta-learn new RL algorithms, such as suggesting that many cases should use LLM to propose new algorithms and that distillation from a black-box algorithm into another network is usually worth trying for a potential cheap performance boost. We hope that these guidelines can help reduce the cost of research in meta-RL while ensuring that meta-learned algorithms are as capable as possible.

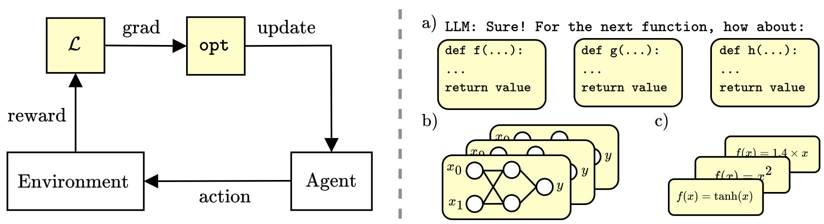


Figure 1: In the RL training loop on the left, we show different components of reinforcement learning which could be replaced by meta-learned algorithms. For example, OPEN (Goldie et al., 2024) is a learned optimiser replacing `opt`, while LPO (Lu et al., 2022) is a learned loss function which replaces $\mathcal{L}$. On the right, we demonstrate a few meta-learning algorithms, such as: a), prompting an LLM to propose new functions; b), evolving a black-box algorithm; or c), using symbolic evolution.

2 Related Work

2.1 Learned Algorithms

The practice of *meta-learning* algorithms is growing in popularity for both RL (Beck et al., 2024) and machine learning in general. There are many learned optimisation algorithms in supervised and unsupervised learning (e.g., (Andrychowicz et al., 2016; Metz et al., 2019b; 2020; Almeida et al., 2021)). Unlike these works, which present new meta-*learned* algorithms, we focus on understanding how the meta-*learning* algorithm affects a number of factors in RL, like generalisation. This is particularly important due to the instability of RL (Van Hasselt et al., 2018; Igl et al., 2021a) and the importance of transfer between environments (Finn et al., 2017; Duan et al., 2016; Jia et al., 2022).

Instead of meta-learning black-box algorithms represented by neural networks, some approaches discover *symbolic* algorithms defined as interpretable mathematical functions. Symbolic algorithms fit naturally into an LLM-based pipeline, since they are easily represented in code. Symbolic programs can be found through symbolic evolution (e.g., Lion (Chen et al., 2023)) or by prompting LLMs to improve algorithms over meta-training (e.g., (Lehman et al., 2022; Lu et al., 2024; Romera-Paredes et al., 2024)). In part of this work, we explore when symbolic algorithms are better than black-box ones, as suggested by Chen et al. (2023).

In RL, a pioneering meta-learned algorithm is Learned Policy Gradient (Oh et al., 2020, LPG), which replaces the actor-critic update, although there are many learned RL algorithms (e.g., (Kirsch et al., 2020; Jackson et al., 2023; Kirsch & Schmidhuber, 2022; Lan et al., 2024)). In addition to LPG, we focus on Learned Policy Optimisation (Lu et al., 2022, LPO), a learned alternative to proximal policy optimisation (Schulman et al., 2017, PPO); and Optimisation for Plasticity, Exploration and Nonstationarity (Goldie et al., 2024, OPEN), a learned optimiser that uses feature engineering for meta-learning. Different to these papers, which propose new meta-learned algorithms for RL, we instead seek to understand how the meta-learning algorithm itself affects performance.

Generalisation after meta-training is important for learned algorithms to be applied in new settings. Jackson et al. (2023) explore using curricula based on unsupervised environment design (Dennis et al., 2021; Parker-Holder et al., 2022) for meta-training as a way to improve LPG generalisation. In this work, we consider how different meta-learning algorithms affect generalisation. As a separate component of the meta-training process, our study is complementary to that of Jackson et al. (2023).

2.2 Distillation

Distillation, which trains a *student* to imitate a *teacher* (Hinton et al., 2015), relates to many meta-learning algorithms. Distillation is often applied to policies (Rusu et al., 2016; Jia et al., 2022), datasets (Wang et al., 2020; Lupu et al., 2024), handcrafted algorithms (Laskin et al., 2023; Son et al., 2025), and reasoning language models (DeepSeek-AI et al., 2025). Distillation from one network to another, called black-box distillation, usually trains a student that is *smaller* than its teacher (Hinton et al., 2015), to reduce inference costs and overfitting, or the *same size* as the teacher (Furlanello et al., 2018), since distillation *itself* acts as a regulariser (Zhang & Sabuncu, 2020; Mobahi et al., 2020). Contrary to these papers, our analysis explores whether applying black-box distillation to learned algorithms provides similar benefits as in other settings.

Rather than distilling from one network to another, symbolic distillation learns a *symbolic program* (Cranmer et al., 2020) that has a similar mapping to the neural network teacher. Symbolic distillation is often applied to physical systems (e.g., (Cranmer et al., 2020; Mengel et al., 2023; Lemos et al., 2023)) for interpretability reasons, but has been extended to learned optimisers (Zheng et al., 2022; Song et al., 2024a). Similarly, Lu et al. (2022) manually distil LPO, a black-box algorithm, into *discovered* policy optimisation. In this paper, we seek to understand *when* symbolic distillation is appropriate for meta-learned RL algorithms. While interpretability is part of our analysis, we also consider whether symbolic distillation improves generalisation of learned algorithms.

3 Background

Reinforcement Learning Reinforcement learning (RL) problems are often modeled as Markov decision processes (Sutton & Barto, 2020, MDPs). An MDP is as a tuple $\langle \mathcal{A}, \mathcal{S}, \mathcal{S}_0, P, \rho, R, \gamma \rangle$. In an MDP, an agent in state $s_t \in \mathcal{S}$, starting at $s_0 \in \mathcal{S}_0$, takes an action $a_t \in \mathcal{A}$ according to its state-conditioned, probabilistic policy $\pi(\cdot|s_t)$ and the state transitions to s_{t+1} based on the transition dynamics $P(\cdot|s_t, a_t)$. In response, the environment generates a reward $r = R(s_t, a_t)$. An agent's policy is trained to maximise its expected discounted return, $J^\pi = \mathbb{E}_{a_{0:\infty}, s_0 \sim \rho, s_{1:\infty} \sim P} \left[\sum_{t=0}^{\infty} \gamma^t R_t\right]$, with a discount factor of $\gamma \in [0, 1)$.

Mirror Learning Mirror learning (Kuba et al., 2024) is a theoretical framework that provides guarantees to a class of RL algorithms including PPO (Schulman et al., 2017) and underpins the architecture of LPO (Lu et al., 2022). A mirror learning algorithm updates a policy according to

$$\pi_{k+1} = \underset{\pi \sim \mathcal{N}(\pi_k)}{\arg\max}\, \mathbb{E}_{s \sim \beta_{\pi_k}} [A_{\pi_k}(s, a)] - \mathbb{E}_{s \sim \nu^{\pi}_{\pi_k}} [\mathcal{D}_{\pi_k}(\pi|s)], \tag{1}$$

where β_{π_k} and $\nu^{\pi}_{\pi_k}$ are sampling and drift distributions over s, and $A(s, a) = Q(s, a) - V(s)$ is the advantage. $\mathcal{D}$, the *drift* function, measures the difference between π and the current policy π_k and is used to penalise large policy updates. A valid drift function must uphold three conditions: be nonnegative everywhere; be zero at $\pi = \pi_k$; and have zero gradient with respect to π when $\pi = \pi_k$. For PPO, the drift function is $\text{ReLU}\left(\left[\frac{\pi(a|s)}{\pi_k(a|s)} - \text{clip}\left(\frac{\pi(a|s)}{\pi_k(a|s)}, 1 \pm \epsilon\right)\right] A_{\pi_k}(s, a)\right)$.

4 Meta-Learning Algorithms

In this section, we describe different meta-learning algorithms and qualitatively discuss possible pros and cons of each.

4.1 Black-Box Meta-Learning

Black-box algorithms are typically represented as neural networks. For example, a black-box learned optimiser might replace gradient descent with a neural network that maps from gradient to a parameter update. Most black-box algorithms are meta-trained using evolution or meta-gradients.

Meta-gradients are often calculated with backpropagation through time (BPTT) with respect to an RL objective, where the algorithm itself is treated as an agent (Oh et al., 2020) and updates are applied after fixed-length rollouts of the algorithm. Rollouts are usually truncated to prevent exploding or vanishing gradients, causing bias (Metz et al., 2022a; Wu et al., 2018). Although Jackson et al. (2024) demonstrate that evolution often learns better algorithms than meta-gradients, to provide diversity in our study, we use meta-gradients for learning LPG as proposed by Oh et al. (2020) and an evolutionary algorithm, evolution strategies (Wierstra et al., 2011; Salimans et al., 2017; Rechenberg, 1973, ES), for other algorithms.

ES is a population-based optimisation approach where a network's parameters, $\tilde{\theta}$, are iteratively updated using a natural gradient estimate for fitness $F(\cdot)$. This is calculated as $\nabla_\theta \mathbb{E}_{\epsilon\sim\mathcal{N}(0,I)} F(\theta + \sigma\epsilon) = \frac{1}{\sigma}\mathbb{E}_{\epsilon\sim\mathcal{N}(0,I)}\{\epsilon F(\theta+\sigma\epsilon)\}$ using a sample average for N sets of parameters sampled, with mean $\tilde{\theta}$. $\tilde{\theta}$ is updated with gradient ascent to maximise $F(\cdot)$, which is often defined as an agent's final return (Lu et al., 2022; Goldie et al., 2024). Unlike symbolic evolution, which is difficult to vectorise due to each program having a separate computation trace (see Section 4.3), the structure of ES can easily exploit GPU vectorisation for parallelisation (Lu et al., 2022; Lange, 2022b).

4.2 Black-Box Distillation

One way to improve the performance of black-box algorithms may be to distil the algorithm into another neural network, as introduced in Section 2.2. In our analysis, we consider two types of black-box distillation: distilling into a network with the *same* architecture (**Same-Size Distillation**); or distilling into a *smaller* network (**Smaller Distillation**), which we implement by halving all layer widths, such that the student underfits the teacher. Underfitting may help generalisation if the teacher has overfit to its original meta-training distribution, and distillation can itself provide learning regularisation (Zhang & Sabuncu, 2020; Mobahi et al., 2020).

We distil by using L_2 regression to match the student's outputs to the teacher for synthetically generated inputs, rather than sampling from the environment. This needs no additional environment transitions and introduces more diversity than sampling from RL training, which can lead to skewed distributions. We evaluate the RL performance of the student on the original meta-training environments periodically, and select the best-performing checkpoint as the distilled algorithm.

4.3 Symbolic Discovery

Evolutionary algorithms combine mutation, which randomly changes members of a population; crossover, which mixes two members of the population; and selection, which selects individuals from a population to pass to the next generation. When evolving an abstract syntax tree (AST), which represents a symbolic program, mutation adds or changes variables, constants or functions in the tree nodes and crossover swaps the nodes, and sometimes their children, between two ASTs.

Lion (Chen et al., 2023) is an interpretable symbolic learned optimiser discovered using symbolic evolution. However, symbolic search is inefficient and requires evaluating many functions (30,000 for a single seed in Lion, despite warm-starting from handcrafted optimisers) and, while computing fitness is quick for supervised learning, RL typically requires expensive environment simulation.

Though ES can be parallelised using a GPU, since the traced computation graphs of members in the population are the same, vectorising symbolic programs is more difficult as different programs have different computation graphs. Complex hand-coded branching logic could overcome this issue, but this would lead to huge performance degradation and significant inefficiency. Therefore, we do not include direct symbolic discovery in our empirical analysis.

4.4 Symbolic Distillation

Rather than evaluating symbolic programs in RL, a quicker approach is to distil black-box algorithms into symbolic programs, making the problem supervised. In addition to outputting interpretable functions, this approach may lead to better generalisation (Zheng et al., 2022; Chen et al., 2023). We base our approach on Zheng et al. (2022), who distil a learned optimiser into a symbolic program. We generate input data using the same statistics as in black-box distillation, albeit generated in one large batch to make the dataset stationary. We apply symbolic evolution (Section 4.3) using PySR (Cranmer, 2023) to find a program with low L_2 loss with respect to the black-box teacher outputs. While PySR has an in-built method for selecting algorithms based on a *combination* of high fitness and low complexity, we consistently find that choosing the most fit (i.e., lowest L_2 loss) function produces better RL results. As such, we select the algorithm that most accurately fits the teacher.

4.5 LLM Proposal

Since the rise of highly capable agentic language models, many researchers have used language models for algorithm discovery (e.g., (Lu et al., 2024; Faldor et al., 2024; Romera-Paredes et al., 2024; Hu et al., 2024; Song et al., 2024b)). Generally, this research is based on the premise that language models generate intelligent proposals, making them more sample efficient than symbolic evolution. As such, LLM-driven discovery pipelines generally evaluate on the order of tens of algorithms, rather than thousands, making them much more practical for evaluating directly in RL.

Since prompt tuning can play a large part in LLM performance, we build on an existing system, DiscoPOP (Lu et al., 2024), and warm-start search from a handcrafted algorithm. The LLM must reason in-context about previous algorithm performance to make suggestions for the next algorithm. In our setting, due to a number of unconventional inputs (particularly in the case of OPEN), we provide the LLM with a brief description of all inputs to the learned algorithm. After training, we select the best in-distribution algorithm for evaluation. We use GPT o3-mini (OpenAI, 2025) as our LLM, since it is a highly capable reasoning model with good performance for coding tasks.

5 Meta-Learned Algorithms

In this section, we introduce the set of meta-learned algorithms to which we apply the meta-learning algorithms introduced in Section 4. Due to the cost of meta-learning experiments, we are both selective and deliberate about which algorithms to include. We choose algorithms that condition on different numbers of inputs, are recursive or not, and affect different parts of RL training.

