Discovering Reinforcement Learning Algorithms

Junhyuk Oh **Matteo Hessel** **Wojciech M. Czarnecki** **Zhongwen Xu**

Hado van Hasselt **Satinder Singh** **David Silver**

DeepMind

Abstract

Reinforcement learning (RL) algorithms update an agent's parameters according to one of several possible rules, discovered manually through years of research. Automating the discovery of update rules from data could lead to more efficient algorithms, or algorithms that are better adapted to specific environments. Although there have been prior attempts at addressing this significant scientific challenge, it remains an open question whether it is feasible to discover alternatives to fundamental concepts of RL such as value functions and temporal-difference learning. This paper introduces a new meta-learning approach that discovers an entire update rule which includes both 'what to predict' (e.g. value functions) and 'how to learn from it' (e.g. bootstrapping) by interacting with a set of environments. The output of this method is an RL algorithm that we call Learned Policy Gradient (LPG). Empirical results show that our method discovers its own alternative to the concept of value functions. Furthermore it discovers a bootstrapping mechanism to maintain and use its predictions. Surprisingly, when trained solely on toy environments, LPG generalises effectively to complex Atari games and achieves non-trivial performance. This shows the potential to discover general RL algorithms from data.

1 Introduction

Reinforcement learning (RL) has a clear objective: to maximise expected cumulative rewards (or average rewards), which is simple, yet general enough to capture many aspects of intelligence. Even though the objective of RL is simple, developing efficient algorithms to optimise such objective typically involves a tremendous research effort, from building theories to empirical investigations. An appealing alternative approach is to automatically discover RL algorithms from data generated by interaction with a set of environments, which can be formulated as a meta-learning problem. Recent work has shown that it is possible to meta-learn a policy update rule when given a value function, and that the resulting update rule can generalise to similar or unseen tasks (see Table 1).

However, it remains an open question whether it is feasible to discover fundamental concepts of RL entirely from scratch. In particular, a defining aspect of RL algorithms is their ability to learn and utilise value functions. Discovering concepts such as value functions requires an understanding of both 'what to predict' and 'how to make use of the prediction'. This is particularly challenging to discover from data because predictions only have an indirect effect on the policy over the course of multiple updates. We hypothesise that a method capable of discovering value functions for itself may also discover other useful concepts, potentially opening up entirely new approaches to RL.

Motivated by the aforementioned open questions, this paper takes a step towards discovering general-purpose RL algorithms. We introduce a meta-learning framework that jointly discovers both 'what the agent should predict' and 'how to use predictions for policy improvement' from data generated by interacting with a distribution of environments. Our architecture, Learned Policy Gradient (LPG),

Preprint. Under review.

Table 1: Methods for discovering RL algorithms.

Algorithm	Discovery	Method	Generality	Train	Test
RL^2 [7, 34]	N/A	∇	Domain-specific	3D maze	Similar 3D maze
EPG [15]	$\hat{\pi}$	ES	Domain-specific	MuJoCo	Similar MuJoCo
ML^3 [5]	$\hat{\pi}$	$\nabla\nabla$	Domain-specific	MuJoCo	Similar MuJoCo
MetaGenRL [17]	$\hat{\pi}$	$\nabla\nabla$	General	MuJoCo	Unseen MuJoCo
LPG	$\hat{\pi}, \hat{y}$	$\nabla\nabla$	General	Toy	Unseen Atari

$\hat{\pi}$: policy update rule, $\hat{y}$: prediction update rule (i.e., semantics of agent's prediction).
∇: gradient descent, $\nabla\nabla$: meta-gradient descent, ES: evolutionary strategy.

does not enforce any semantics on the agent's vector-valued outputs but instead allows the update rule (i.e., the meta-learner) to decide what this vector should be predicting. We then propose a meta-learning framework to discover such update rule from multiple learning agents, each of which interacts with a different environment.

Experimental results show that our algorithm can discover useful functions, and use those functions effectively to update the agents policy. Furthermore, empirical analysis shows that the discovered functions converge towards an encoding of a notion of value function, and furthermore maintain this value function via a form of bootstrapping. We also evaluated the ability of the discovered RL algorithm to generalise to new environments. Surprisingly, even though the update rule was discovered solely from interactions with a very small set of toy environments, it was able to generalise to a number of complex Atari games [2], as shown in Figure 9. To our knowledge, this is the first to show that it is possible to discover an entire update rule, and that the update rule discovered from toy domains can be competitive with human-designed algorithms on a challenging benchmark.

2 Related Work

Early Work on Learning to Learn The idea of learning to learn has been discussed for a long time with various formulations such as improving genetic programming [26], learning a neural network update rule [3], learning rate adaptations [29], self-weight-modifying RNNs [27], and transfer of domain-invariant knowledge [31]. Such work showed that it is possible to learn not only to optimise fixed objectives, but also to improve the way to optimise at a meta-level.

Learning to Learn for Few-Shot Task Adaptation Learning to learn has received much attention in the context of few-shot learning [25, 33]. MAML [9, 10] allows to meta-learn initial parameters by backpropagating through the parameter updates. RL^2 [7, 34] formulates learning itself as an RL problem by unrolling LSTMs [14] across the agent's entire lifetime. Other approaches include simple approximation [23], RNNs with Hebbian learning [19, 20], and gradient preconditioning [11]. All these do not clearly separate between agent and algorithm, thus, the resulting meta-learned algorithms are specific to a single agent architecture by definition of the problem.

Learning to Learn for Single Task Online Adaptation A different corpus of work focuses on learning to learn a single task within a single lifetime. Xu et al. [37] introduced the meta-gradient RL approach; this uses backpropagation through the agent's updates, to calculate the gradient with respect to the meta-parameters of the update. This approach has been applied to meta-learn various forms of algorithmic components such as the discount factor [37], intrinsic rewards [40], auxiliary tasks [32], returns [35], auxiliary policy updates [41], off-policy corrections [38], and update target [36]. In contrast, our work has an orthogonal goal: to discover general algorithms that are effective for a broader class of agents and environments instead of being adaptive to a particular environment.

Discovering Reinforcement Learning Algorithms There have been a few attempts to meta-learn general algorithms from interactions with a distribution of environments (see Table 1 for comparison). EPG [15] uses an evolutionary strategy to find a policy update rule. Zheng et al. [39] showed that general knowledge for exploration can be meta-learned in the form of reward function. ML^3 [5] meta-learns a loss function using meta-gradients. However, the prior work is limited to domain-specific algorithms, in that they can generalise only up to similar tasks within the same domain. Most recently, MetaGenRL [17] was proposed to meta-learn a domain-invariant policy update rule, capable of generalizing from a few MuJoCo environments to other MuJoCo environments. However, no prior

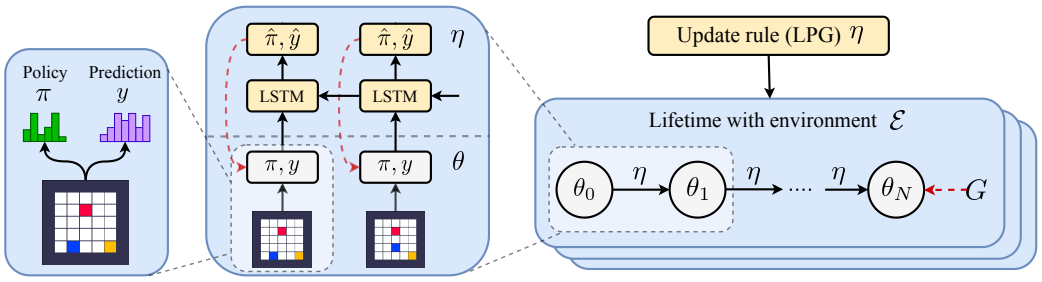


Figure 1: Meta-training of learned policy gradient (LPG). (Left) The agent parameterised by θ produces action-probabilities π and a prediction vector y for a state. (Middle) The update rule (LPG) parameterised by η takes the agent outputs as input and unrolls an LSTM backward to produce targets for the agent outputs ($\hat{\pi}$, $\hat{y}$). (Right) The update rule η is meta-learned from multiple lifetimes, in each of which a distinct agent interacts with an environment sampled from a distribution, and updates its parameters θ using the shared update rule. The meta-gradient is calculated to maximise the return after every $K < N$ parameter updates by sliding window, averaged over all parallel lifetimes.

work has attempted to discover the *full* update rule; instead they all relied on value functions, arguably the most fundamental building block of RL, for bootstrapping. In contrast, our LPG meta-learns its own mechanism for bootstrapping. Additionally, this paper is the first to show that a radical generalisation from toy environments to a challenging benchmark is possible.

3 Meta-Learning Framework for Learned Policy Gradient

The goal of the proposed meta-learning framework is to find the optimal update rule, parameterised by η, from a distribution of environments $p(\mathcal{E})$ and initial agent parameters $p(\theta_0)$:

$$\eta^* = \arg\max_{\eta} \mathbb{E}_{\mathcal{E}\sim p(\mathcal{E})} \mathbb{E}_{\theta_0 \sim p(\theta_0)} \left[G \right], \tag{1}$$

where $G = \mathbb{E}_{\pi_{\theta_N}}[\sum_t^{\infty} \gamma^t r_t]$ is the expected return at the end of the lifetime. Intuitively, the objective aims to find an update rule η such that when it is used to update the agent's parameters until the end of its lifetime ($\theta_0 \rightarrow \cdots \rightarrow \theta_N$), the agent maximises the expected return in the given environment. The resulting update rule is called Learned Policy Gradient (LPG). The overview of meta-training process is summarised in Figure 1 and Algorithm 1.

3.1 LPG Architecture

As illustrated in Figure 1, LPG is an update rule parameterised by meta-parameters η which requires an agent to produce a policy $\pi_\theta(a|s)$ and a m-dimensional categorical prediction vector $y_\theta(s) \in [0,1]^m$. The LPG is a backward LSTM [14] network that produces as output how to update the policy and the prediction vector $\hat{\pi} \in \mathbb{R}, \hat{y} \in [0,1]^m$ from the trajectory of the agent. More specifically, it takes $x_t = [r_t, d_t, \gamma, \pi(a_t|s_t), y_\theta(s_t), y_\theta(s_{t+1})]$ at each time-step t, where r_t is a reward, d_t is a binary value indicating episode-termination, and γ is a discount factor. By construction, LPG is invariant to observation space and action space, as it does not take them as input. Instead, it only takes the probability of the chosen action $\pi(a|s)$. This structure allows the LPG architecture to be applicable to entirely different environments while preventing overfitting.

3.2 Agent Update (θ)

Agent parameters are updated by performing gradient ascent in the direction of:

$$\Delta\theta \propto \mathbb{E}_{\pi_\theta} \left[\nabla_\theta \log \pi_\theta(a|s)\hat{\pi} - \alpha_y \nabla_\theta D_{\mathrm{KL}}(y_\theta(s) \| \hat{y}) \right], \tag{2}$$

where $\hat{\pi}$ and $\hat{y}$ are the output of LPG. $D_{\mathrm{KL}}(P\|Q) = \sum_x P(x) \log \frac{P(x)}{Q(x)}$ is the Kullback–Leibler divergence. α_y is a coefficient for the prediction update respectively. At a high-level, $\hat{\pi}$ specifies how the action-probability should be adjusted, and has a direct effect on the agent's behaviour. $\hat{y}$ specifies a target categorical distribution that the agent should predict for a given state, and does not have an effect on the policy until the LPG discovers useful semantics (e.g., value function) of it and uses y to indirectly change the policy by bootstrapping, which makes the discovery problem challenging.

Note that the proposed framework is not restricted to this particular form of agent update and architecture (e.g., categorical prediction with KL-divergence). We explore this specific form partly

Algorithm 1 Meta-Training of Learned Policy Gradient

Input: $p(\mathcal{E})$: Environment distribution, $p(\theta_0)$: Initial agent parameter distribution
Initialise meta-parameters η and hyperparameter sampling distribution $p(\alpha|\mathcal{E})$
Sample batch of environment-agent-hyperparameters $\{\mathcal{E} \sim p(\mathcal{E}), \theta \sim p(\theta_0), \alpha \sim p(\alpha|\mathcal{E})\}_i$
repeat
  for all lifetimes $\{\mathcal{E}, \theta, \alpha\}_i$ **do**
    Update parameters θ using η and α for K times using Eq. (2)
    Compute meta-gradient using Eq. (4)
    if lifetime ended **then**
      Update hyperparameter sampling distribution $p(\alpha|\mathcal{E})$
      Reset lifetime $\mathcal{E} \sim p(\mathcal{E}), \theta \sim p(\theta_0), \alpha \sim p(\alpha|\mathcal{E})$
    end if
  end for
  Update meta-parameters η using the meta-gradients averaged over all lifetimes.
until η converges

inspired by the success of Distributional RL [1, 6]. However, we do not enforce any semantics on y but allow the LPG to discover the semantics of y from data.

3.3 LPG Update (η)

LPG is meta-trained by taking into account how much it improves the performances of a population of agents interacting with different kinds of environments. Specifically, the meta-gradients are calculated by applying policy gradient to the objective in Eq. (1) as follows:

$$\Delta\eta \propto \mathbb{E}_{\mathcal{E}}\mathbb{E}_{\theta_0}\left[\nabla_\eta \log \pi_{\theta_N}(a|s)G\right] \tag{3}$$

Intuitively, we perform parameter updates for N times using the update rule η from θ_0 to θ_N until the end of the lifetime and estimate policy gradient for the updated parameters θ_N to find the meta-gradient direction that maximises the expected return (G) of θ_N. This requires backpropagation through the agent's update process as in [37, 10]. In practice, due to memory constraints, we consider a smaller sliding window, and perform a truncated backpropagation every $K < N$ parameter updates.

Regularisation We find that the optimisation can be very hard and unstable, mainly because the LPG needs to learn an appropriate semantics of predictions $\hat{y}$, as well as learning to use predictions y properly for bootstrapping without access to the value function. To stabilise training, we propose to add the following regularisers (on the targets $\hat{\pi}$ and $\hat{y}$), resulting in the meta-gradient:

$$\mathbb{E}_{\mathcal{E}}\mathbb{E}_{\theta_0}\left[\nabla_\eta \log \pi_{\theta_N}(a|s)G + \beta_0\nabla_\eta\mathcal{H}(\pi_{\theta_N}) + \beta_1\nabla_\eta\mathcal{H}(y_{\theta_N}) - \beta_2\nabla_\eta\|\hat{\pi}\|_2^2 - \beta_3\nabla_\eta\|\hat{y}\|_2^2\right], \tag{4}$$

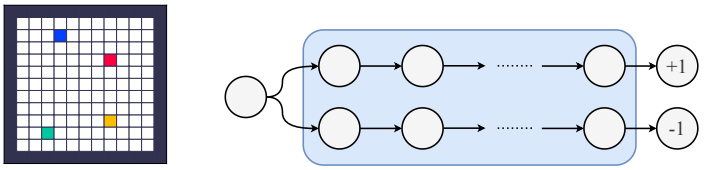
where $\mathcal{H}(\cdot)$ is the entropy, and $\{\beta_i\}$ are meta-hyperparameters for each regularisation term. $\mathcal{H}(y)$ penalises too deterministic predictions, which shares the same motivation with policy entropy regularisation $\mathcal{H}(\pi)$ [21]. These are not applied to the agent but applied to the update rule so that the resulting LPG has such properties. The L2-regularisation for $\hat{\pi}, \hat{y}$ prevents the updates from being too aggressive. We discuss the effect of these regularisers in Section 4.4.