Learned Policy Optimisation LPO (Lu et al., 2022) is a learned algorithm that replaces the mirror drift function in PPO (Schulman et al., 2017; Kuba et al., 2024). The LPO network has no bias, to satisfy the mirror learning conditions at $r := \frac{\pi}{\pi_k} = 1$, and passes through a ReLU for nonnegativity. Inputs to LPO are transformations of r, the policy ratio, and A, the advantage typically calculated with generalised advantage estimation (GAE) (Schulman et al., 2018), and are defined as

$$\mathbf{x} = \left[(1-r), (1-r)^2, (1-r)A, (1-r)^2 A, \log(r), \log(r)A, \log(r)^2 A\right]. \tag{2}$$

We follow Lu et al. (2022) in initialising LPO near the PPO drift function to ease learning.

Learned Policy Gradient LPG (Oh et al., 2020) meta-learns a policy update rule based on actor-critic algorithms (Sutton & Barto, 2020), which update a policy (actor) using a learned value (critic). It is typically trained with meta-gradients (Oh et al., 2020; Jackson et al., 2024) and takes inputs of

$$\left[r_t, d_t, \gamma, \pi(a_t|s_t), y_\theta(s_t), y_\theta(s_{t+1})\right], \tag{3}$$

from fixed-length policy rollouts, using a backward-LSTM (Hochreiter & Schmidhuber, 1997). Here, r_t is a reward, d_t is a done flag, γ is a discount factor, $\pi(a_t|s_t)$ is the probability of taking action a_t in s_t and $y_\theta(\cdot)$ is an n-dimensional categorical distribution acting as a bootstrap vector.

Optimisation for Plasticity Loss, Exploration, and Non-stationarity OPEN (Goldie et al., 2024) is a meta-learned optimiser for RL that conditions on features measuring the presence of certain difficulties in RL optimisation, in addition to typical learned optimiser inputs (Metz et al., 2020). Its design takes into account: plasticity loss (Abbas et al., 2023; Lyle et al., 2023; Dohare et al., 2024), where an agent loses the ability to learn new things, which is overcome by conditioning OPEN on neuron dormancy (Sokar et al., 2023) and allowing it to behave differently on deeper layers in the agent; exploration (Cesa-Bianchi et al., 2017; Burda et al., 2018; Aubret et al., 2023; Sukhija et al., 2025), which prevents agents from getting trapped in local minima and which is boosted in OPEN by making the update slightly stochastic, as in noisy nets (Fortunato et al., 2019) or parameter space noise (Plappert et al., 2018); and non-stationarity (Igl et al., 2021b), which is measured based on how long training has gone on (like (Jackson et al., 2024)) and how many iterations have been spent optimising with a given data batch (similar to Ellis et al. (2024)). The full set of inputs are

$$\mathbf{x} = [p, g, m_{0.1}, m_{0.5}, m_{0.9}, m_{0.99}, m_{0.999}, m_{0.9999}, t\_p, b\_p, \text{dorm}, l\_p]\,, \tag{4}$$

where p is the current parameter, g is its gradient with respect to the PPO objective and m_x is an exponential moving average of gradient with scale x. g and m_x are both transformed as $x \to \{\log(|x| + \epsilon), \text{sgn}(x)\}$ to ease learning (Lan et al., 2024). The extra inputs are t_p and b_p, which measure time on the training and batch scale; dorm, which is the dormancy of the neuron downstream of the parameter; and l_p measures the depth of a parameter. The optimiser is applied to each parameter in a network independently. Whereas OPEN originally uses a *recurrent* architecture, here we explore the different meta-learning algorithms for *both* a feed-forward and recurrent OPEN.

No Features As in Goldie et al. (2024), we consider a 'No Features' learned optimiser that is similar to OPEN but includes only parameter, gradient, and momentum information. We include it as an example of a simple learned optimiser and hence only consider a feed-forward version of it.

6 Evaluation

There is no single measure of success for meta-learning algorithms. For instance, some users may choose to sacrifice some return for the sake of interpretability. Therefore, when comparing the different meta-learning algorithms, we consider a number of performance measures. In Section 8, we suggest design principles for future meta-learned algorithms with the following qualities in mind:

- In-distribution (i.d.) return, where we evaluate the algorithm on its meta-training task or tasks;
- Out-of-distribution (o.o.d.) return, where the algorithm is evaluated for meta-*test* generalisation to environments outside its training distribution;
- The sample cost of meta-learning, where training is stopped at peak in-distribution performance;
- The meta-train runtime (wall clock) for learning the algorithm;
- The meta-test runtime (wall clock); and
- How interpretable the algorithm is, judged subjectively as *low*, *medium*, or *high*.

For feed-forward algorithms, we meta-learn from both a *single* environment, Ant from Brax (Freeman et al., 2021; Todorov et al., 2012), and the *multiple* environments in MinAtar (Lange, 2022a; Young & Tian, 2019), following Goldie et al. (2024). These settings are selected to enable *fast* meta-learning without having overlap between the different meta-training distributions. For the recurrent implementation of OPEN, we use a pretrained optimiser from Goldie et al. (2024) instead of meta-training one ourselves, to allow for comparison against a publicly available baseline. Here, we focus only on the multiple environment setting to limit the cost of distillation, which is more expensive for recurrent algorithms. We meta-test these algorithms on a diverse set of environments: Freeway, Space Invaders, Asterix and Breakout from MinAtar (Lange, 2022a; Young & Tian, 2019)[2]; Humanoid, Hopper, Walker and Ant from Brax (Freeman et al., 2021; Todorov et al., 2012); Cartpole

[2]Seaquest is not available in the Gymnax implementation of MinAtar.

from OpenAI gym (Lange, 2022a; Brockman et al., 2016); and Craftax-Classic (Matthews et al., 2024; Hafner, 2021). For LPG, to align to prior research, we follow Oh et al. (2020) by meta-training on randomly distributed gridworlds and, as in Jackson et al. (2023), explore transfer to MinAtar. We specify all hyperparameters in Supplementary Material A, including any hyperparameters needed for the LLM proposed functions, which are tuned for the warm-start algorithm in each environment separately, before meta-training. Instead of a standardised evaluation set, we believe that our approach is more informative for the *actual* use cases of these meta-learned algorithms.

Due to the high cost and chance of failure, we do not apply symbolic distillation to recurrent algorithms. While Zheng et al. (2022) distil a recurrent learned optimiser with a single input using a fixed window of inputs, LPG has 19 inputs and OPEN has 20. For a window size of 20, as in Zheng et al. (2022), we would therefore require over 380 symbolic variables. Such a high dimensional problem is extremely difficult for symbolic regression to solve and would require so many AST nodes as to be computationally infeasible, given the search space grows exponentially with the size of the tree.

When plotting results, we normalise returns for each environment independently by dividing by the mean black-box learning score. Results are aggregated into 'In' and 'Out Of' Distribution based on the meta-training distribution and, unless otherwise stated, show the interquartile-mean (IQM) of return with 95% stratified bootstrap confident intervals for 16 environment seeds (Agarwal et al., 2021). In addition to understanding how well each method performs in- and out-of-distribution, our in-distribution results for distillation verify whether it was successful.

We include unnormalised and unaggregated results in Supplementary B, and the symbolic and LLM-proposed algorithms in Supplementary C. We show all initial LLM prompts in Supplementary D, and an example LLM discussion is in Supplementary E. We also provide extra results, from meta-training in gridworlds, in Supplementary F.

Due to the high cost of meta-learning, we follow standard procedure from the literature by meta-learning each algorithm for a single seed (Goldie et al., 2024; Metz et al., 2022b; Lan et al., 2024; Metz et al., 2019a) without meta-hyperparameter tuning.

7 Results

In this section, we present results from all experiments introduced in Section 6.

7.1 Learned Policy Optimisation

We firstly consider LPO, with results shown in Figure 2. We find that all distillation examples perform similarly, and often give minor generalisation gains without harming i.d. performance. Even though the LLM-proposed algorithms perform significantly than the others, they achieve the best o.o.d. performance. This is unsurprising: the LLM proposed algorithms in Supplementary C are both very similar and related to the warm-start function, PPO, and so are expected to generalise across a wide task distribution. Based on these results, LLM proposal is the best approach if generalisation is the priority. For an algorithm which performs well both i.d. and o.o.d., same-size distillation of a black-box algorithm is possibly the best option.

We visualise the gradients of all LPO functions in Supplementary G, following Lu et al. (2022).

7.2 Feed-Forward No Features

We show performance for the No Features optimiser in Figure 3. In Ant, the black-box optimisers fail to learn; based on Goldie et al. (2024), No Features is a weak learned algorithm for RL, making this failure unsurprising. This does highlight a clear limitation of distillation, though: if the original algorithm is poor, distillation is unlikely to *fix* it. Symbolic distillation also struggles, likely as the 8 inputs make this a relatively high dimensional problem for symbolic evolution. Overall, LLM proposal is by far the strongest baseline, both in-distribution and for generalisation.

Figure 2: IQM of final returns for LPO trained on Ant (top) and MinAtar (bottom). The distillation experiments all have similar returns, with same-size distillation offering improved generalisation over the algroithm learned in MinAtar. LLM Proposal is poor in-distribution, but has strong out-of-distribution generalisation performance.

The LLM likely performs well for a few reasons: gradient-based optimisation is well covered in the LLM's training corpus; all inputs to the optimiser are easy to understand; and the LLM has access to a per-environment learning rate tuned for its initialisation of SGD, which effectively relies on few-shot meta-test evaluation. The use of hyperparameters can be seen as an advantage, for flexibility, or disadvantage, if meta-test time samples are expensive.

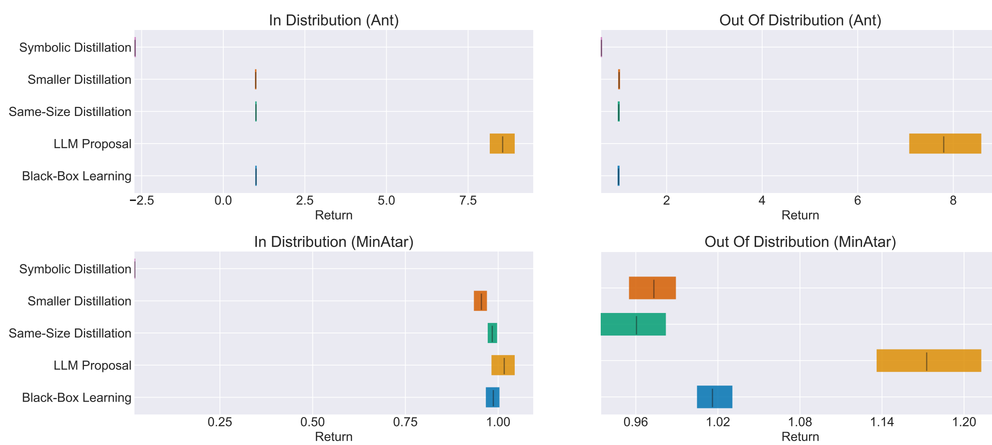


Figure 3: IQM of final returns for the No Features optimiser after meta-training in Ant (top) and MinAtar (bottom). Black-Box Learning struggles to learn in Ant, hurting the distilled optimiser performances. In MinAtar, symbolic distillation produced an optimiser which produces NaN returns in generalisation, so is omitted from the plot.

7.3 Feed-Forward OPEN

In Figure 4, we show the performance of a *feed-forward* implementation of OPEN after meta-training in Ant and MinAtar. OPEN has more inputs than the other algorithms analysed thus far, which is likely why the LLM and symbolic distillation catastrophically fail. Anecdotally, we find that symbolic distillation is unable to search the high dimensional space and instead converges to relatively simple, almost constant, algorithms, and the language model is unable to correctly use the additional input features despite explanations of their meaning. In fact, despite giving the LLM the shapes and ranges of all inputs, many of the algorithms it proposes in training produce errors.

Similar to No Features, distilling into a smaller model can worsen performance. However, same-size distillation produces a small generalisation benefit for the model trained on MinAtar. It is likely that the smaller model's representational capacity is too low, but that the regularisation effect of same-size distillation aids generalisation.

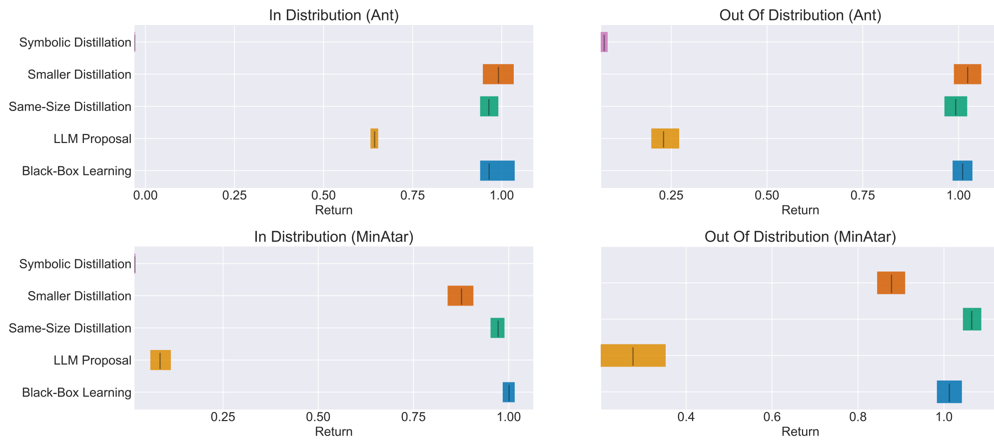


Figure 4: IQM of final returns for meta-training feed-forward OPEN in Ant (top) and MinAtar (bottom). Symbolic distillation from meta-learning in MinAtar caused NaNs out of distribution, so is omitted from the plot. The LLM and symbolic distillation both clearly struggle.

7.4 Recurrent LPG

In Figure 5, we explore the generalisation performance of meta-training LPG in grid-worlds for black-box learning and distillation only. Due to the formulation in LPG of y_θ as a categorical distribution, finding an algorithm grounded in literature to warm-start LLM proposals, as needed in DiscoPOP, is impractical. Therefore, as well as excluding symbolic distillation of LPG as a recurrent algorithm, we omit LLM Proposal and underscore this key limitation of LLM proposal: it needs *something* to start from, which may not always be available.