3.4 Balancing Agent Hyperparameters for Stabilisation (α)

While previous approaches [5, 17] used fixed agent hyperparameters (e.g., learning rate) during meta-training, we find it problematic when meta-training across entirely different environments. For example, if the learning rate used for environment A happens to be relatively larger than that for environment B, the optimal scale of $\hat{\pi}$ should be smaller for A and larger for B. Since the update rule η is environment-agnostic, it would get contradicting meta-gradients between two environments, making meta-training unstable. Furthermore, it is impossible to pre-balance hyperparameters due to their dependence on η, which changes during meta-training making the problem of balancing hyperparameters inherently non-stationary. To address this, we modify the objective in Eq. 1 to:

$$\eta^* = \arg\max_\eta \mathbb{E}_{\mathcal{E}\sim p(\mathcal{E})} \max_\alpha \mathbb{E}_{\theta_0\sim p(\Theta)}\left[G\right], \tag{5}$$

where $\alpha = \{\alpha_{\text{lr}}, \alpha_y\}$ are a learning rate and a coefficient for prediction update (see Eq. (2)). This objective seeks the optimal update rule given the optimal hyperparameters for each environment. To



(a) Grid world (b) Delayed chain MDP

Figure 2: Training environments. (a) The agent receives the corresponding rewards by collecting objects. (b) The first action determines the reward at the end of the episode.

optimise this, in practice, we propose to use a bandit $p(\alpha|\mathcal{E})$ that samples hyperparameters for each lifetime and updates the sampling distribution according to the return at the end of each lifetime. By making $p(\alpha|\mathcal{E})$ adapt to each environment, hyperparameters are automatically balanced across environments, which makes meta-gradient less noisy. Note that this is done only during meta-training. During meta-testing on unseen environments, hyperparameters need to be manually selected in the same way that we do for existing RL algorithms. Further details on how this was done in our experiments are described in the supplementary material.

4 Experiment

The experiments are designed to answer the following research questions:

- Can LPG discover a useful semantics of predictions for efficient bootstrapping?
- What are the discovered semantics of predictions?
- How crucial is it to discover the semantics of predictions?
- How crucial are the regularisers and hyperparameter balancing?
- Can LPG generalise from toy environments to complex Atari games?

4.1 Experimental Setup

Training Environments For meta-training of LPG, we introduce three different kinds of toy domains as illustrated Figure 2. *Tabular grid worlds* are grid worlds with fixed object locations. *Random grid worlds* have randomised object locations for each episode. *Delayed chain MDPs* are simple MDPs with delayed rewards. There are 5 variations of environments for each domain with various number of rewarding states and episode lengths. The training environments are designed to captures basic RL challenges such as delayed reward, noisy reward, and long-term credit assignment. Most of the training environments are tabular without involving any function approximators. The details of all environments are described in the supplementary material.

Implementation Details We used a 30-dimensional prediction vector $y \in [0, 1]^{30}$. During meta-training, we updated the agent parameters after every 20 time-steps. Since most of the training episodes span over 20-2000 steps, LPG must discover a long-term semantics for the predictions y to be able to maximise long-term future rewards from partial trajectories. The algorithm is implemented using JAX [4]. More implementation details are described in the supplementary material.

Baselines As discussed in Section 2 and Table 1, most of the prior work does not support generalisation across entirely different environments except MetaGenRL [17]. However, MetaGenRL is designed for continuous control and based on DDPG [28, 18]. Instead, to investigate the importance of discovering prediction semantics, we compare to our own baseline **LPG-V**, a variant of LPG that, like MetaGenRL, only learns a policy update rule ($\hat{\pi}$) given a value function trained by TD(λ) [30] without discovering its own prediction semantics.[1] Additionally, we also compare against an advantage actor-critic (A2C) [21] as a canonical human-discovered algorithm baseline.

4.2 Specialising in Training Environments

We evaluated the LPG on the training environments to see whether LPG has discovered an effective update rule. The result in Figure 3 shows that the LPG outperforms A2C on most of the training

[1]LPG-V does not fully represent MetaGenRL as it includes other advances introduced in this paper.

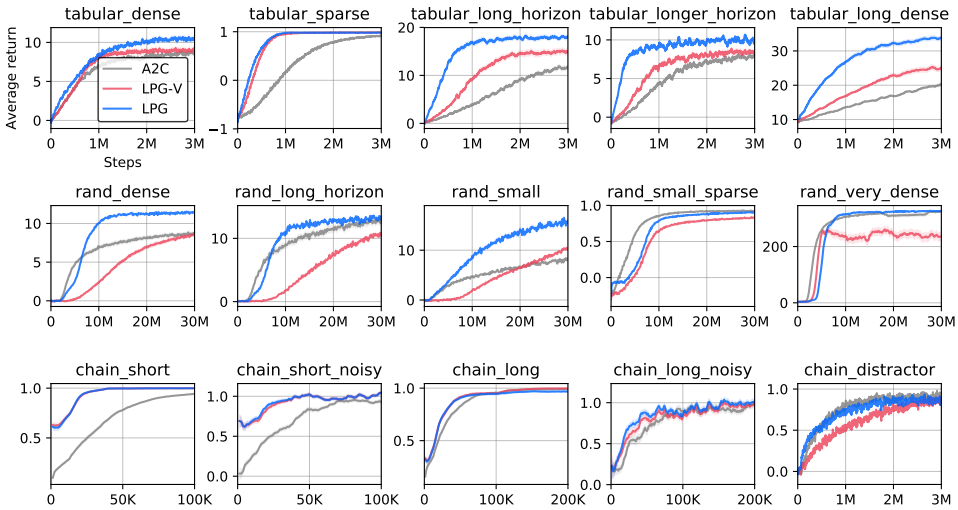
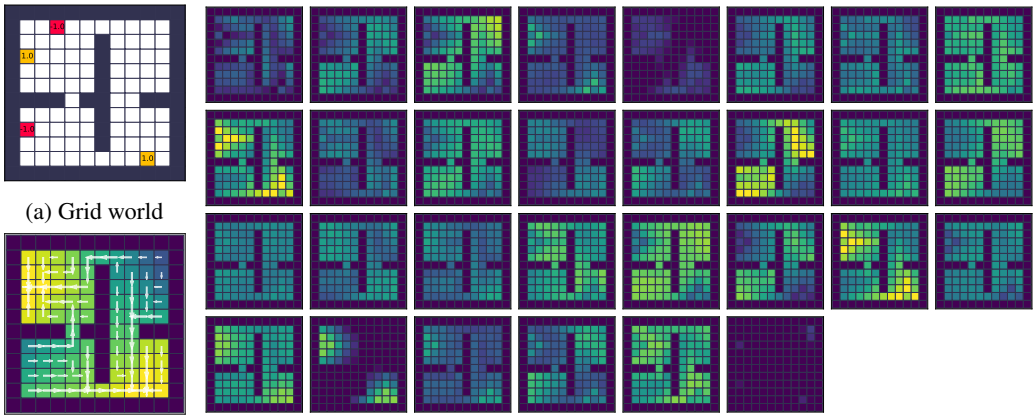


Figure 3: Evaluation on the training environments. Shaded areas show standard errors from 64 random seeds.



(a) Grid world

(b) Policy and value

(c) Predictions (y)

Figure 4: Visualisation of predictions. (a) A grid world with positive goals (yellow) and negative goals (red). (b) A near-optimal policy and its true values. (c) Visualisation of $y \in [0, 1]^{30}$ for the given policy in (b).

environments. This shows that the proposed framework can discover an update rule that outperforms the *outer* algorithm used for discovery (i.e., policy gradient in Eq (4)) on the given environments. In addition, the result suggests that LPG specialises in certain classes of environments, and that LPG can be potentially an even better solution than hand-designed RL algorithms if one is interested in a specific class of RL problems. On the other hand, LPG-V is much worse than LPG while not clearly better than A2C. This shows that discovering the semantics of prediction is the key for the performance, which justifies our approach in contrast to the prior work that only learns a policy update rule while relying on grounded value functions (see Table 1 for comparison).

4.3 Analysis of Learned Policy Gradient

What does the prediction (y) look like? Since the discovered semantics of prediction is the key for the performance, a natural question is what are the discovered concepts and how they work. To answer this, we first visualised the predictions for a given tabular grid world instance and for a fixed policy in Figure 4. Specifically, we updated only y using the LPG while fixing the policy parameters, which is analogous to policy evaluation.[2] The visualisation in Figure 4c shows that some predictions have large values around positive rewarding states, and they are propagated to nearby states similarly to the true values in Figure 4b. This visualisation implicitly shows that the LPG is asking the agent to predict future rewards and use such information for bootstrapping.

[2]To avoid overfitting, we created an unseen grid world task with an unseen action space.

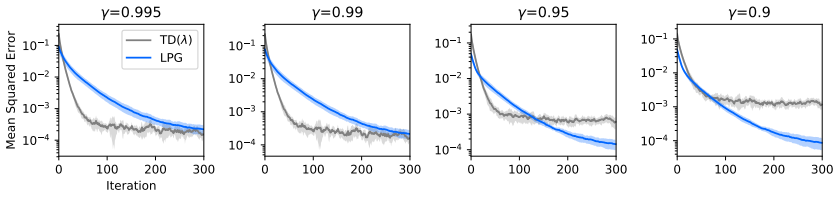
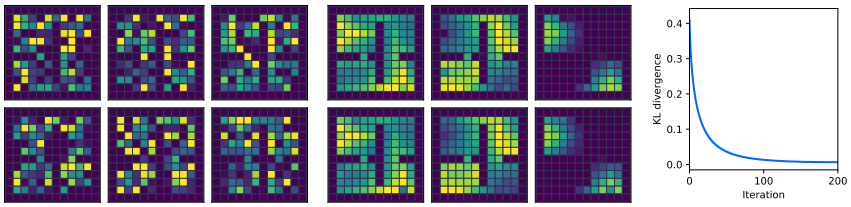


Figure 5: Value regression from predictions over the course of policy evaluation. Each plot shows mean squared errors to true values at various discount factors averaged over held-out 10 grid world instances.



(a) Initial predictions (b) After convergence (c) KL divergence

Figure 6: Convergence of predictions. (a-b) An example of two randomly initialised predictions (y_0, y_1) are shown at the top (y_0) and the bottom (y_1) (a) before and (b) after convergence. 3-dimensions are selected for visualisation. (c) The progression of $D_{\mathrm{KL}}(y_0||y_1)$ averaged over 128 grid world instances.

Does the prediction (y) capture true values and beyond? To further investigate how rich the predictions are, we generated y vectors as in Figure 4c for many grid world instances with a discount factor of 0.995 and trained a value regression model $g : \mathbb{R}^{30} \mapsto \mathbb{R}$, a 1-layer multi-layer perceptron (MLP), to predict true values just from predictions y for various discount factors from 0.995 to 0.9. We then evaluated how accurate the value regression is on a held-out set of grid worlds. For a comparison, we also trained a value regression model $h : \mathbb{R} \mapsto \mathbb{R}$ for TD(λ) from values at discount factor 0.995 to values at the other discount factors.

Interestingly, the result in Figure 5 shows that the value regression from y is almost as good as TD(λ) at the original discount factor (0.995) as updated by the LPG, which implies that the information in y is rich enough to recover the original concept of value function. More interestingly, Figure 5 also shows that y captures true values at lower discount factors, even though it was generated with a discount factor of 0.995. On the other hand, the information in the scalar value with TD(λ) is too limited to capture values at lower discount factors. This result suggests that the proposed framework can automatically discover a rich and useful semantics of predictions that can almost recover the value functions at various horizons, even though such a semantics was not enforced during meta-training.

Does the prediction (y) converge? The convergence of the predictions learned by different RL methods is one of their most critical properties. Classical algorithms, such as temporal-difference (TD) learning, have a convergence guarantee to a precisely defined semantics (i.e., the expected return) in tabular settings [30]. On the other hand, LPG does not have such a guarantee, because the prediction semantics is meta-learned with the sole aim to improve the performance of the agent, which means that LPG can, in principle, contain non-convergent dynamical systems that could cycle or diverge. Therefore, we empirically investigate the convergence property of LPG. The result in Figure 6 shows that two different prediction vectors (y_0, y_1) converge to almost the same values when updated by the LPG. This implies that a stationary semantics, to which predictions converge, has naturally emerged from the proposed framework even without any theoretical constraint.

4.4 Ablation Study

As discussed in Section 3.2, we find that meta-training is very hard and unstable as it is required to discover the entire update rule. Figure 7 summarises the effect of each idea introduced by this paper. 'LPG w/o Softmax(y)' uses $y \in \mathbb{R}^{30}$ without softmax but with $\|y - \hat{y}\|_2^2$ instead of KL-divergence in Eq. (2). 'LPG w/o Entropy(y)' and 'LPG w/o L2' are without entropy regularisation of y and without L2 regularisation of $\hat{\pi}, \hat{y}$ in Eq. (4) respectively. 'LPG fixed hyper' is trained with fixed hyperparameters for each training environment instead of balancing them during meta-training as introduced in Section 3.4. The result in Figure 7 shows that all of these ideas are crucial for the performance, and training tends to be very unstable without either of them. On the other hand, we

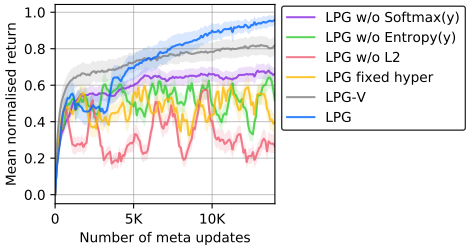


Figure 7: Ablation study. Each curve shows normalised return ($G_{\text{norm}} \in [0, 1]$) averaged over 15 training environments throughout meta-training.

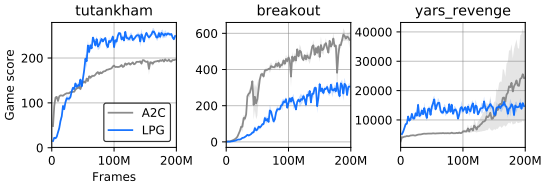


Figure 8: Example learning curves on Atari games. LPG outperforms A2C on `tutankham`, learns slowly on `breakout`, and prematurely converges to a sub-optimal policy on `yars-revenge`. The learning curves across all Atari games are available in the supplementary material.

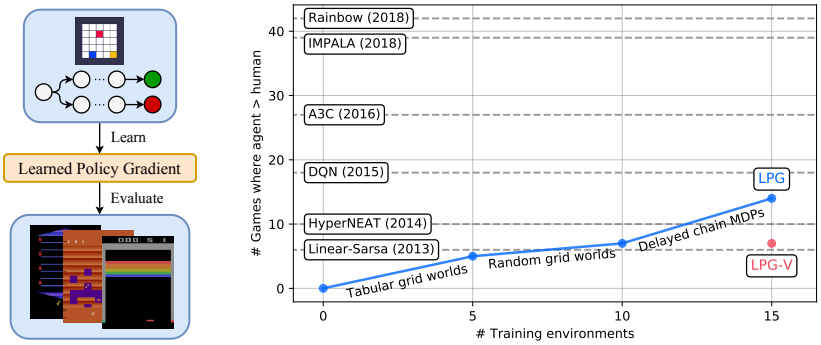


Figure 9: Generalisation from toy environments to Atari. X-axis is the number of toy environments used to meta-learn the LPG. Y-axis is the number of Atari games where the agent outperforms humans at the end of training. Dashed lines correspond to state-of-the-art algorithms for each year [2, 12, 22, 21, 8, 13].[3]

found that meta-training of LPG-V is stable even without regularisers. However, LPG-V converges to a sub-optimal update rule, whereas LPG eventually finds a better update rule by discovering what to predict. This result supports our hypothesis that discovering alternatives to value functions has the greater potential to find better update rules, although optimisation can be more difficult.