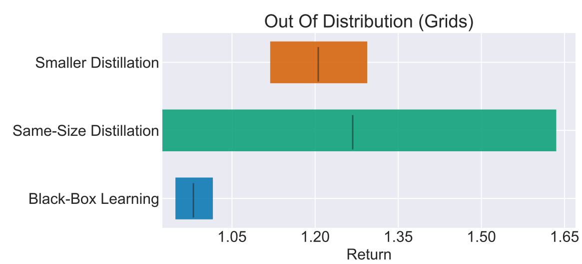


Figure 5: O.o.d. performance of LPG after training in gridworlds and transferring to MinAtar. Both types of distillation lead to higher mean performance, but wider confidence intervals.

In these results, distillation leads to improved IQM generalisation performance both when the student is smaller *and* the same size, albeit with overlapping confidence intervals when the student is the same size. Given that LPG uses a large network ($\sim$200k parameters, compared to $\sim$1k for OPEN), the regularisation from distillation likely helps reduce variance, improving generalisation.

7.5 Recurrent OPEN

Unlike LPG, which rolls out for only 20 steps at a time, OPEN unrolls over the *entire* course of RL training, which can potentially be tens of thousands of steps. As such, for stability and computational reasons, we cannot distil from data sequences as long as RL training. Instead, we distil a pretrained OPEN optimiser over 'Long Rollouts', where we train on sequences of 100 steps, and 'Short Rollouts', where the generated sequences are 20 steps long.

Figure 6 shows that distillation of recurrent OPEN is poor, suggesting that distilling an algorithm with long unrolls is too hard. This contrasts with feed-forward OPEN, where distillation occasionally helps and rarely hurts performance. LLM proposal, which was initialised with Adam (a better optimiser than SGD, which initialises feed-forward LLM proposed optimisers), produces a strong optimiser in o.o.d. environments than black-box learning. This is likely due to the fact that the best LLM algorithm is *very* similar to Adam, having been discovered early in training. It also uses a per-environment learning rate tuned for Adam and only uses extra features to have a per-layer learning rate; later attempts to incorporate more features lead to significantly worse performance. Overall, the black-box learning algorithm in this setting learns a performant but overfit algorithm and the LLM a simple but more general optimiser, although it does not change much from its initialisation.

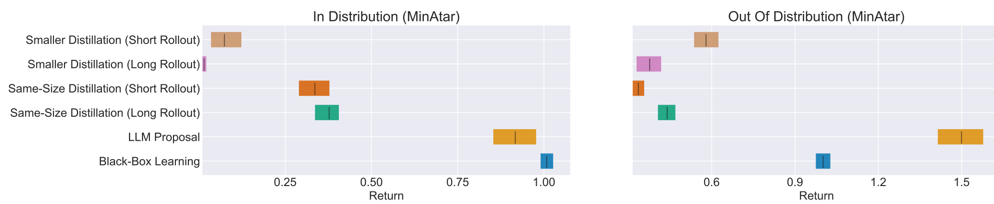


Figure 6: IQM of final returns for recurrent OPEN meta-trained in MinAtar. Distillation struggles due to the long unroll length of OPEN. The LLM performs well, but requires a tuned learning rate.

7.6 Additional Metrics

In this section we provide a more holistic evaluation of the different meta-learning algorithms, which is summarised in Table 1.

Black-box learning incurs a high sample cost since it requires many iterations of learning in an online environment, but distilling from this black-box algorithm requires no additional samples as distillation uses synthetic data. Since LLM proposal only evaluates individual algorithms produced by the language model, it requires comparatively few online interactions and is thus sample efficient.

In terms of speed, symbolic distillation can be the slowest of all techniques since its meta-training time scales exponentially with the maximum number of nodes in the AST, although it can be quicker for simpler algorithms. This contrasts with black-box distillation methods, whose speed remains broadly similar no matter the function being distilled. Using an LLM is fast, both because the search is warm-started from a known algorithm and as it only requires evaluating a small number of high quality algorithms, unlike the more random search in symbolic evolution.

Whereas black-box algorithms are almost completely uninterpretable, symbolic distillation and especially the LLM produce highly interpretable algorithms. This disparity arises because, whereas the LLM explains its proposals in plain-text at generation time, symbolic distillation generally introduces many constants into the equations that can obfuscate behaviour.

We find that symbolic distillation is unable to scale to functions with more than a small number of inputs. While LLM proposal is better, as it makes intelligent suggestions rather than randomly searching, we find that it is unable to incorporate all features from OPEN into a performant algorithm and requires warm-starting. Therefore, as the only meta-learning algorithm which can meta-train on long rollouts with many features, we believe black-box learning is the most scalable algorithm.

Approach	Samples	Train Time	Test Time	Interpretability	Scalability
Black-Box Learning	High	Slow	Slow	Bad	Best
Same-Size Distillation	No Extra	Slow	Medium	Bad	Good
Smaller Distillation	No Extra	Medium	Fast	Bad	Good
Symbolic Distillation	No Extra	Medium-Slow	Fast	Medium	Bad
LLM Proposal	Low	Fast	Fast	Good	Medium

Table 1: A summary of how each meta-learning algorithm performs across different metrics.

8 Design Principles

Based on the results in Section 7, we propose a number of design principles to incorporate into meta-learning pipelines moving forward. These are described below

- For a meta-learned algorithm with *few* inputs, or inputs which are easy to understand (i.e., an LLM can interpret them), prompting an LLM for new algorithms is a sample-efficient way to find new algorithms that generalise well. This has three caveats: there must be an easy-to-define, performant function from which to start the search; it must be possible to run hyperparameter tuning for the algorithm in the meta-test environment; and in-distribution performance of the algorithm will likely be worse than learning a black-box function (especially for high meta-samples).
- As long as it is possible to define a warm-start initialisation function, it is almost always better to prompt a language model for algorithm proposals over applying symbolic distillation. In fact, besides yielding interpretable functions, symbolic distillation is unlikely to improve performance, contrary to the suggestion of Chen et al. (2023) that symbolic functions should generalise better.
- Black-box distillation can often, but not always, improve generalisation. We recommend applying black-box distillation into the same-sized network for all black-box learned algorithms that are feed-forward or have short recurrent rollouts; given there is no increased sample cost and training is quick, this can occasionally yield cheap performance gains. On balance, smaller distillation can cause bigger drops in performance for smaller potential gains.
- Black-box algorithms are practically the only way to meta-learn algorithms which use a large number of features. If a meta-learned algorithm has many inputs, like OPEN, then an LLM is unlikely to propose a performant algorithm which also incorporates all of the input features.

9 Limitations and Future Work

There are a number of possible directions for future work. Firstly, while we discuss the reliance of LLM-proposed algorithms on hyperparameters, it would be interesting to explore *how* reliant the algorithms are on hyperparameter selection, and whether they are more sensitive than handcrafted algorithms, based on approaches like Adkins et al. (2025).

Secondly, an unexplored axis in our study is how the representation in black-box distillation affects performance. For instance, while we consider changing the black-box layer widths, we do not explore the effect of changing architectures entirely on performance. Inspired by work in algorithm distillation (Laskin et al., 2023; Son et al., 2025), it could be insightful to test distillation from recurrent or feed-forward algorithms to transformers (Vaswani et al., 2023).

Finally, we believe the findings presented here could be built upon by blending different meta learning algorithms. For instance, one avenue could test whether symbolic distillation scales better to high-dimensional problems if inputs were encoded by a black-box network, or whether LLMs could be warm-started from a symbolically distilled algorithm. Similarly, understanding the effect of different prompting styles would be a valuable addendum to this work.

10 Conclusion

This work presents a large-scale empirical analysis comparing many different meta-learning algorithms for RL: learning a black-box algorithm; distilling the algorithm into a same-size or smaller network; distilling the algorithm into a symbolic function; or prompting a language model to propose new algorithms. Based on our results, we propose a number of best-practice design principles for learning algorithms in RL. These include generally using language models for discovering new algorithms, so long as search can be initialised from something performant and it is possible to tune hyperparameters, and trying same-sized black-box distillation to potentially improve generalisation. These design suggestions can be used to ensure learned algorithms are as performant as possible for RL, while simultaneously reducing the need for unnecessary experiments.

References

Zaheer Abbas, Rosie Zhao, Joseph Modayil, Adam White, and Marlos C. Machado. Loss of Plasticity in Continual Deep Reinforcement Learning, March 2023. URL http://arxiv.org/abs/2303.07507.

Joshua Achiam, Ethan Knight, and Pieter Abbeel. Towards characterizing divergence in deep q-learning. *arXiv preprint arXiv:1903.08894*, 2019.

Jacob Adkins, Michael Bowling, and Adam White. A Method for Evaluating Hyperparameter Sensitivity in Reinforcement Learning, February 2025. URL http://arxiv.org/abs/2412.07165. arXiv:2412.07165 [cs].

Rishabh Agarwal, Max Schwarzer, Pablo Samuel Castro, Aaron Courville, and Marc G Bellemare. Deep reinforcement learning at the edge of the statistical precipice. *Advances in Neural Information Processing Systems*, 2021.

Diogo Almeida, Clemens Winter, Jie Tang, and Wojciech Zaremba. A Generalizable Approach to Learning Optimizers, June 2021. URL http://arxiv.org/abs/2106.00958.

Marcin Andrychowicz, Misha Denil, Sergio Gómez Colmenarejo, Matthew W. Hoffman, David Pfau, Tom Schaul, Brendan Shillingford, and Nando 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, pp. 3988–3996, Red Hook, NY, USA, 2016. Curran Associates Inc. ISBN 978-1-5108-3881-9.

Arthur Aubret, Laetitia Matignon, and Salima Hassas. An information-theoretic perspective on intrinsic motivation in reinforcement learning: A survey. *Entropy*, 25(2):327, 2023.

Jacob Beck, Risto Vuorio, Evan Zheran Liu, Zheng Xiong, Luisa Zintgraf, Chelsea Finn, and Shimon Whiteson. A Survey of Meta-Reinforcement Learning, August 2024. URL http://arxiv.org/abs/2301.08028.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI gym, 2016.

Yuri Burda, Harrison Edwards, Amos Storkey, and Oleg Klimov. Exploration by random network distillation. *arXiv preprint arXiv:1810.12894*, 2018.

Nicolò Cesa-Bianchi, Claudio Gentile, Gábor Lugosi, and Gergely Neu. Boltzmann exploration done right. *Advances in neural information processing systems*, 30, 2017.

Xiangning Chen, Chen Liang, Da Huang, Esteban Real, Kaiyuan Wang, Yao Liu, Hieu Pham, Xuanyi Dong, Thang Luong, Cho-Jui Hsieh, Yifeng Lu, and Quoc V. Le. Symbolic Discovery of Optimization Algorithms, May 2023. URL http://arxiv.org/abs/2302.06675.

Miles Cranmer. Interpretable Machine Learning for Science with PySR and SymbolicRegression.jl, May 2023. URL http://arxiv.org/abs/2305.01582. arXiv:2305.01582 [astro-ph].

Miles Cranmer, Alvaro Sanchez-Gonzalez, Peter Battaglia, Rui Xu, Kyle Cranmer, David Spergel, and Shirley Ho. Discovering Symbolic Models from Deep Learning with Inductive Biases, November 2020. URL http://arxiv.org/abs/2006.11287.

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jiawei Wang, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Shengfeng Ye, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wanjia Zhao, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning, 2025. URL https://arxiv.org/abs/2501.12948. _eprint: 2501.12948.

Michael Dennis, Natasha Jaques, Eugene Vinitsky, Alexandre Bayen, Stuart Russell, Andrew Critch, and Sergey Levine. Emergent Complexity and Zero-shot Transfer via Unsupervised Environment Design, February 2021. URL http://arxiv.org/abs/2012.02096.

Shibhansh Dohare, J Fernando Hernandez-Garcia, Qingfeng Lan, Parash Rahman, A Rupam Mahmood, and Richard S Sutton. Loss of plasticity in deep continual learning. *Nature*, 632(8026): 768–774, 2024.

Yan Duan, John Schulman, Xi Chen, Peter L Bartlett, Ilya Sutskever, and Pieter Abbeel. Rl^2: Fast reinforcement learning via slow reinforcement learning. *arXiv preprint arXiv:1611.02779*, 2016.

Benjamin Ellis, Matthew T. Jackson, Andrei Lupu, Alexander D. Goldie, Mattie Fellows, Shimon Whiteson, and Jakob Foerster. Adam on Local Time: Addressing Nonstationarity in RL with Relative Adam Timesteps, December 2024. URL http://arxiv.org/abs/2412.17113. arXiv:2412.17113 [cs].

Maxence Faldor, Jenny Zhang, Antoine Cully, and Jeff Clune. OMNI-EPIC: Open-endedness via Models of human Notions of Interestingness with Environments Programmed in Code, May 2024. URL http://arxiv.org/abs/2405.15568. arXiv:2405.15568 [cs].

Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *International conference on machine learning*, pp. 1126–1135. PMLR, 2017.

Meire Fortunato, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, Ian Osband, Alex Graves, Vlad Mnih, Remi Munos, Demis Hassabis, Olivier Pietquin, Charles Blundell, and Shane Legg. Noisy Networks for Exploration, July 2019. URL http://arxiv.org/abs/1706.10295. arXiv:1706.10295 [cs].

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.

Tommaso Furlanello, Zachary C. Lipton, Michael Tschannen, Laurent Itti, and Anima Anandkumar. Born Again Neural Networks, June 2018. URL http://arxiv.org/abs/1805.04770. arXiv:1805.04770 [stat].

Alexander David Goldie, Chris Lu, Matthew Thomas Jackson, Shimon Whiteson, and Jakob Nicolaus Foerster. Can Learned Optimization Make Reinforcement Learning Less Difficult?, July 2024. URL http://arxiv.org/abs/2407.07082.