4.5 Generalising from Toy Environments to Atari Games

To see how general LPG can be when discovered solely from toy environments, we evaluated the LPG directly on complex Atari games. As summarised in Figure 9, the LPG generalises to Atari games reasonably well when compared to the advanced RL algorithms. This is surprising in that the training environments consist of mostly tabular environments with basic tasks that are much simpler than Atari games, and the LPG has never seen such complex domains during meta-training. Nevertheless, the agents trained with the LPG can learn complex behaviours across many Atari games achieving super-human performances on 14 games, without relying on any hand-designed RL components such as value function but rather using its own update rule discovered from scratch.

We found that specific types of training environments, such as delayed chain MDPs, significantly improved the generalisation performance (see Figure 9). This suggests that there may be a small but carefully designed set of environments that capture important challenges in RL so that when used for meta-training, the resulting LPG is general enough to perform well across many complex domains.

Although LPG is still behind the advanced RL algorithms such as A2C, the fact that LPG outperforms A2C on not just the training environments but also a few Atari games (see Figure 8 for example) implies that LPG specialises in a particular type of RL problems instead of being strictly worse than A2C. On the other hand, Figure 9 shows that the generalisation performance improves quickly as the number of training environments grows, which suggests that it may be feasible to discover a general-purpose RL algorithm once a larger set of environments are available for meta-training.

[3]This figure is for approximately showing the progression of LPG in parallel with human-discovered algorithms not for a strict comparison between algorithms due to different preprocessing and function approximators.

5 Conclusion

This paper made the first attempt to meta-learn a full RL update rule by jointly discovering both 'what to predict' and 'how to bootstrap', replacing existing RL concepts such as value function and TD-learning. The results from a small set of toy environments showed that the discovered LPG maintains rich information in the prediction, which was crucial for efficient bootstrapping. We believe this is just the beginning of the fully data-driven discovery of RL algorithms; there are many promising directions to extend our work, from procedural generation of environments, to new advanced architectures and alternative ways to generate experience. The radical generalisation from the toy domains to Atari games shows that it may be feasible to discover an efficient RL algorithm from interactions with environments, which would potentially lead to entirely new approaches to RL.

Broader Impact

The proposed approach has a potential to dramatically accelerate the process of discovering new reinforcement learning (RL) algorithms by automating the process of discovery in a data-driven way. If the proposed research direction succeeds, this could shift the research paradigm from manually developing RL algorithms to building a proper set of environments so that the resulting algorithm is efficient.

Additionally, the proposed approach may also serve as a tool to assist RL researchers in developing and improving their hand-designed algorithms. In this case, the proposed approach can be used to provide insights about what a good update rule looks like depending on the architecture that researchers provide as input, which could speed up the manual discovery of RL algorithms.

On the other hand, due to the data-driven nature of the proposed approach, the resulting algorithm may capture unintended bias in the training set of environments. In our work, we do not provide domain-specific information except rewards when discovering an algorithm, which makes it hard for the algorithm to capture bias in training environments. However, more work is needed to remove bias in the discovered algorithm to prevent potential negative outcomes.

Acknowledgement

We thank Simon Osindero for his helpful feedback on the manuscript.

References

[1] M. G. Bellemare, W. Dabney, and R. Munos. A distributional perspective on reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 449–458. JMLR. org, 2017.

[2] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

[3] Y. Bengio, S. Bengio, and J. Cloutier. *Learning a synaptic learning rule*. Citeseer, 1990.

[4] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, and S. Wanderman-Milne. JAX: composable transformations of Python+NumPy programs, 2018.

[5] Y. Chebotar, A. Molchanov, S. Bechtle, L. Righetti, F. Meier, and G. Sukhatme. Meta-learning via learned loss. *arXiv preprint arXiv:1906.05374*, 2019.

[6] W. Dabney, M. Rowland, M. G. Bellemare, and R. Munos. Distributional reinforcement learning with quantile regression. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[7] Y. Duan, J. Schulman, X. Chen, P. L. Bartlett, I. Sutskever, and P. Abbeel. Rl ^{2}: Fast reinforcement learning via slow reinforcement learning. *arXiv preprint arXiv:1611.02779*, 2016.

[8] L. Espeholt, H. Soyer, R. Munos, K. Simonyan, V. Mnih, T. Ward, Y. Doron, V. Firoiu, T. Harley, I. Dunning, et al. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. *arXiv preprint arXiv:1802.01561*, 2018.

[9] C. Finn, P. Abbeel, and S. Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1126–1135. JMLR. org, 2017.

[10] C. Finn and S. Levine. Meta-learning and universality: Deep representations and gradient descent can approximate any learning algorithm. *arXiv preprint arXiv:1710.11622*, 2017.

[11] S. Flennerhag, A. A. Rusu, R. Pascanu, H. Yin, and R. Hadsell. Meta-learning with warped gradient descent. *arXiv preprint arXiv:1909.00025*, 2019.

[12] M. Hausknecht, J. Lehman, R. Miikkulainen, and P. Stone. A neuroevolution approach to general atari game playing. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(4):355–366, 2014.

[13] M. Hessel, J. Modayil, H. Van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver. Rainbow: Combining improvements in deep reinforcement learning. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

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

[15] R. Houthooft, Y. Chen, P. Isola, B. Stadie, F. Wolski, O. J. Ho, and P. Abbeel. Evolved policy gradients. In *Advances in Neural Information Processing Systems*, pages 5400–5409, 2018.

[16] N. P. Jouppi, C. Young, N. Patil, D. A. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, R. C. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. In-datacenter performance analysis of a tensor processing unit. *CoRR*, abs/1704.04760, 2017.

[17] L. Kirsch, S. van Steenkiste, and J. Schmidhuber. Improving generalization in meta reinforcement learning using learned objectives. In *International Conference on Learning Representations*, 2020.

[18] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[19] T. Miconi, J. Clune, and K. O. Stanley. Differentiable plasticity: training plastic neural networks with backpropagation. *arXiv preprint arXiv:1804.02464*, 2018.

[20] T. Miconi, A. Rawal, J. Clune, and K. O. Stanley. Backpropamine: training self-modifying neural networks with differentiable neuromodulated plasticity. *arXiv preprint arXiv:2002.10585*, 2020.

[21] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937, 2016.

[22] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[23] A. Nichol, J. Achiam, and J. Schulman. On first-order meta-learning algorithms. *arXiv preprint arXiv:1803.02999*, 2018.

[24] I. Osband, Y. Doron, M. Hessel, J. Aslanides, E. Sezener, A. Saraiva, K. McKinney, T. Lattimore, C. Szepezvari, S. Singh, et al. Behaviour suite for reinforcement learning. *arXiv preprint arXiv:1908.03568*, 2019.

[25] A. Santoro, S. Bartunov, M. Botvinick, D. Wierstra, and T. Lillicrap. Meta-learning with memory-augmented neural networks. In *International conference on machine learning*, pages 1842–1850, 2016.

[26] J. Schmidhuber. Evolutionary principles in self-referential learning. Master's thesis, Technische Universitat Munchen, Germany, 1987.

[27] J. Schmidhuber. A 'self-referential'weight matrix. In *International Conference on Artificial Neural Networks*, pages 446–450. Springer, 1993.

[28] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller. Deterministic policy gradient algorithms. In *International conference on machine learning*, 2014.

[29] R. S. Sutton. Adapting bias by gradient descent: An incremental version of delta-bar-delta. In *AAAI*, pages 171–176, 1992.

[30] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[31] S. Thrun and T. M. Mitchell. Learning one more thing. In *IJCAI*, 1995.

[32] V. Veeriah, M. Hessel, Z. Xu, J. Rajendran, R. L. Lewis, J. Oh, H. P. van Hasselt, D. Silver, and S. Singh. Discovery of useful questions as auxiliary tasks. In *Advances in Neural Information Processing Systems*, pages 9306–9317, 2019.

[33] O. Vinyals, C. Blundell, T. Lillicrap, D. Wierstra, et al. Matching networks for one shot learning. In *Advances in neural information processing systems*, pages 3630–3638, 2016.

[34] J. X. Wang, Z. Kurth-Nelson, D. Tirumala, H. Soyer, J. Z. Leibo, R. Munos, C. Blundell, D. Kumaran, and M. Botvinick. Learning to reinforcement learn. *arXiv preprint arXiv:1611.05763*, 2016.

[35] Y. Wang, Q. Ye, and T.-Y. Liu. Beyond exponentially discounted sum: Automatic learning of return function. *arXiv preprint arXiv:1905.11591*, 2019.

[36] Z. Xu, H. van Hasselt, M. Hessel, J. Oh, S. Singh, and D. Silver. Meta-gradient reinforcement learning with an objective discovered online. *arXiv preprint*, 2020.

[37] Z. Xu, H. P. van Hasselt, and D. Silver. Meta-gradient reinforcement learning. In *Advances in neural information processing systems*, pages 2396–2407, 2018.

[38] T. Zahavy, Z. Xu, V. Veeriah, M. Hessel, J. Oh, H. van Hasselt, D. Silver, and S. Singh. Self-tuning deep reinforcement learning. *arXiv preprint arXiv:2002.12928*, 2020.

[39] Z. Zheng, J. Oh, M. Hessel, Z. Xu, M. Kroiss, H. van Hasselt, D. Silver, and S. Singh. What can learned intrinsic rewards capture? *arXiv preprint arXiv:1912.05500*, 2019.

[40] Z. Zheng, J. Oh, and S. Singh. On learning intrinsic rewards for policy gradient methods. In *Advances in Neural Information Processing Systems*, pages 4644–4654, 2018.

[41] W. Zhou, Y. Li, Y. Yang, H. Wang, and T. M. Hospedales. Online meta-critic learning for off-policy actor-critic methods. *arXiv preprint arXiv:2003.05334*, 2020.

A Training Environments

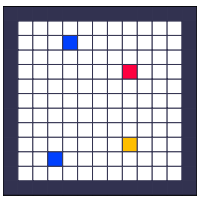
A.1 Tabular Grid World

When an agent collects an object, it receives the corresponding reward r, and the episode terminates with a probability of ϵ_{term} associated with the object. The object disappears when collected, and reappears with a probability of $\epsilon_{\text{respawn}}$ for each time-step. In the following sections, we describe each object type i as $\{N \times [r, \epsilon_{\text{term}}, \epsilon_{\text{respawn}}]\}_i$, where N is the number of objects with type i.

Observation Space In tabular grid worlds, object locations are randomised across lifetimes but fixed within a lifetime. Thus, there are only $p \times 2^{\text{m}}$ possible states in each lifetime, where p is the number of possible positions, and m is the total number of objects. An agent is simply represented by a table with distinct $\pi(a|s)$ and $y(s)$ values for each state without any function approximation.

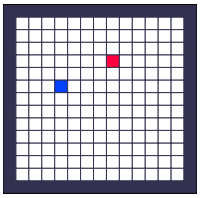
Action Space There are two different action spaces. One version consists of 9 movement actions for adjacent positions (including staying at the same position) and 9 actions for collecting objects at adjacent positions. The other version has only 9 movement actions. In this version, an object is automatically collected when the agent visits it. We randomly sample either one of the action spaces for each lifetime during meta-training.

A.1.1 Dense



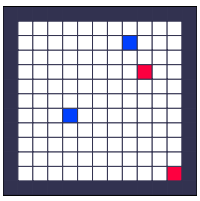
Component	Description
Observation	State index (integer)
Number of actions	9 or 18
Size	11×11
Objects	$2 \times [1, 0, 0.05]$, $[-1, 0.5, 0.1]$, $[-1, 0, 0.5]$
Maximum steps per episode	500

A.1.2 Sparse



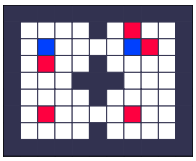
Component	Description
Observation	State index (integer)
Number of actions	9 or 18
Size	13×13
Objects	$[1, 1, 0]$, $[-1, 1, 0]$
Maximum steps per episode	50

A.1.3 Long Horizon



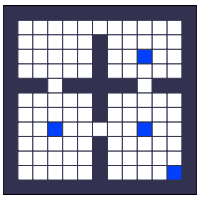
Component	Description
Observation	State index (integer)
Number of actions	9 or 18
Size	11×11
Objects	$2 \times [1, 0, 0.01]$, $2 \times [-1, 0.5, 1]$
Maximum steps per episode	1000

A.1.4 Longer Horizon



Component	Description
Observation	State index (integer)
Number of actions	9 or 18
Size	7×9
Objects	$2 \times [1, 0.1, 0.01]$, $5 \times [-1, 0.8, 1]$
Maximum steps per episode	2000

A.1.5 Long Dense

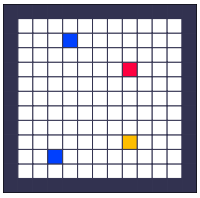


Component	Description
Observation	State index (integer)
Number of actions	9 or 18
Size	11×11
Objects	$4 \times [1, 0, 0.005]$
Maximum steps per episode	2000

A.2 Random Grid World

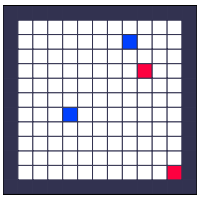
The random grid worlds are almost the same as the tabular grid worlds except that object locations are randomised within a lifetime. More specifically, object locations are randomly determined at the beginning of each episode, and objects re-appear at random locations after being collected. Due to the randomness, the state space is exponentially large, which requires function approximation to represent an agent. The observation consists of a tensor $\{0, 1\}^{N \times H \times W}$, where N is the number of object types, $H \times W$ is the size of the grid.

A.2.1 Dense



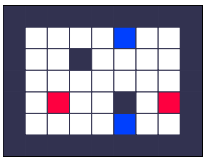
Component	Description
Observation	$\{0, 1\}^{N \times H \times W}$
Number of actions	9 or 18
Size	11×11
Objects	$2 \times [1, 0, 0.05]$, $[-1, 0.5, 0.1]$, $[-1, 0, 0.5]$
Maximum steps per episode	500

A.2.2 Long Horizon



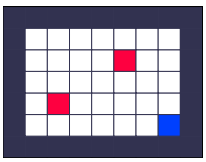
Component	Description
Observation	$\{0, 1\}^{N \times H \times W}$
Number of actions	9 or 18
Size	11×11
Objects	$2 \times [1, 0, 0.01]$, $2 \times [-1, 0.5, 1]$
Maximum steps per episode	1000

A.2.3 Small



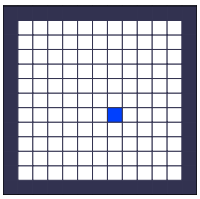
Component	Description
Observation	$\{0,1\}^{N\times H\times W}$
Number of actions	9 or 18
Size	5×7
Objects	$2 \times [1, 0, 0.05]$, $2 \times [-1, 0.5, 0.1]$
Maximum steps per episode	500