Danijar Hafner. Benchmarking the spectrum of agent capabilities. *arXiv preprint arXiv:2109.06780*, 2021.

Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the Knowledge in a Neural Network, March 2015. URL http://arxiv.org/abs/1503.02531. arXiv:1503.02531 [stat].

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8): 1735–1780, November 1997. ISSN 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. URL https://doi.org/10.1162/neco.1997.9.8.1735.

Shengran Hu, Cong Lu, and Jeff Clune. Automated Design of Agentic Systems, August 2024. URL http://arxiv.org/abs/2408.08435.

Maximilian Igl, Gregory Farquhar, Jelena Luketina, Wendelin Boehmer, and Shimon Whiteson. Transient non-stationarity and generalisation in deep reinforcement learning. In *International Conference on Learning Representations*, 2021a. URL https://openreview.net/forum?id=Qun8fv4qSby.

Maximilian Igl, Gregory Farquhar, Jelena Luketina, Wendelin Boehmer, and Shimon Whiteson. Transient non-stationarity and generalisation in deep reinforcement learning. In *International conference on learning representations*, 2021b. URL https://openreview.net/forum?id=Qun8fv4qSby.

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, October 2023. URL http://arxiv.org/abs/2310.02782.

Matthew Thomas Jackson, Chris Lu, Louis Kirsch, Robert Tjarko Lange, Shimon Whiteson, and Jakob Nicolaus Foerster. Discovering temporally-aware reinforcement learning algorithms. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=MJJcs3zbmi.

Zhiwei Jia, Xuanlin Li, Zhan Ling, Shuang Liu, Yiran Wu, and Hao Su. Improving policy optimization with generalist-specialist learning. In *International Conference on Machine Learning*, pp. 10104–10119. PMLR, 2022.

Louis Kirsch and Jürgen Schmidhuber. Meta Learning Backpropagation And Improving It, March 2022. URL http://arxiv.org/abs/2012.14905.

Louis Kirsch, Sjoerd van Steenkiste, and Juergen Schmidhuber. Improving generalization in meta reinforcement learning using learned objectives. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=S1evHerYPr.

Jakub Grudzien Kuba, Christian Schroeder de Witt, and Jakob Foerster. Mirror Learning: A Unifying Framework of Policy Optimisation, November 2024. URL http://arxiv.org/abs/2201.02373. arXiv:2201.02373 [cs].

Qingfeng Lan, A. Rupam Mahmood, Shuicheng Yan, and Zhongwen Xu. Learning to Optimize for Reinforcement Learning, June 2024. URL http://arxiv.org/abs/2302.01470.

Robert Tjarko Lange. gymnax: A JAX-based reinforcement learning environment library, 2022a. URL http://github.com/RobertTLange/gymnax.

Robert Tjarko Lange. evosax: JAX-based Evolution Strategies. *arXiv preprint arXiv:2212.04180*, 2022b.

Michael Laskin, Luyu Wang, Junhyuk Oh, Emilio Parisotto, Stephen Spencer, Richie Steigerwald, DJ Strouse, Steven Stenberg Hansen, Angelos Filos, Ethan Brooks, maxime gazeau, Himanshu Sahni, Satinder Singh, and Volodymyr Mnih. In-context reinforcement learning with algorithm distillation. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=hy0a5MMPUv.

Joel Lehman, Jonathan Gordon, Shawn Jain, Kamal Ndousse, Cathy Yeh, and Kenneth O. Stanley. Evolution through Large Models, June 2022. URL http://arxiv.org/abs/2206.08896.

Pablo Lemos, Niall Jeffrey, Miles Cranmer, Shirley Ho, and Peter Battaglia. Rediscovering orbital mechanics with machine learning. *Machine Learning: Science and Technology*, 4(4):045002, 2023.

Chris Lu, Jakub Kuba, Alistair Letcher, Luke Metz, Christian Schroeder de Witt, and Jakob Foerster. Discovered policy optimisation. *Advances in Neural Information Processing Systems*, 35:16455–16468, 2022.

Chris Lu, Samuel Holt, Claudio Fanconi, Alex J. Chan, Jakob Foerster, Mihaela van der Schaar, and Robert Tjarko Lange. Discovering Preference Optimization Algorithms with and for Large Language Models, September 2024. URL http://arxiv.org/abs/2406.08414.

Andrei Lupu, Chris Lu, Jarek Luca Liesen, Robert Tjarko Lange, and Jakob Nicolaus Foerster. Behaviour Distillation. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=qup9xD8mW4.

Clare Lyle, Zeyu Zheng, Evgenii Nikishin, Bernardo Avila Pires, Razvan Pascanu, and Will Dabney. Understanding plasticity in neural networks, August 2023. URL http://arxiv.org/abs/2303.01486.

Michael Matthews, Michael Beukman, Benjamin Ellis, Mikayel Samvelyan, Matthew Jackson, Samuel Coward, and Jakob Foerster. Craftax: a lightning-fast benchmark for open-ended reinforcement learning. In *International conference on machine learning (ICML)*, 2024.

Tanner Mengel, Patrick Steffanic, Charles Hughes, Antonio Carlos Oliveira da Silva, and Christine Nattrass. Interpretable machine learning methods applied to jet background subtraction in heavy-ion collisions. *Physical Review C*, 108(2):L021901, 2023.

Luke Metz, Niru Maheswaranathan, Brian Cheung, and Jascha Sohl-Dickstein. Meta-Learning Update Rules for Unsupervised Representation Learning, February 2019a. URL http://arxiv.org/abs/1804.00222. arXiv:1804.00222 [cs, stat].

Luke Metz, Niru Maheswaranathan, Jeremy Nixon, C. Daniel Freeman, and Jascha Sohl-Dickstein. Understanding and correcting pathologies in the training of learned optimizers, June 2019b. URL http://arxiv.org/abs/1810.10180.

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, September 2020. URL http://arxiv.org/abs/2009.11243.

Luke Metz, C. Daniel Freeman, Samuel S. Schoenholz, and Tal Kachman. Gradients are Not All You Need, January 2022a. URL http://arxiv.org/abs/2111.05803. arXiv:2111.05803 [cs].

Luke Metz, James Harrison, C. Daniel Freeman, Amil Merchant, Lucas Beyer, James Bradbury, Naman Agrawal, Ben Poole, Igor Mordatch, Adam Roberts, and Jascha Sohl-Dickstein. VeLO: Training Versatile Learned Optimizers by Scaling Up, November 2022b. URL http://arxiv.org/abs/2211.09760.

Hossein Mobahi, Mehrdad Farajtabar, and Peter Bartlett. Self-distillation amplifies regularization in hilbert space. *Advances in Neural Information Processing Systems*, 33:3351–3361, 2020.

Johan Obando Ceron, Marc Bellemare, and Pablo Samuel Castro. Small batch deep reinforcement learning. *Advances in Neural Information Processing Systems*, 36:26003–26024, 2023.

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*, 33:1060–1070, 2020.

OpenAI. Openai o3-mini, January 2025. URL https://openai.com/index/openai-o3-mini/.

Emilio Parisotto, Francis Song, Jack Rae, Razvan Pascanu, Caglar Gulcehre, Siddhant Jayakumar, Max Jaderberg, Raphael Lopez Kaufman, Aidan Clark, Seb Noury, et al. Stabilizing transformers for reinforcement learning. In *International conference on machine learning*, pp. 7487–7498. PMLR, 2020.

Jack Parker-Holder, Minqi Jiang, Michael Dennis, Mikayel Samvelyan, Jakob Foerster, Edward Grefenstette, and Tim Rocktäschel. Evolving Curricula with Regret-Based Environment Design. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu, and Sivan Sabato (eds.), *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pp. 17473–17498. PMLR, July 2022. URL https://proceedings.mlr.press/v162/parker-holder22a.html.

Matthias Plappert, Rein Houthooft, Prafulla Dhariwal, Szymon Sidor, Richard Y. Chen, Xi Chen, Tamim Asfour, Pieter Abbeel, and Marcin Andrychowicz. Parameter Space Noise for Exploration, January 2018. URL http://arxiv.org/abs/1706.01905. arXiv:1706.01905 [cs].

Ingo Rechenberg. Evolutionsstrategie : Optimierung technischer systeme nach prinzipien der biologischen evolution. 1973. URL https://api.semanticscholar.org/CorpusID:60975248.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, January 2024. ISSN 1476-4687. DOI: 10.1038/s41586-023-06924-6. URL https://www.nature.com/articles/s41586-023-06924-6.

Andrei A. Rusu, Sergio Gomez Colmenarejo, Caglar Gulcehre, Guillaume Desjardins, James Kirkpatrick, Razvan Pascanu, Volodymyr Mnih, Koray Kavukcuoglu, and Raia Hadsell. Policy Distillation, January 2016. URL http://arxiv.org/abs/1511.06295. arXiv:1511.06295 [cs].

Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. Evolution Strategies as a Scalable Alternative to Reinforcement Learning, September 2017. URL http://arxiv.org/abs/1703.03864. arXiv:1703.03864 [cs, stat].

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms, August 2017. URL http://arxiv.org/abs/1707.06347.

John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-Dimensional Continuous Control Using Generalized Advantage Estimation, October 2018. URL http://arxiv.org/abs/1506.02438.

Ghada Sokar, Rishabh Agarwal, Pablo Samuel Castro, and Utku Evci. The Dormant Neuron Phenomenon in Deep Reinforcement Learning, June 2023. URL http://arxiv.org/abs/2302.12902.

Jaehyeon Son, Soochan Lee, and Gunhee Kim. Distilling Reinforcement Learning Algorithms for In-Context Model-Based Planning, February 2025. URL http://arxiv.org/abs/2502.19009. arXiv:2502.19009 [cs].

Xiaotian Song, Peng Zeng, Yanan Sun, and Andy Song. Generalizable Symbolic Optimizer Learning. 2024a.

Xingyou Song, Yingtao Tian, Robert Tjarko Lange, Chansoo Lee, Yujin Tang, and Yutian Chen. Position: Leverage Foundational Models for Black-Box Optimization, May 2024b. URL http://arxiv.org/abs/2405.03547.

Bhavya Sukhija, Stelian Coros, Andreas Krause, Pieter Abbeel, and Carmelo Sferrazza. MaxinfoRL: Boosting exploration in reinforcement learning through information gain maximization. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=R4q3cY3kQf.

Richard S. Sutton and Andrew Barto. *Reinforcement learning: an introduction*. Adaptive computation and machine learning. The MIT Press, Cambridge, Massachusetts London, England, second edition edition, 2020. ISBN 978-0-262-03924-6.

Hongyao Tang and Glen Berseth. Improving deep reinforcement learning by reducing the chain effect of value and policy churn. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://openreview.net/forum?id=cQoAgPBARc.

Emanuel Todorov, Tom Erez, and Yuval Tassa. MuJoCo: A physics engine for model-based control. In *2012 IEEE/RSJ international conference on intelligent robots and systems*, pp. 5026–5033. IEEE, 2012. DOI: 10.1109/IROS.2012.6386109.

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.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention Is All You Need, August 2023. URL http://arxiv.org/abs/1706.03762. arXiv:1706.03762 [cs].

Tongzhou Wang, Jun-Yan Zhu, Antonio Torralba, and Alexei A. Efros. Dataset Distillation, February 2020. URL http://arxiv.org/abs/1811.10959. arXiv:1811.10959 [cs].

Daan Wierstra, Tom Schaul, Tobias Glasmachers, Yi Sun, and Jürgen Schmidhuber. Natural Evolution Strategies, June 2011. URL http://arxiv.org/abs/1106.4487.

Yuhuai Wu, Mengye Ren, Renjie Liao, and Roger Grosse. Understanding short-horizon bias in stochastic meta-optimization. In *International Conference on Learning Representations*, 2018. URL https://openreview.net/forum?id=H1MczcgR-.

Kenny Young and Tian Tian. MinAtar: An atari-inspired testbed for thorough and reproducible reinforcement learning experiments. *arXiv preprint arXiv:1903.03176*, 2019.

Zhilu Zhang and Mert Sabuncu. Self-distillation as instance-specific label smoothing. *Advances in Neural Information Processing Systems*, 33:2184–2195, 2020.

Wenqing Zheng, Tianlong Chen, Ting-Kuei Hu, and Zhangyang Wang. Symbolic Learning to Optimize: Towards Interpretability and Scalability, May 2022. URL http://arxiv.org/abs/2203.06578.

Supplementary Materials

The following content was not necessarily subject to peer review.

A Hyperparameters

In this section, we provide all hyperparameters used in this paper. Due to the number of experiments used here, we break our list of hyperparameters into multiple subsections. To prevent unnecessary hyperparameter tuning, where our implementations are based in open-source releases from other works we replicate their hyperparameters.

A.1 Policy Optimisation Hyperparameters

The learned optimiser experiments (i.e. OPEN, No Feat) use PPO for policy optimisation, and use the same hyperparameters as the LPO experiments. Our PPO hyperparameters are largely grounded in Lu et al. (2022) and Goldie et al. (2024), and are show in Table 2.

Note that, for LPO, only the LLM proposals get access to *PPO Clip* ϵ as it needs to be initialised at PPO.

Table 2: PPO and LPO hyperparameters. The Brax and MinAtar suites used common PPO parameters.

Hyperparameter	**Environment**			
	MinAtar	**Brax**	**Cartpole**	**Craftax**
Number of Environments N_{envs}	64	2048	4	1024
Number of Environment Steps N_{steps}	128	10	128	20
Total Timesteps T	$1e7$	$3e7$	$5e5$	$3e7$
Number of Minibatches $N_{minibatch}$	8	32	4	16
Number of Epochs L	4	4	4	2
Discount Factor γ	0.99	0.99	0.99	0.99
GAE λ	0.95	0.95	0.95	0.95
PPO Clip ϵ	0.2	0.2	0.2	0.2
Value Function Coefficient c_1	0.5	0.5	0.5	0.5
Entropy Coefficient c_2	0.01	0.0	0.01	0.01
Max Gradient Norm	0.5	0.5	0.5	0.5
Layer Width W	64	64	64	64
Number of Hidden Layers H	2	2	2	2
Activation	relu	tanh	tanh	tanh

LPG uses a different set of hyperparameters since it has a different algorithmic backbone. We use hyperparameters from Jackson et al. (2023). We use the 'all_shortlife' class of gridworlds for meta-training, and show the LPG hyperparameters in Table 3.

A.2 Optimiser Hyperparameters

Depending on whether the algorithm was feed-forward or recurrent, the learned optimisers require a per-environment learning rate tuned for either SGD or Adam. We provide all optimiser hyperparameters for LPO and PPO with learned optimisers in Tables 4-6, and for LPG in Table 7. For LPO and PPO, we tune optimiser hyperparameters *individually* per environment. We round all values for SGD. LPO uses a slightly different learning rate than the learned optimisers in some cases, since we used standard β values of $[\beta_1 = 0.9, \beta_2 = 0.999]$ for the learned optimisers but tuned them for LPO, as they are not part of the learned algorithm. All learning rates use linear annealing over the course of training.

Table 3: Hyperparameters for policy optimisation and the agent in LPG experiments.

Hyperparameter	Environment	
	Gridworld	**MinAtar**
Number of Environments N_{envs}	64	64
Number of Environment Steps N_{steps}	20	20
Total Timesteps T	$3e6$	$1e7$
Number of Minibatches $N_{minibatch}$	64	64
Discount Factor γ	0.99	0.99
GAE λ	0.95	0.95
Entropy Coefficient c_2	0.01	0.01
Max Gradient Norm	0.5	1.0
Layer Width W	32	32 (conv)
Number of Hidden Layers H	1	2
Activation	relu	relu

Table 4: PPO and LPO hyperparameters for MinAtar environments.

Hyperparameter	Environment			
	Asterix	**Breakout**	**Freeway**	**SpaceInvaders**
LPO Learning Rate	$3e{-}3$	$1e{-}2$	$1e{-}3$	$7e{-}3$
β_1	0.9	0.9	0.9	0.9
β_2	0.999	0.99	0.99	0.99
SGD Learning Rate	0.52	1.02	0.56	1.17
L2O Adam Learning Rate	$3e{-}3$	$7e{-}3$	$1e{-}3$	$3e{-}3$

Table 5: PPO and LPO hyperparameters for Brax environments.

Hyperparameter	Environment			
	Ant	**Humanoid**	**Walker**	**Hopper**
LPO Learning Rate	$3e{-}4$	$3e{-}4$	$1e{-}3$	$8e{-}4$
β_1	0.99	0.9	0.9	0.9
β_2	0.99	0.999	0.999	0.999
SGD Learning Rate	0.17	0.053	0.52	0.27
L2O Adam Learning Rate	$3e{-}4$	$3e{-}4$	$1e{-}3$	$8e{-}4$

Table 6: PPO and LPO hyperparameters for Cartpole and Craftax.

Hyperparameter	Environment	
	Cartpole	**Craftax**
LPO Learning Rate	$1e{-}3$	$5e{-}4$
β_1	0.9	0.9
β_2	0.999	0.999
SGD Learning Rate	$2.5e{-}4$	0.46
L2O Adam Learning Rate	$3e{-}3$	$5e{-}4$

Table 7: LPG optimiser hyperparameters.

Hyperparameter	Environment	
	Gridworld	**MinAtar**
Learning Rate	$1e{-}3$	$5e{-}4$

A.3 Meta-Learning Hyperparameters

In tables 8 and 9 we provide all necessary hyperparameters for *meta-learning*. In table 10, we include hyperparamters for symbolic distillation. We do not tune hyperparameters for black-box learning due to the computational cost of meta-learning. We run a small sweep over learning rates for black-box distillation. For symbolic distillation, we mostly follow the implementations in Cranmer (2023), albeit using a custom set of possible symbolic functions and generally allowing more complex programs.

For distillation, we sweep over learning rates in $[0.1, 0.02, 0.001]$. For smaller distillation, we halve all layer widths.

Table 8: Meta-learning hyperparameters for LPO and learned optimisers.

Hyperparameter	**Meta-Learned Algorithm**			
	LPO	**No Features**	**Feed-Forward OPEN**	**Recurrent OPEN**
ES Learning Rate	$3e{-2}$	$3e{-2}$	$3e{-2}$	–
ES LR Decay	0.999	0.999	0.999	–
ES σ_{init}	$3e{-2}$	$3e{-2}$	$3e{-2}$	–
ES σ_{decay}	0.999	0.999	0.999	–
Population Size	64	64	64	–
Number Dense Layers	1	2	2	2
GRU Size (MinAtar)	–	–	–	16
Dense Layer Size (MinAtar)	128	32	32	32
GRU Size (Ant)	–	–	–	8
Dense Layer Size (Ant)	128	16	16	16

Table 9: Meta-learning Hyperparameters for LPG, following Jackson et al. (2023)

Hyperparameter	**Environment Gridworld**
Num Steps	5000
Embedding Width	16
GRU width	256
Target Width	8
Agent Target KL Divergence	0.5
Learning Rate	$1e{-4}$
LPG Max Grad Norm	0.55
Num Agent Updates	5
LPG Policy Entropy Coeff	$5e{-2}$
LPG Target Entropy Coeff	$1e{-3}$
LPG Policy L_2 *Coeff*	$5e{-3}$
LPG Target L_2 *Coeff*	$1e{-3}$

Table 10: Symbolic Distillation Hyperparameters, following Cranmer (2023). We use warm-starting after every RL evaluation of the best fit algorithm; as such, while the PySR 'Number Iterations' is 10, we loop over this process 40 times (effectively leading to 400 iterations).

Hyperparameter	**Environment**	
	LPO	**Feed-Forward OPEN/No Features**
Max Size	40	60
Populations 31	160	
Number Iterations	10	10
Batch Size	5000	5000
Weight Optimise	0.001	0.001

B Returns by Environment

In this section, we include plots of the returns achieved by all learned algorithms in all of the environments we test. Unlike in the main body of the paper, we do not aggregate any results here.

All algorithms were run for 16 environment seeds. We plot IQM with 95% stratified bootstrap confidence intervals, following (Agarwal et al., 2021). For clarity, we separate the returns into two rows; each pair of rows corresponds to a single trained algorithm.

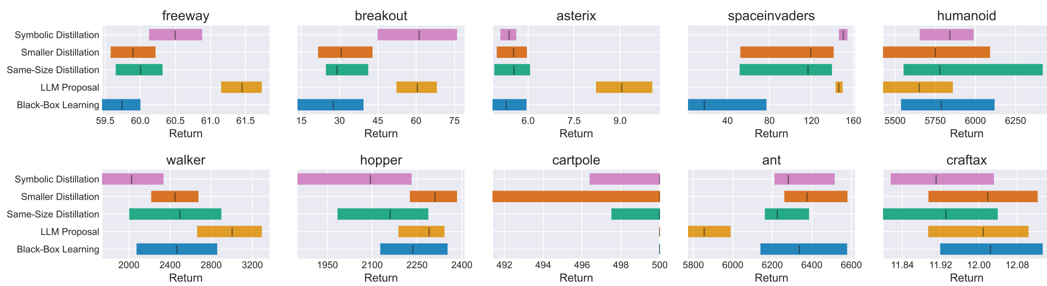


Figure 7: IQM of final meta-test returns after meta-training LPO in Ant.

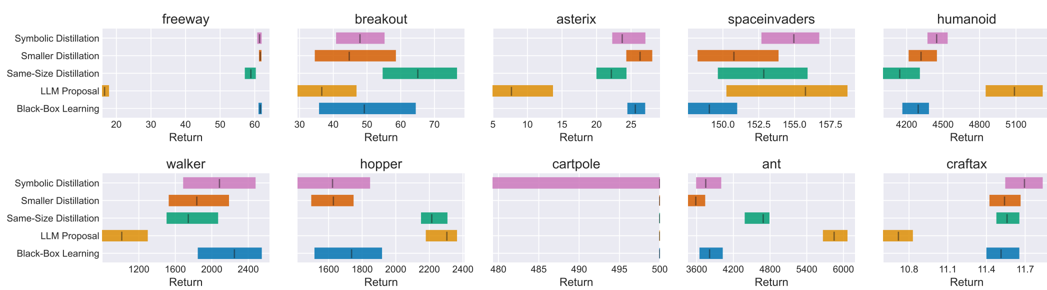


Figure 8: IQM of final meta-test returns for LPO meta-trained in MinAtar.

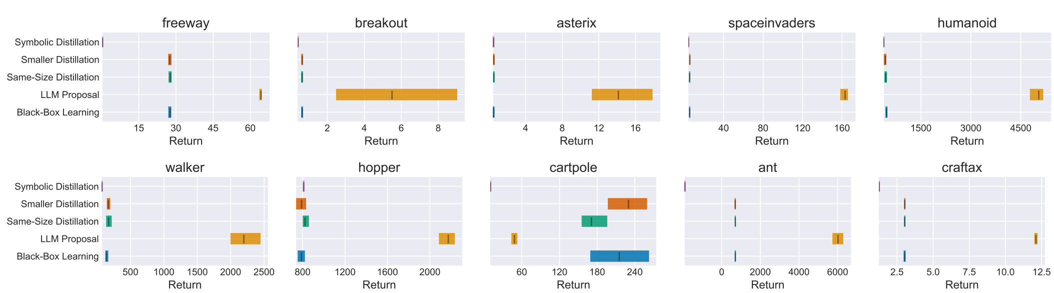


Figure 9: IQM of final returns after meta-training the No Features optimizer in Ant.

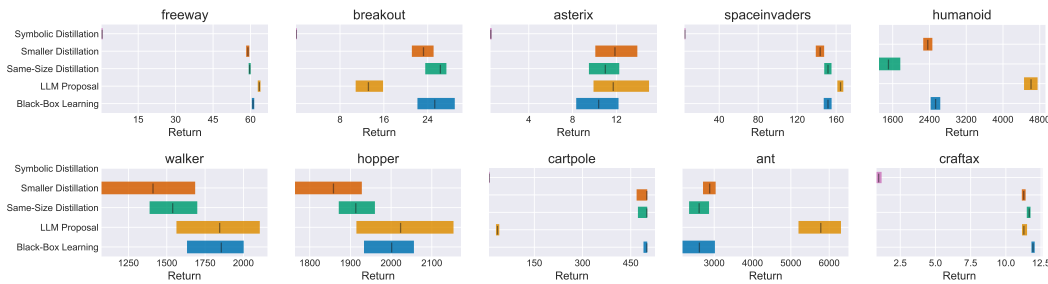


Figure 10: IQM of final returns from meta-training the No Features optimizer in MinAtar.

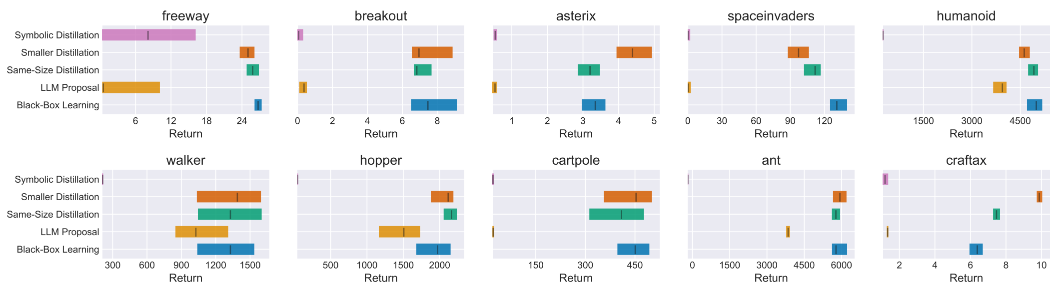


Figure 11: IQM of final returns for meta-training feed-forward OPEN in Ant.

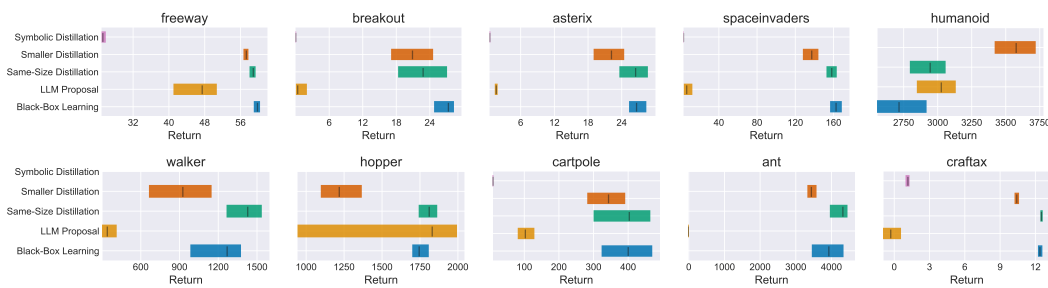


Figure 12: IQM of final returns for meta-training feed-forward OPEN in MinAtar.

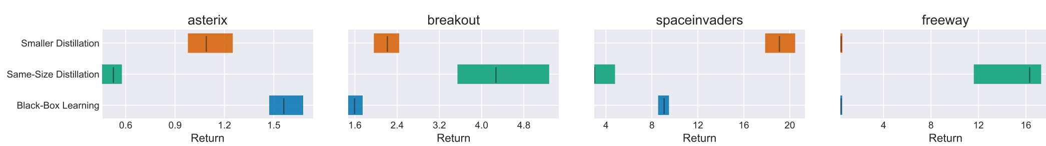


Figure 13: IQM of final returns for meta-training Recurrent LPG in Grids.

Figure 14: IQM of final returns for meta-training recurrent OPEN in MinAtar.

C Symbolic and LLM-Proposed Functions

In this section, we include the functions discovered by both symbolic distillation and LLM proposal.

C.1 Symbolic Distillation Functions

Firstly, we consider the functions discovered by symbolic distillation. For readability, we have shortened all constants in the functions to two significant figures and reformatted the discovered programs. In practice, the symbolic functions are defined in a single line. For LPO, we enforce that functions should be defined in terms of $\log(r)$ and $(r-1)$ to bias search towards valid programs.