A.2.4 Small Sparse



Component	Description
Observation	$\{0,1\}^{N\times H\times W}$
Number of actions	9 or 18
Size	5×7
Objects	$[1, 1, 1]$, $2 \times [-1, 1, 1]$
Maximum steps per episode	50

A.2.5 Very Dense

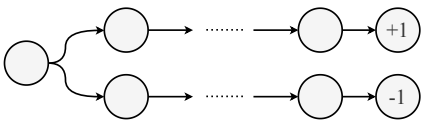


Component	Description
Observation	$\{0,1\}^{N\times H\times W}$
Number of actions	9 or 18
Size	11×11
Objects	$[1, 0, 1]$
Maximum steps per episode	2000

A.3 Delayed Chain MDP

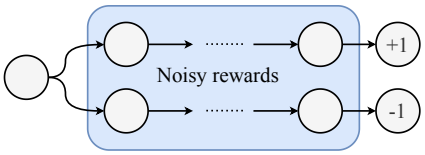
This environment is inspired by the *Umbrella* environment in Behaviour Suite [24]. The agent has a binary choice (a_0, a_1) for each time-step. The first action determines the reward at the end of the episode (1 or -1). The episode terminates after a fixed number of steps (i.e., chain length), which is sampled randomly from a pre-defined range for each lifetime and fixed within a lifetime. For each episode, we randomly determine which action leads to a positive reward and sample the corresponding chain MDP. There is no state aliasing because all states are distinct. Optionally, there can be noisy rewards $\{1, -1\}$ for the states in the middle that are independent of the agent's action.

A.3.1 Short



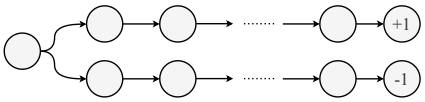
Component	Description
Observation	State index (integer)
Number of actions	2
Chain length	$[5, 30]$
Noisy rewards	No

A.3.2 Short and Noisy



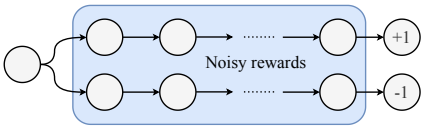
Component	Description
Observation	State index (integer)
Number of actions	2
Chain length	$[5, 30]$
Noisy rewards	Yes

A.3.3 Long



Component	Description
Observation	State index (integer)
Number of actions	2
Chain length	$[5, 50]$
Noisy rewards	No

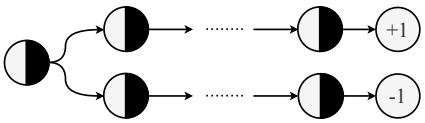
A.3.4 Long and Noisy



Component	Description
Observation	State index (integer)
Number of actions	2
Chain length	$[5, 50]$
Noisy rewards	Yes

A.3.5 State Distraction

In this delayed chain MDP, an observation $s_t \in \{0, 1\}^{22}$ consists of two relevant bits: whether a_0 is the correct action and whether the agent has chosen the correct action, and noisy bits $\{0, 1\}^{20}$ that are randomly sampled independently for all states. The agent is required to find out the relevant bits while ignoring the noisy bits in the observation.



Component	Description
Observation	$\{0, 1\}^{22}$
Number of actions	2
Chain length	$[5, 30]$
Noisy rewards	No

B Implementation Details

B.1 Meta-Training

We trained LPGs by simulating 960 parallel lifetimes (i.e., batch size for meta-gradients), each of which has a learning agent interacting with a sampled environment, for approximately 10^{10} steps of interactions in total. In each lifetime, the agent updates its parameters using a batch of trajectories generated from 64 parallel environments (i.e., batch size for agent). Each trajectory consists of 20 steps. Thus, each parameter update consists of 64×20 steps. The meta-hyperparameters used for meta-training is summarised in Table 2.

Details of LPG Architecture The LPG network takes $x_t = [r_t, d_t, \gamma, \pi(a_t|s_t), y_\theta(s_t), y_\theta(s_{t+1})]$ at each time-step t, where r_t is a reward, d_t is a binary value indicating episode-termination, and γ is a discount factor. $y_\theta(s_t)$ and $y_\theta(s_{t+1})$ are mapped to a scalar using a shared embedding network (φ): Dense(16)-Dense(1). A backward LSTM with 256 units takes $[r_t, d_t, \gamma, \pi(a_t|s_t), \varphi(y_\theta(s_t)), \varphi(y_\theta(s_{t+1}))]$ as input and produces $\hat{\pi}$ and $\hat{y}$ as output. We slightly modified the LSTM core such that the hidden states are reset for terminal states ($d_t = 0$), which blocks information from flowing across episodes. In our preliminary experiment, this improved generalisation performance by making it difficult for LPG to exploit environment-specific patterns. Rectified linear unit (ReLU) was used as activation function throughout the experiment.

Details of LPG Update In Section 3.3, the meta-gradient for updating LPG is described as the outcome of REINFORCE for simplicity. In practice, however, we used advantage actor-critic (A2C) [21] to calculate the meta-gradient, which requires learning value functions for bootstrapping. Note that value functions were trained only to reduce the variance of meta-gradient. LPG itself has no access to value functions during meta-training and meta-testing. In principle, the *outer* algorithm used for discovery can be any RL algorithm, as long as they are designed to maximise cumulative rewards.

Details of Hyperparameter Balancing As described in Section 3.4, we trained a bandit $p(\alpha|\mathcal{E})$ to automatically sample better agent hyperparameters for each environment to make meta-training more stable. More specifically, the bandit samples hyperparameters at the beginning of each lifetime according to:

$$p(\alpha|\mathcal{E}) \propto \exp\left(\frac{R(\alpha, \mathcal{E}) + \rho/\sqrt{N(\alpha, \mathcal{E})}}{\tau}\right), \tag{6}$$

where $R(\alpha, \mathcal{E})$ is the final return at the end of the agent's lifetime with hyperparameters α in environment $\mathcal{E}$, which is averaged over the last 10 lifetimes. $N(\alpha, \mathcal{E})$ is the number of lifetimes simulated. τ is a constant temperature, and ρ is a coefficient for exploration bonus. Intuitively, we keep track of how well each α performs and sample hyperparmeters that tend to produce a larger final return with exploration bonus. In our experiments, α consists of two hyperparameters: learning rate (α_{lr}) and KL cost (α_y) for updating the agent's predictions. Table 3 shows the range of hyperparameters searched by the bandit. Note that this hyperparameter balancing requires multiple lifetimes of experience, which can be done only during meta-training. During meta-testing on unseen environments, α needs to be manually selected.

Preventing Early Divergence We found that meta-training can be unstable especially early in training, because the randomly initialised update rule (η) tends to make agents diverge or deterministic, which eventually causes exploding meta-gradients. To address this issue, we reset the lifetime whenever the entropy of the policy becomes 0, which means the policy becomes deterministic. We observed that this is triggered a few times early in training but eventually is not triggered later in training as the update rule improves.

Table 2: Meta-hyperparameters for meta-training.

Hyperparameter	Value	Searched values
Optimiser	Adam	-
Learning rate	0.0001	{0.0005, 0.0001, 0.00003}
Discount factor (γ)	{0.997, 0.995, 0.99}	-
Policy entropy cost (β_0)	{0.01, 0.02}	-
Prediction entropy cost (β_1)	0.001	{0.001, 0.0001}
L2 regularisation weight for $\hat{\pi}$ (β_2)	0.001	{0.01, 0.001}
L2 regularisation weight for $\hat{y}$ (β_3)	0.001	{0.01, 0.001}
Bandit temperature (τ)	0.1	{1, 0.1}
Bandit exploration bonus (ρ)	0.2	{1, 0.2}
Number of steps for each trajectory	20	-
Number of parameter updates (K)	5	-
Number of parallel lifetimes	960	-
Number of parallel environments per lifetime	64	-

Discount factor and policy entropy cost are randomly sampled from the specified range for each lifetime.