For OPEN, rand is a randomly sampled noise variable. In OPEN, this is applied at the output of the black-box algorithm.

Discovered symbolic program for LPO after meta-training in Ant.

```
def LPO_Symbolic(r, A):
   log_r = log(r)
   r_minus_1 = r - 1

   numerator = -(
      tanh(math.tanh(log_r) ** 2) + (-log_r * 0.99) ** 2
   )

   denominator = min(
      tanh(abs(A) * tanh(-0.19)),
      min(A - 0.50, r_minus_1, -0.53) /
      (abs((-0.80) ** 2) / abs(0.50))
   )

   return numerator / denominator
```

Discovered symbolic program for LPO after meta-training in MinAtar.

```
def LPO_Symbolic(r, A):
   log_r = log(r)
   r_minus_1 = r - 1

   term1 = min(0.15, r_minus_1 * -0.97)
   term2 = min(-0.28 - log_r, A ** 2)
   term3 = max(term1, term2) * tanh(abs(A - 0.46))

   numerator = (term3 ** 2) + tanh((r_minus_1 * A) / max(1.32 ** 2, A - log_r))

   return max(numerator, -0.86)
```

Discovered symbolic program for No Features optimiser after meta-training in Ant.

```
def No_Feat_Symbolic(p, g, m_1, m_5, m_9, m_99, m_999, m_9999):
   coef = 0.00030 / sin(cos(relu(-0.06)))

   numerator = (p / 0.87) + ((g + m_999) + ((m_5 - 0.32) + tanh(m_99)))
   denominator = relu(1.60)

   return coef * tanh(exp(numerator / denominator))
```

Discovered symbolic program for No Features optimiser after meta-training in MinAtar.

```
def No_Feat_Symbolic(p, g, m_1, m_5, m_9, m_99, m_999, m_9999):
   term1 = (-3.89 - m_99) - (m_999 - (tanh(-0.60) * g))
   term2 = m_5 + (m_9 + 0.18)
```

```
    return relu(tanh((term1 - term2) * -0.00037))
```

Discovered symbolic program for Feed-forward OPEN after meta-training in Ant.

```
def OPEN_Symbolic(p, log_g, sgn_g, log_m_1, sgn_m_1, log_m_5,
sgn_m_5, log_m_9, sgn_m_9, log_m_99, sgn_m_99, log_m_999,
sgn_m_999, log_m_9999, sgn_m_9999, t_p, b_p, dorm, l_p, rand
):
    term1 = tanh(exp(abs(log_m_999))) * rand
    term2 = (term1 / (1.11**2)) * -0.00041

    return -0.00041 + (term2 / 2.36)
```

Discovered symbolic program for Feed-forward OPEN after meta-training in MinAtar.

```
def OPEN_Symbolic(p, log_g, sgn_g, log_m_1, sgn_m_1, log_m_5,
sgn_m_5, log_m_9, sgn_m_9, log_m_99, sgn_m_99, log_m_999,
sgn_m_999, log_m_9999, sgn_m_9999, t_p, b_p, dorm, l_p, rand
):
    term1 = sin(1.04) + (sgn_p / 1.32)
    term2 = ((log_g + sgn_m_99) + (log_m_99 / 1.24)) * 1.28
    exponent = (term1 + term2) * 0.12

    return (exp(exponent) * 0.00057) - tanh(0.00020 * rand)
```

C.2 LLM Proposed Functions

In this subsection, we provide the best function proposed by the LLM for each of the meta-training tasks. Unlike the symbolic search, these algorithms were all warm-started from a handcrafted algorithm. All functions are named and annotated by the LLM; we make no edits.

LLM proposed function for LPO after meta-training in Ant.

```
def log_clip_penalty_v2(
    ratio: float,
    advantage: float,
    epsilon: float
) -> jnp.ndarray:
    # Compute the log deviation from identity
    delta = jnp.log(ratio)

    # Compute the penalized deviation for positive advantage when delta > epsilon
    positive_term = jnp.where(
        (advantage > 0) & (delta > epsilon),
        jnp.abs(advantage) * (delta - epsilon) ** 2,
        0.0
    )

    # Compute the penalized deviation for negative advantage when delta < -epsilon
    negative_term = jnp.where(
        (advantage < 0) & (delta < -epsilon),
        jnp.abs(advantage) * (-epsilon - delta) ** 2,
        0.0
    )

    # Sum the contributions to obtain the drift
    drift = positive_term + negative_term
    return drift
```

LLM proposed function for LPO after meta-training in MinAtar.

```
def log_poly_clip(
    ratio: float,
    advantage: float,
    epsilon: float
) -> 'jnp.ndarray':
    import jax.numpy as jnp

    # Compute normalized log deviation; at ratio=1, x=0
    x = jnp.log(ratio) / epsilon

    # Define a blended polynomial-logarithmic factor
    # At x=0, factor = 0 and its derivative is also 0, since a Taylor expansion yields factor ~ x^4/8
    factor = (x ** 2) / 2.0 - jnp.log(1.0 + (x ** 2) / 2.0)

    # Multiply by advantage and enforce non-negativity
    drift = jnp.maximum(0.0, factor * advantage)

    return drift
```

LLM proposed function for No Features optimiser after meta-training in Ant.

```
def HybridBetaMomentum(
    p: jnp.ndarray,
    m_0_1: jnp.ndarray,
    m_0_5: jnp.ndarray,
    m_0_9: jnp.ndarray,
    m_0_99: jnp.ndarray,
    m_0_999: jnp.ndarray,
    m_0_9999: jnp.ndarray,
    g: jnp.ndarray,
    lr: float
) -> jnp.ndarray:
    # Blend medium-term momentum and very long-term momentum
    m_blend = 0.7 * m_0_5 + 0.3 * m_0_999

    # Compute a dynamic scaling factor based on sign alignment
    # If the gradient and blended momentum agree in sign, use 1.05 to slightly amplify,
    # otherwise use 0.95 to dampen the update
    scale_factor = jnp.where(g * m_blend > 0, 1.05, 0.95)

    # Combine the gradient with the blended momentum modulated by the dynamic scaling factor
    combined_update = scale_factor * (g + m_blend)

    # Return the update scaled by the learning rate
    update = lr * combined_update
    return update
```

LLM proposed function for No Features optimiser after meta-training in MinAtar.

```
def relative_magnitude_update(
    p: jnp.ndarray,
    m_0_1: jnp.ndarray,
    m_0_5: jnp.ndarray,
    m_0_9: jnp.ndarray,
    m_0_99: jnp.ndarray,
    m_0_999: jnp.ndarray,
    m_0_9999: jnp.ndarray,
    g: jnp.ndarray,
    lr: float
```

```python
) -> jnp.ndarray:
    import jax.numpy as jnp
    import jax.nn as jnn
    eps = 1e-8

    # Stack momentum buffers and compute the average momentum
    moments = jnp.stack([m_0_1, m_0_5, m_0_9, m_0_99, m_0_999, m_0_9999], axis=0)
    m_avg = jnp.mean(moments, axis=0)

    # Compute the ratio of the absolute values of average momentum to the current gradient
    ratio = jnp.abs(m_avg) / (jnp.abs(g) + eps)

    # Compute elementwise blending weight using a sigmoid with steepness parameter s
    s = 5.0 # Controls how quickly blending switches based on the ratio
    # When ratio > 1, momentum is stronger; weight will be near 1, otherwise near 0
    weight = jnn.sigmoid(s * (ratio - 1.0))

    # Blend the average momentum and the current gradient based on the weight
    blended = weight * m_avg + (1 - weight) * g

    # Scale by the learning rate
    update = lr * blended
    return update
```

LLM proposed function for Feed-forward OPEN after meta-training in Ant.

```python
def robust_extrapolated(
    p: jnp.ndarray,
    m_0_1: jnp.ndarray,
    m_0_5: jnp.ndarray,
    m_0_9: jnp.ndarray,
    m_0_99: jnp.ndarray,
    m_0_999: jnp.ndarray,
    m_0_9999: jnp.ndarray,
    l_p: jnp.ndarray,
    b_p: jnp.ndarray,
    t_p: jnp.ndarray,
    dorm: jnp.ndarray,
    g: jnp.ndarray,
    rand: jnp.ndarray,
    lr: float,
) -> jnp.ndarray:
    epsilon = 1e-7

    # Step 1: Aggregate momentum across multiple timescales
    aggregated_mom = (m_0_1 + m_0_5 + m_0_9 + m_0_99 + m_0_999 + m_0_9999) / 6.0

    # Step 2: Compute aggregated squared momentum and estimate variance
    aggregated_sq = (jnp.square(m_0_1) + jnp.square(m_0_5) + jnp.square(m_0_9) +
                     jnp.square(m_0_99) + jnp.square(m_0_999) + jnp.square(m_0_9999)) / 6.0
    variance = jnp.maximum(aggregated_sq - jnp.square(aggregated_mom), epsilon)
    std_est = jnp.sqrt(variance) + epsilon

    # Step 3: Derive a confidence measure from the signal-to-noise ratio
    confidence = jnp.tanh(jnp.abs(aggregated_mom) / std_est)

    # Step 4: Blend the raw gradient with the aggregated momentum (Polyak heavy-ball style)
    weighted_update = 0.5 * g + 0.5 * aggregated_mom

    # Step 5: Scale the blended update by (1 + confidence) to favor high-confidence updates
```

```
    adjusted_update = weighted_update * (1.0 + confidence)

    # Step 6: Add decaying random noise for exploration
    noise_weight = 0.01 * (1.0 - t_p) # more noise early in training
    noise = noise_weight * rand

    # Step 7: Combine the adjusted update with noise
    combined_update = adjusted_update + noise

    # Step 8: Compute damping factors based on training/batch progress and layer depth
    progress_scaling = jnp.exp(-0.5 * (b_p + t_p))
    layer_scaling = 1.0 - l_p

    # Step 9: Final update with normalization by dormancy
    update = lr * combined_update * progress_scaling * layer_scaling / (dorm + epsilon)

    # Step 10: Ensure numerical stability by replacing NaNs or infinities
    update = jnp.nan_to_num(update, nan=0.0, posinf=1e7, neginf=-1e7)
    return update
```

LLM proposed function for Feed-forward OPEN after meta-training in MinAtar.

```
def Power_Sign_Adaptive(
    p: jnp.ndarray,
    m_0_1: jnp.ndarray,
    m_0_5: jnp.ndarray,
    m_0_9: jnp.ndarray,
    m_0_99: jnp.ndarray,
    m_0_999: jnp.ndarray,
    m_0_9999: jnp.ndarray,
    l_p: jnp.ndarray,
    b_p: jnp.ndarray,
    t_p: jnp.ndarray,
    dorm: jnp.ndarray,
    g: jnp.ndarray,
    rand: jnp.ndarray,
    lr: float,
) -> jnp.ndarray:
    import jax.numpy as jnp
    import jax

    # Compute effective momentum as a weighted average of historic momenta
    eff_mom = (0.1 * m_0_1 +
            0.15 * m_0_5 +
            0.2 * m_0_9 +
            0.25 * m_0_99 +
            0.2 * m_0_999 +
            0.1 * m_0_9999)

    # Blend raw gradient and effective momentum using an exponential decay based on b_p
    blend_weight = jnp.exp(-b_p)
    combined = blend_weight * g + (1.0 - blend_weight) * eff_mom

    # Compute adaptive exponent which transitions from 0.5 (sign-driven update) early to 1.0 later
    exponent = 0.5 + 0.5 * t_p # when t_p=0 -> exponent=0.5, when t_p=1 -> exponent=1.0

    # Apply the power sign transformation: preserve sign, raise magnitude to the adaptive exponent
    power_sign_update = jnp.sign(combined) * (jnp.abs(combined) ** exponent)

    # Scale update by layer depth: deeper layers receive relatively larger updates
```

```
    layer_scale = 1.0 + l_p

    # Adjust for neuron dormancy, ensuring a minimum value of 1 to avoid division by zero
    dorm_factor = jnp.maximum(dorm, 1.0)

    # Add small stochastic noise that decays with training progress for exploration
    noise = 0.005 * rand * (1.0 - t_p)

    update = lr * power_sign_update * layer_scale / dorm_factor + noise
    return update
```

LLM proposed function for Recurrent OPEN after meta-training in MinAtar.

```
def AdaptiveLayerRAdam(
    p: jnp.ndarray,
    m_0_1: jnp.ndarray,
    m_0_5: jnp.ndarray,
    m_0_9: jnp.ndarray,
    m_0_99: jnp.ndarray,
    m_0_999: jnp.ndarray,
    m_0_9999: jnp.ndarray,
    l_p: jnp.ndarray,
    b_p: jnp.ndarray,
    t_p: jnp.ndarray,
    dorm: jnp.ndarray,
    g: jnp.ndarray,
    rand: jnp.ndarray,
    lr: float,
    iteration: float,
    var: jnp.ndarray
) -> jnp.ndarray:
    # Update the running variance with a decay factor of 0.99
    decay = 0.99
    var_new = decay * var + (1 - decay) * jnp.square(g)
    var_hat = var_new / (1 - decay**iteration)

    # Compute bias-corrected momentum estimates
    # Fast momentum: m_0_1 corresponds to a fast adaptation (beta=0.1 assumed)
    m_fast = m_0_1 / (1 - 0.1**iteration)
    # Slow momentum: m_0_9 corresponds to a slower running average (beta=0.9 assumed)
    m_slow = m_0_9 / (1 - 0.9**iteration)

    # Combine momentum estimates based on layer proportion (l_p)
    # Shallow layers (l_p close to 0) favor slow momentum, deeper layers favor fast momentum
    momentum = l_p * m_fast + (1 - l_p) * m_slow

    # Implement RAdam style variance rectification
    beta = 0.9
    rho_inf = 2 / (1 - beta) - 1 # Typically 19 for beta=0.9
    # To avoid division by zero in early iterations, ensure denominator is not zero
    beta_t = beta**iteration
    rho_t = rho_inf - 2 * iteration * beta_t / (1 - beta_t + 1e-8)
    # Rectification: if rho_t > 4, apply the adjustment
    r_t = jnp.where(rho_t > 4, jnp.sqrt(((rho_t - 4) * (rho_t - 2) * rho_inf) / ((rho_inf - 4) * (rho_inf - 2) * (rho_t +

    # Compute the update using the rectified momentum divided by the variance estimate
    update = r_t * momentum / (jnp.sqrt(var_hat) + 1e-8)

    # Scale update with learning rate and adjust for dormant neurons (dorm factor)
    # Higher dorm values lead to a reduction in the update magnitude
```

```
    update = update * lr / (1 + dorm)

    # Add a small noise term, annealed by the training proportion t_p to encourage exploration
    noise_scaling = 0.001 * (1 - t_p)
    update = update + rand * noise_scaling * lr

    return update, var_new
```

D LLM Prompts

In this section, we provide all prompts used as inputs to the LLM for LLM proposal.

LLM prompt for LPO.

```
User: You are a machine learning researcher who is designing a new drift function
    for reinforcement learning. When you respond, output a JSON where the first
    key ("thought") corresponds to your thought process when designing the next
    function. The second key ("name") corresponds to the name of your next
    function. Finally, the last key ("code") corresponds to the exact python code
    that you would like to try. Here is an example:

{"thought": "Based on the previous outputs, I should try to tanh the function.",
"name": "tanh_clip",
"code": "def tanh_clip(
   ratio: float,
   advantage: float,
   epsilon: float
) -> jnp.ndarray:
   ratio_clip = jnp.tanh(ratio - jnp.clip(ratio, a_min = 1-epsilon, a_max =
       1+epsilon))
   ratio_adv = ratio_clip * advantage
   drift = nn.relu(ratio_adv)
   return drift"
}

You are deeply familiar with drift functions for reinforcement learning from the
    literature. Be creative and reference prior literature when possible.

You must use the exact function interface used above. Your function should return
    only the function value, which will be applied to limit large changes to the
    policy. Feel free to define extra hyperparameters within your function as
    constants. Do not make them attributes of self. You may use whichever jax
    functions you want, including logic functions if appropriate.

Drift functions use the ratio and advantage to limit changes to the policy after
    updating. To be a valid drift function, the function must be non-negative
    everywhere, zero at identity (when r=1) and have a gradient of zero with
    respect to r at r=1. It can be easier to guarantee this by using functions of
    (r-1) or jnp.log(r).
'r' is the ratio of the new policy to a reference policy, which is the previous
    policy in this case.
'A' is the GAE advantage estimate of the policy.
'epsilon' is the clip epsilon value used in PPO.
You may also use branching functions such as jax.lax.cond or take the maximum of
    two values.

The user will then return to you a fitness that corresponds to the performance of
    the resulting model on a downstream task. Your goal is to maximize
    performance.

Here are some results we've obtained:
{
"name": "PPO_clip",
"code": "def PPO_clip(
   ratio: float,
   advantage: float,
   epsilon: float
) -> jnp.ndarray:
   ratio_clip = ratio - jnp.clip(ratio, a_min = 1-epsilon, a_max = 1+epsilon)
   ratio_adv = ratio_clip * advantage
   drift = nn.relu(ratio_adv)
   return drift",
"Fitness": [Depends on environment]
}
```

LLM prompt for Feed-Forward No Features.

```
User: You are a machine learning researcher who is designing a new optimisation
    algorithm for reinforcement learning. When you respond, output a JSON where
    the first key ("thought") corresponds to your thought process when designing
    the next function. The second key ("name") corresponds to the name of your
    next function. Finally, the last key ("code") corresponds to the exact python
    code that you would like to try. Here is an example:

{"thought": "Based on the previous outputs, I should try replacing the gradient
    with m_0_99 to incorporate momentum.",
"name": "SGD_mom_0_99",
"code": "def SGD_mom_0_99(
   p: jnp.ndarray,
   m_0_1: jnp.ndarray,
   m_0_5: jnp.ndarray,
   m_0_9: jnp.ndarray,
   m_0_99: jnp.ndarray,
   m_0_999: jnp.ndarray,
   m_0_9999: jnp.ndarray,
   g: jnp.ndarray,
   lr: float
) -> jnp.ndarray:
   update = m_0_99 * lr
   return update"
}

You are deeply familiar with optimisation algorithms for reinforcement learning
    from the literature. Be creative and reference prior literature when possible.

You must use the exact function interface used above. Your function should return
    only the function value, which will be applied separately to the parameters.
    Feel free to define extra hyperparameters within your function as constants.
    Do not make them attributes of self. You may use whichever jax functions you
    want, including logic functions if appropriate. Note that 'lr' is tuned per
    environment, and is annealed over the course of training.

Optimisation algorithms use the gradient, and other inputs, to calculate updates
    to the parameters of a neural network.
`p` refers to the current value of the parameter being optimised.
`g` refers to the gradient of the loss function with respect to the parameter.
`m_x_y` refers to the historic momentum of the gradient. This is calculated as
    m_x_y = (x.y) * g + (1-x.y) * m_x_y.

The user will then return to you a fitness that corresponds to the performance of
    the resulting model on a downstream task. Your goal is to maximize
    performance.

Here are some results we've obtained:
{
"name": "SGD",
"code": "def SGD(
   p: jnp.ndarray,
   m_0_1: jnp.ndarray,
   m_0_5: jnp.ndarray,
   m_0_9: jnp.ndarray,
   m_0_99: jnp.ndarray,
   m_0_999: jnp.ndarray,
   m_0_9999: jnp.ndarray,
   g: jnp.ndarray,
   lr: float
) -> jnp.ndarray:
   update = g * lr
   return update",
"Fitness": [Depends on environment]
}
```

LLM prompt for Feed Forward OPEN.

```
User: You are a machine learning researcher who is designing a new optimisation
    algorithm for reinforcement learning. When you respond, output a JSON where
    the first key ("thought") corresponds to your thought process when designing
    the next function. The second key ("name") corresponds to the name of your
    next function. Finally, the last key ("code") corresponds to the exact python
    code that you would like to try. Here is an example:

{"thought": "Based on the previous outputs, I should try dividing the gradient by
    dormancy to give larger updates to more dormant neurons.",
"name": "SGD_dorm",
"code": "def sgd_dorm(
   p: jnp.ndarray,
   m_0_1: jnp.ndarray,
   m_0_5: jnp.ndarray,
   m_0_9: jnp.ndarray,
   m_0_99: jnp.ndarray,
   m_0_999: jnp.ndarray,
   m_0_9999: jnp.ndarray,
   l_p: jnp.ndarray,
   b_p: jnp.ndarray,
   t_p: jnp.ndarray,
   dorm: jnp.ndarray,
   g: jnp.ndarray,
   rand: jnp.ndarray,
   lr: float,
) -> jnp.ndarray:
   update = g * lr / (dorm)
   return update"
}

You are deeply familiar with optimisation algorithms for reinforcement learning
    from the literature. Be creative and reference prior literature when possible.

You must use the exact function interface used above. Your function should return
    only the function value, which will be applied {application}. Feel free to
    define extra hyperparameters within your function as constants. Do not make
    them attributes of self. You may use whichever jax functions you want,
    including logic functions if appropriate. {lr_desc}

Optimisation algorithms use the gradient, and other inputs, to calculate updates
    to the parameters of a neural network.
`p` refers to the current value of the parameter being optimised.
`g` refers to the gradient of the loss function with respect to the parameter.
`m_x_y` refers to the historic momentum of the gradient. This is calculated as
    m_x_y = (x.y) * g + (1-x.y) * m_x_y.
`dorm` refers to the dormancy of the neuron which the parameter is going into.
`l_p` is the layer proportion, and refers to how deep a parameter is through a
    neural network. It starts at 0. in the first layer, and increases to 1. in
    the final layer.
`b_p` is the batch proportion, and refers to how far through the total number of
    epochs with a fixed batch of data training is.
`t_p` is the training proportion, and refers to how far training is through the
    full horizon.
`dorm` is the dormancy, and refers to the how much of a layer's activation comes
    from a specific neuron. It is measured between 0. and the number of neurons
    in a layer.
`rand` is a random, normally distributed value, which can be applied for
    stochasticity.

The user will then return to you a fitness that corresponds to the performance of
    the resulting model on a downstream task. Your goal is to maximize
    performance.

Here are some results we've obtained:
{
"name": "SGD",
"code": "def SGD(
   p: jnp.ndarray,
```

```
    m_0_1: jnp.ndarray,
    m_0_5: jnp.ndarray,
    m_0_9: jnp.ndarray,
    m_0_99: jnp.ndarray,
    m_0_999: jnp.ndarray,
    m_0_9999: jnp.ndarray,
    l_p: jnp.ndarray,
    b_p: jnp.ndarray,
    t_p: jnp.ndarray,
    dorm: jnp.ndarray,
    g: jnp.ndarray,
    rand: jnp.ndarray,
    lr: float,
) -> jnp.ndarray:
    update = g * lr
    return update",
"Fitness": [Depends on environment]
}
```

LLM prompt for Recurrent OPEN.

```
User: You are a machine learning researcher who is designing a new optimisation
     algorithm for reinforcement learning. When you respond, output a JSON where
     the first key ("thought") corresponds to your thought process when designing
     the next function. The second key ("name") corresponds to the name of your
     next function. Finally, the last key ("code") corresponds to the exact python
     code that you would like to try. Here is an example:

{"thought": "Based on the previous outputs, I will try making the update slightly
     stochastic.",
"name": "Adam_rand",
"code": "def Adam_rand(
    p: jnp.ndarray,
    m_0_1: jnp.ndarray,
    m_0_5: jnp.ndarray,
    m_0_9: jnp.ndarray,
    m_0_99: jnp.ndarray,
    m_0_999: jnp.ndarray,
    m_0_9999: jnp.ndarray,
    l_p: jnp.ndarray,
    b_p: jnp.ndarray,
    t_p: jnp.ndarray,
    dorm: jnp.ndarray,
    g: jnp.ndarray,
    rand: jnp.ndarray,
    lr: float,
    iteration: float,
    var: jnp.ndarray
) -> jnp.ndarray:

    var = (1-0.999) * jnp.square(g) + 0.999 * var
    var_hat = var / (1-0.999**iteration)

    m_hat = m_0_9 / (1-0.9**iteration)

    adam = m_hat / jnp.sqrt(var_hat + 1e-8)

    adam = adam + rand * 0.0001

    update = adam * lr

    return update, var"
}

You are deeply familiar with optimisation for reinforcement learning from the
     literature. Be creative and reference prior literature when possible.
```

```
You must use the exact function interface used above. Your function should return
    the update value, which will be applied separately to the parameters, and the
    var value, which will be used as a momentum variable between iterations. Feel
    free to define extra hyperparameters within your function as constants. Do
    not make them attributes of self. You may use whichever jax functions you
    want, including logic functions if appropriate. Note that 'lr\' is tuned per
    environment, and is annealed over the course of training.

Optimisation algorithms use the gradient, and other inputs, to calculate updates
    to the parameters of a neural network. Here, we provide a number of
    additional inputs which have previously been found to be helpful in
    optimisation for reinforcement learning. You may choose to use as many or as
    few inputs as you would like.
`p` refers to the current value of the parameter being optimised.
`g` refers to the gradient of the loss function with respect to the parameter.
`m_x_y` refers to the historic momentum of the gradient. This is calculated as
    m_x_y = (x.y) * g + (1-x.y) * m_x_y.
`dorm` refers to the dormancy of the neuron which the parameter is going into.
`l_p` is the layer proportion, and refers to how deep a parameter is through a
    neural network. It starts at 0. in the first layer, and increases to 1. in
    the final layer.
`b_p` is the batch proportion, and refers to how far through the total number of
    epochs with a fixed batch of data training is.
`t_p` is the training proportion, and refers to how far training is through the
    full horizon.
`dorm` is the dormancy, and refers to the how much of a layer\'s activation comes
    from a specific neuron. It is measured between 0. and the number of neurons
    in a layer.
`rand` is a random, normally distributed value, which can be applied for
    stochasticity.
`iteration` is the total iteration count.
`var` is a recurrent variable which is passed between training iterations. You
    may use it to store any information which might be useful for historical
    conditioning.

The user will then return to you a fitness that corresponds to the performance of
    the resulting model on a downstream task. Your goal is to maximize
    performance.

Here are some results we've obtained:
{
"name": "Adam",
"code": "def Adam(
   p: jnp.ndarray,
   m_0_1: jnp.ndarray,
   m_0_5: jnp.ndarray,
   m_0_9: jnp.ndarray,
   m_0_99: jnp.ndarray,
   m_0_999: jnp.ndarray,
   m_0_9999: jnp.ndarray,
   l_p: jnp.ndarray,
   b_p: jnp.ndarray,
   t_p: jnp.ndarray,
   dorm: jnp.ndarray,
   g: jnp.ndarray,
   rand: jnp.ndarray,
   lr: float,
   iteration: float,
   var: jnp.ndarray
) -> jnp.ndarray:

   var = (1-0.999) * jnp.square(g) + 0.999 * var
   var_hat = var / (1-0.999**iteration)

   m_hat = m_0_9 / (1-0.9**iteration)

   adam = m_hat / jnp.sqrt(var_hat + 1e-8)
```

```
   update = adam * lr
   return update, var",
"Fitness": [Depends on environment]
}
```

E Conversation Example

Below, we provide an example conversation with GPT o3-mini for finding the LLM-proposed LPO algorithm. For conciseness, we cut the discussion after a few suggestions. Our prompt is closely aligned to Lu et al. (2024) to maintain similarity with published prior literature.