Table 3: Agent hyperparameters for each training environment.

Environment	Architecture	Optimiser	Learning rate (α_{lr})	KL cost (α_y)	Lifetime
`dense`	Tabular	SGD	$\{20, 40, 80\}$	$\{0.1, 0.5, 1\}$	3M
`sparse`	Tabular	SGD	$\{20, 40, 80\}$	$\{0.1, 0.5, 1\}$	3M
`long_horizon`	Tabular	SGD	$\{20, 40, 80\}$	$\{0.1, 0.5, 1\}$	3M
`longer_horizon`	Tabular	SGD	$\{20, 40, 80\}$	$\{0.1, 0.5, 1\}$	3M
`long_dense`	Tabular	SGD	$\{20, 40, 80\}$	$\{0.1, 0.5, 1\}$	3M
`dense`	C(16)-D(32)	Adam	$\{0.0005, 0.001, 0.002, 0.005\}$	$\{0.1, 0.5, 1\}$	30M
`long_horizon`	C(16)-D(32)	Adam	$\{0.0005, 0.001, 0.002, 0.005\}$	$\{0.1, 0.5, 1\}$	30M
`small`	D(32)	Adam	$\{0.0005, 0.001, 0.002, 0.005\}$	$\{0.1, 0.5, 1\}$	30M
`sparse`	D(32)	Adam	$\{0.0005, 0.001, 0.002, 0.005\}$	$\{0.1, 0.5, 1\}$	30M
`very_dense`	C(32-16-16)-D(256)	Adam	$\{0.0005, 0.001, 0.002, 0.005\}$	$\{0.1, 0.5, 1\}$	30M
`short`	Tabular	SGD	$\{20, 40, 80\}$	$\{0.1, 0.5, 1\}$	1M
`short_noisy`	Tabular	SGD	$\{20, 40, 80\}$	$\{0.1, 0.5, 1\}$	1M
`long`	Tabular	SGD	$\{20, 40, 80\}$	$\{0.1, 0.5, 1\}$	1M
`long_noisy`	Tabular	SGD	$\{20, 40, 80\}$	$\{0.1, 0.5, 1\}$	1M
`distractor`	D(16)	Adam	$\{0.002, 0.005, 0.01\}$	$\{0.1, 0.5, 1\}$	2M

'C(N1-N2-...)' represents convolutional layers with N1, N2, ... filters for each layer.
'D(N)' represents a dense layer with N units.
Lifetime is defined as the total number of steps.

B.2 Meta-Testing

We selected the best update rule (η) and hyperparameters according to the validation performance on two Atari games (`breakout, boxing`), and used them to evaluate across all 57 Atari games. We found that subtracting a baseline slightly improves the performance on Atari games as follows:

$$\Delta\theta \propto \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a|s)(\hat{\pi} - f_\theta(s)) - \alpha_y \nabla_\theta D_{\mathrm{KL}}(y_\theta(s)\|\hat{y}) - \frac{1}{2}||f_\theta(s) - \hat{\pi}||^2\right], \tag{7}$$

where $f_\theta(s)$ is an action-independent baseline function. The hyperparameters are summarised in Table 4, and the learning curves are shown in Figure 10.

B.3 Computing Infrastructure

Our implementation is based on JAX [4] using TPUs [16]. The training environments are also implemented in JAX, which enables running on TPU as well. It took approximately 24 hours to converge using a 16-core TPU-v2.

Table 4: Hyperparameters used for meta-testing on Atari games.

Hyperparameter	Value	Searched values
Optimiser	Adam	-
Network architecture	C(32)-C(64)-C(64)-D(512)	-
Learning rate (α_{lr})	0.0005	{0.001, 0.0005, 0.0003}
KL cost (α_{y})	0.5	{1, 0.5, 0.1}
Discount factor (γ)	0.995	-
Number of steps for each trajectory	20	-
Number of parallel environments (batch size)	30	-

C Generalisation to Atari Games

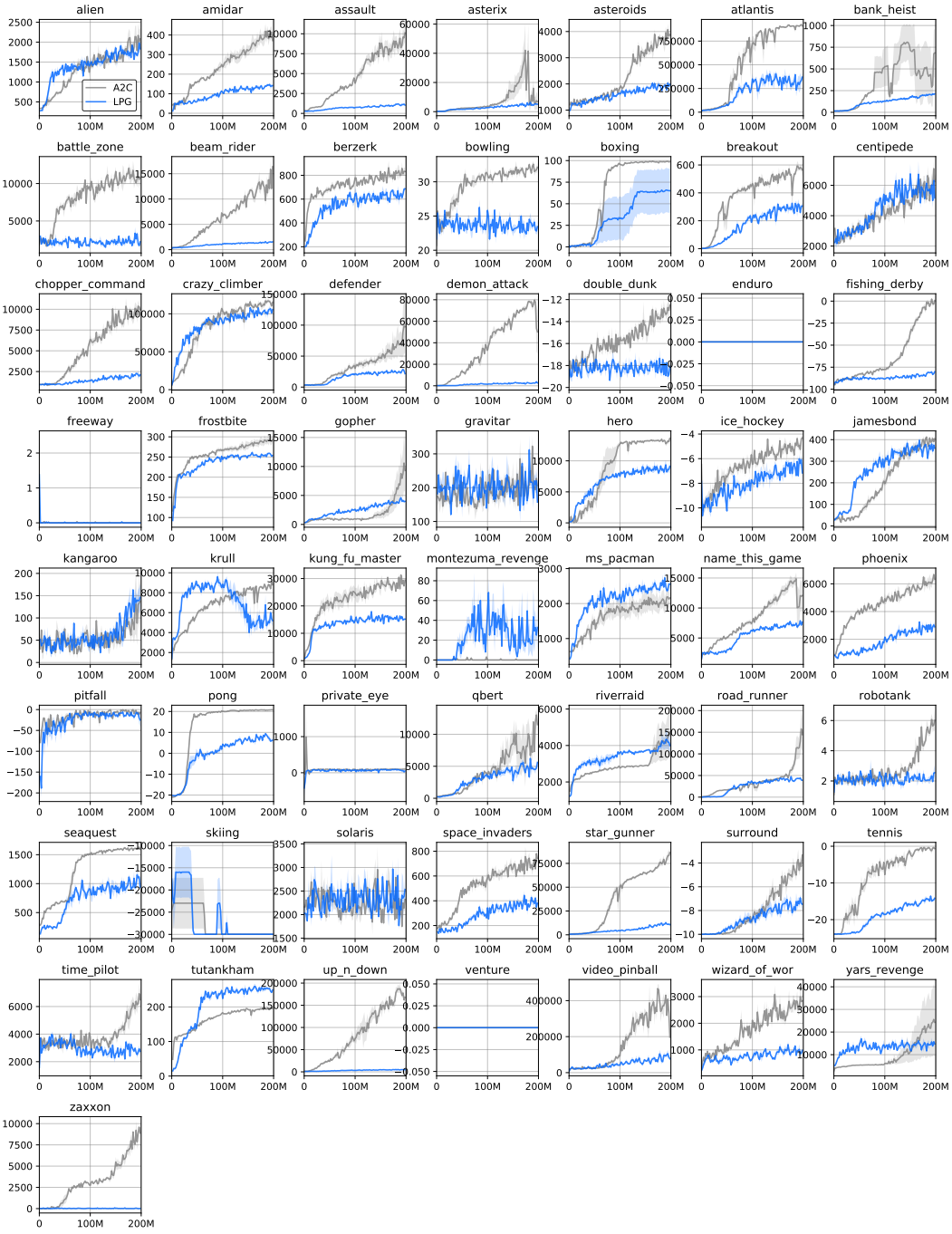


Figure 10: Learning curves on Atari games. X-axis and y-axis represent the number of frames and episode return respectively. Shaded areas show standard errors from 3 independent runs.