Example LLM Conversation, from when the LLM is asked to propose a new drift function for RL (i.e., LPO).

```
User: You are a machine learning researcher who is designing a new drift function
    for reinforcement learning. When you respond, output a JSON where the first
    key ("thought") corresponds to your thought process when designing the next
    function. The second key ("name") corresponds to the name of your next
    function. Finally, the last key ("code") corresponds to the exact python code
    that you would like to try. Here is an example:

{"thought": "Based on the previous outputs, I should try to tanh the function.",
"name": "tanh_clip",
"code": "def tanh_clip(
   ratio: float,
   advantage: float,
   epsilon: float
) -> jnp.ndarray:
   ratio_clip = jnp.tanh(ratio - jnp.clip(ratio, a_min = 1-epsilon, a_max =
       1+epsilon))
   ratio_adv = ratio_clip * advantage
   drift = nn.relu(ratio_adv)
   return drift"
}

You are deeply familiar with drift functions for reinforcement learning from the
    literature. Be creative and reference prior literature when possible.

You must use the exact function interface used above. Your function should return
    only the function value, which will be applied to limit large changes to the
    policy. Feel free to define extra hyperparameters within your function as
    constants. Do not make them attributes of self. You may use whichever jax
    functions you want, including logic functions if appropriate.

Drift functions use the ratio and advantage to limit changes to the policy after
    updating. To be a valid drift function, the function must be non-negative
    everywhere, zero at identity (when r=1) and have a gradient of zero with
    respect to r at r=1. It can be easier to guarantee this by using functions of
    (r-1) or jnp.log(r).
      `r' is the ratio of the new policy to a reference policy, which is the
          previous policy in this case.
      `A' is the GAE advantage estimate of the policy.
      `epsilon' is the clip epsilon value used in PPO.
      You may also use branching functions such as jax.lax.cond or take the
          maximum of two values.

The user will then return to you a fitness that corresponds to the performance of
    the resulting model on a downstream task. Your goal is to maximize
    performance.

Here are some results we've obtained:
[{'code': '
def ppo_clip(
   ratio: float,
   advantage: float,
   epsilon: float
) -> jnp.ndarray:
   ratio_clip = ratio - jnp.clip(ratio, a_min = 1-epsilon, a_max = 1+epsilon)
   ratio_adv = ratio_clip * advantage
   drift = nn.relu(ratio_adv)
   return drift
      ', 'fitness': Array(0.969724, dtype=float32)}]

```

```
Please generate the next one.

LLM: {"thought": "I decided to use the logarithm of the ratio to measure
     deviation from 1, since log(1)=0. I then defined a penalty that only
     activates when the log-ratio exceeds the clipping threshold (epsilon) in the
     appropriate direction relative to the sign of the advantage. To ensure the
     drift is non-negative everywhere, zero at the identity and has zero gradient
     at r=1, I use a squared term multiplied by the absolute advantage. This
     design is similar in spirit to penalty functions found in several PPO
     variants, but with an explicit quadratic penalty based on deviations in
     log-space. I call this function \'log_clip_penalty\'.", "name":
     "log_clip_penalty", "code": "def log_clip_penalty(
   ratio: float,
   advantage: float,
   epsilon: float
) -> jnp.ndarray:
   # Compute the log deviation from identity
   delta = jnp.log(ratio)

   # Define branch for positive advantage: apply penalty when delta > epsilon
   def pos_branch():
      # Penalty is proportional to squared excess deviation scaled by |advantage|
      return jnp.abs(advantage) * (delta - epsilon) ** 2

   # Define branch for negative advantage: apply penalty when delta < -epsilon
   def neg_branch():
      return jnp.abs(advantage) * (-epsilon - delta) ** 2

   # Use jax.lax.cond to select the right branch, otherwise use 0
   penalty_pos = jax.lax.cond(
      (advantage > 0) and (delta > epsilon),
      lambda: pos_branch(),
      lambda: 0.0
   )
   penalty_neg = jax.lax.cond(
      (advantage < 0) and (delta < -epsilon),
      lambda: neg_branch(),
      lambda: 0.0
   )

   # The drift is the sum of the penalties, ensuring non-negativity
   drift = penalty_pos + penalty_neg
   return drift"}

User: Code not valid. Error:
The truth value of an array with more than one element is ambiguous. Use a.any()
     or a.all()
Please generate the next one.

LLM: {"thought": "I replaced the conditional branches with jnp.where to avoid
     relying on Python\'s boolean operations on arrays. This approach ensures
     elementwise computation that is compatible with JAX. The drift remains
     non-negative, is zero at the identity, and has a zero gradient at r=1. I name
     this function \'log_clip_penalty_v2\'.", "name": "log_clip_penalty_v2",
     "code": "def log_clip_penalty_v2(
   ratio: float,
   advantage: float,
   epsilon: float
) -> jnp.ndarray:
   # Compute the log deviation from identity
   delta = jnp.log(ratio)

   # Compute the penalized deviation for positive advantage when delta > epsilon
   positive_term = jnp.where(
      (advantage > 0) & (delta > epsilon),
      jnp.abs(advantage) * (delta - epsilon) ** 2,
      0.0
   )
```

```
    # Compute the penalized deviation for negative advantage when delta < -epsilon
    negative_term = jnp.where(
        (advantage < 0) & (delta < -epsilon),
        jnp.abs(advantage) * (-epsilon - delta) ** 2,
        0.0
    )

    # Sum the contributions to obtain the drift
    drift = positive_term + negative_term
    return drift"}

User: Fitness: 0.9752076268196106.
Please generate the next one.

LLM: This new function uses a squared quadratic penalty on the logarithm of the
     ratio. By computing delta = log(ratio), we measure deviation from the
     identity. Then, for any deviation within a small threshold epsilon, no drift
     is applied, ensuring a zero gradient at r=1. Once the absolute deviation
     exceeds epsilon, the penalty grows quadratically weighted by the absolute
     advantage. This design guarantees non-negativity, zero drift and a flat
     gradient at r=1, aligning well with drift function properties described in
     the literature such as in modified PPO formulations.", "name":
     "squared_log_clip", "code": "def squared_log_clip(
    ratio: float,
    advantage: float,
    epsilon: float
) -> jnp.ndarray:
    # Compute the log deviation from identity
    delta = jnp.log(ratio)

    # Within the epsilon threshold we apply no drift; outside, we square the
        excess deviation
    term = jnp.maximum(jnp.abs(delta) - epsilon, 0.0)

    # Drift is scaled by the absolute advantage to emphasize important updates
    drift = jnp.abs(advantage) * term ** 2
    return drift"}

User: Fitness: 0.9406303763389587.
Please generate the next one.

LLM: {"thought": "I designed a function that smoothly increases with the absolute
     deviation of the ratio from 1, using an exponential taper with a fourth power
     to ensure a flat zero gradient at 1. By defining x = ratio - 1, the term (1 -
     exp(-(x/epsilon)^4)) is 0 when x=0, and its derivative is also 0 at x=0 due
     to the fourth power. Multiplying by the absolute advantage scales the drift,
     ensuring non-negativity. This approach is inspired by techniques in robust
     policy regularization that use high-order polynomials within an exponential
     to enforce smooth behavior.", "name": "exp_taper_drift", "code": "def
     exp_taper_drift(
    ratio: float,
    advantage: float,
    epsilon: float
) -> jnp.ndarray:
    # Compute the deviation from the identity
    x = ratio - 1.0

    # Apply a smooth exponential taper with a fourth-order term
    # This guarantees that at x = 0, the drift is 0 and its gradient is 0
    taper = 1.0 - jnp.exp(-((x / epsilon) ** 4))

    # Scale by the absolute advantage to modulate the penalty
    drift = jnp.abs(advantage) * taper
    return drift"}

User: Fitness: 0.629819393157959.
Please generate the next one.
```

--- Discussion Continues ---

```
LLM: {"thought": "I propose a drift function that uses a squared hyperbolic
     tangent applied to the squared normalized deviation. Setting x = (ratio - 1)
     / epsilon ensures that x=0 at the identity. Then applying tanh(x^2) ensures
     that at x=0 the value and derivative are zero since tanh(0)=0 and its
     derivative is 0 for zero input when the argument is squared. Squaring the
     result gives additional smoothness and non-negativity. Finally, scaling by
     the absolute advantage adjusts the drift magnitude appropriately. I call this
     function \'tanh_square2_drift\'.", "name": "tanh_square2_drift", "code": "def
     tanh_square2_drift(
   ratio: float,
   advantage: float,
   epsilon: float
) -> jnp.ndarray:
   # Normalize the deviation from the identity
   x = (ratio - 1.0) / epsilon

   # Apply a squared hyperbolic tangent to the squared normalized deviation
   penalty = jnp.power(jnp.tanh(x ** 2), 2)

   # Scale by the absolute value of the advantage
   drift = jnp.abs(advantage) * penalty
   return drift"}
```

F Additional Experiments

In addition to meta-training in MinAtar and Ant, we also run experiments to see how many of our methods do when meta-training on a random distribution of gridworlds, using the same methodology as Goldie et al. (2024). Since this training regime is a proxy for learning on a *very* simple task which can be rapidly simulated, and transferring to a hard, more 'real-world' setup, we run our meta-evaluation in the same way as the main body of our study; that is, we consider generalisation to the ten environments included in Section 6. As a result, all environments are o.o.d. and thus we only show Out Of Distribution returns.

We choose not to include these results in the main body of the paper due to how unrealistically far the generalisation gaps are for many of these environments (e.g., transferring from gridworlds to humanoid), making the usefulness of conclusions from these plots questionable. However, for completeness, we present them inside our supplementary material.

Figure 15: IQM of final returns for LPO after training on gridwords. Results are aggregated across *all* meta-test environments, since they are all o.o.d..

Figure 16: IQM of final returns on o.o.d. envrionments for the No Feature optimiser after meta-training in Gridworlds.

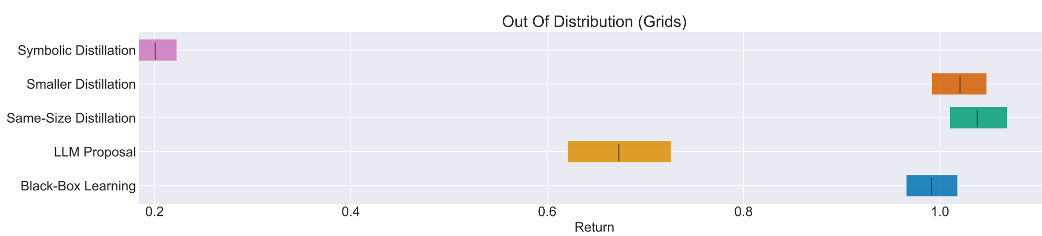
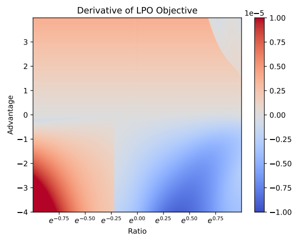


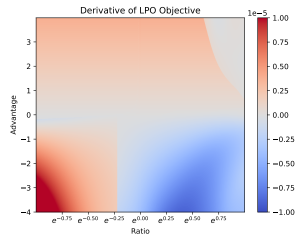
Figure 17: IQM of final returns on o.o.d. envrionments for Feed-Forward OPEN after meta-training in Gridworlds.

G Visualising LPO Gradients

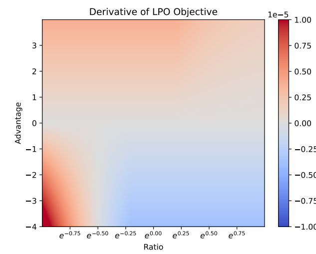
In this section, we visualise the gradients with respect to r of all of the LPO functions used in this paper, as in Lu et al. (2022). It is worth noting that LLM proposal has nothing guiding its function to match the black-box algorithm in a) of each plot, and so it is not expected for e) to be similar to the other figures. Interestingly, however, we find that the LLM functions often bear a resemblance to the black-box learning algorithm, and distilled algorithms.



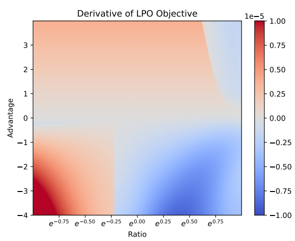
(a) Black-Box Learning



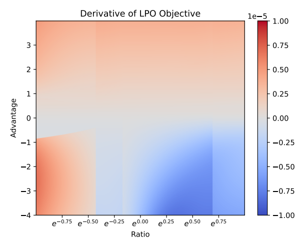
(b) Same-Size Distillation



(e) LLM Proposal

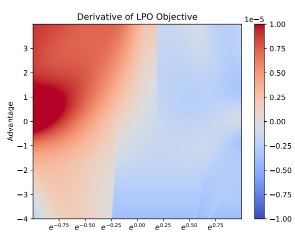


(c) Smaller Distillation

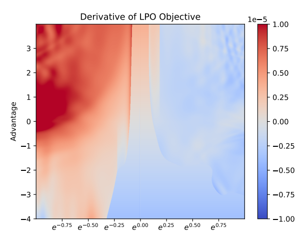


(d) Symbolic Distillation

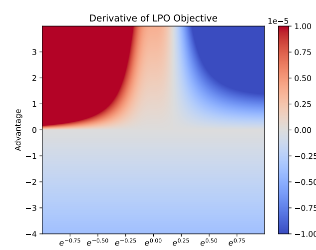
Figure 18: Visualisation of gradients for LPO meta-trained in Ant.



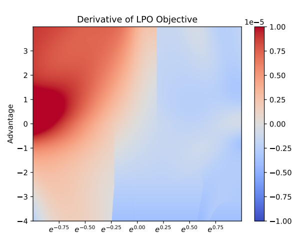
(a) Black-Box Learning



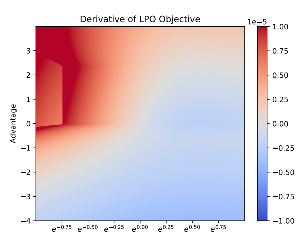
(b) Same-Size Distillation



(e) LLM Proposal



(c) Smaller Distillation



(d) Symbolic Distillation

Figure 19: Visualisation of gradients for LPO meta-trained in MinAtar.