Meta-learning Adaptive Intrinsic Reward Weighting for Curiosity-Driven Reinforcement Learning

by
Batsirayi Mupamhi Ziki

Supervised by:
Associate Professor Jonathan Shock

Co-supervised by:
Dr. Andries Smit

This thesis is submitted for the degree of Master of Science in Applied Mathematics

Department of Mathematics and Applied Mathematics
University of Cape Town

30 June 2025

Plagiarism Declaration

1. I know that plagiarism is a serious form of academic misconduct.
2. I have read the document about avoiding plagiarism, am familiar with its contents and have avoided all forms of plagiarism mentioned there.
3. Where I have used the word of others, I have indicated this by the use of quotation marks.
4. I have referenced all quotations and properly acknowledged ideas borrowed from others.
5. I have not and shall not allow others to plagiarise my work.
6. I declare that this is my own work.

Signature: Signed by candidate

Batsirayi Ziki
June 2025

Acknowledgments

First of all, I would like to acknowledge the sources of funding that allowed me to complete my master's degree. I am especially grateful to the Department of Mathematics and Applied Mathematics at UCT for awarding me the Science Faculty Master's Equity Scholarship. I am incredibly thankful for that support. I would also like to thank the University and the donors of the Manuel and Luby Washkansky Scholarship. Thanks to this scholarship, I was able to purchase a new laptop for my research. I am truly grateful for these two scholarships, which allowed me to focus entirely on my studies without the added stress of financial concerns.

I would also like to thank my supervisors. Associate Professor Jonathan Shock, thank you for your guidance, your patience, and your constant encouragement. You always pushed me to perform at my best and to aim high, and I am especially grateful for the many recommendations you wrote on my behalf. More importantly, thank you for caring about me not just as a student, but also as a person.

To my co-supervisor, Dr Andries Smit, thank you for your mentorship from the start of my internship at InstaDeep all the way through my master's. Your help with code, technical guidance, and advice has been invaluable in helping me grow as a researcher. It has been an honour to learn from someone so accomplished in the field. I am especially grateful for your willingness to meet with me on short notice, even for small JAX issues. Your support means a lot to me.

Next, I would like to thank the TRC Program at Google for granting me free access to their cloud infrastructure. Through this program, I was able to use their TPUs, specifically version 4, which significantly accelerated my research. Tasks that would have taken weeks, or even a month, were reduced to just a few days. I am incredibly grateful for their support, and I appreciate how responsive and generous they were in extending my trial whenever I needed additional compute resources.

I would also like to thank InstaDeep, particularly the Cape Town office, for their incredible support. I am especially grateful for being allowed to keep my PC at

the office so I could continue working while my TPU jobs were running, without having to worry about load shedding. I truly appreciated their open-door policy whenever I needed a change of environment from campus or home. The supportive atmosphere, the curiosity about my research, and the opportunity to interact with incredibly intelligent people, many of whom think very differently from me, helped me grow in many ways.

I also want to thank the amazing community I'm part of: the Beach Bobbers. Meeting every Monday and Thursday for cold dips in the ocean truly became an essential part of my routine. Those sessions were an endorphin boost and a grounding ritual. What started as casual meetups with strangers became friendships with some of the most genuine and caring people I know. They always made space to listen, support, and uplift me. Mondays and Thursdays with the Bobbers were genuinely essential to me completing this degree, and I'm deeply grateful to everyone in that community.

And last but certainly not least, my most important acknowledgement is to my family. I want to thank my parents for supporting me throughout this journey, and especially for all their prayers. I just want to say: your prayers worked, I've made it to the finish line! I'm so deeply grateful not only for the support you've given me during my master's, but for the love and encouragement you've shown me my entire life. You've always been my source of strength and morale, sending me money so I could grab takeout, calling to check in, answering my calls at all hours, and always being willing to pray for me or simply listen when I needed someone. You were the very first people to believe in me, and your unwavering belief has carried me through so much. The past couple of decades of your support, raising me, encouraging me to be a better person, and reminding me that no matter what, you've always got my back, has shaped who I am today. I will always be grateful for everything you've done for me.

To my sisters, thank you too. Our weekly Sunday gaming sessions meant a lot to me. And sure, sometimes you annoyed me, but you supported me even more, so I'd say it was a net positive. I'm really thankful for the fun, the distraction, and the care you both gave me. To my whole family, thank you. I am forever in your debt, and it's a debt I can never truly repay. Your love and support helped me more than I can say, and I'll carry that with me always.

Research Abstract

For both organisms and artificial agents, exploration is essential to continue learning and avoid becoming trapped in suboptimal behaviours. Reinforcement learning (RL) agents can also face exploration challenges in environments with sparse feedback. Curiosity-driven exploration algorithms can help address these challenges by providing intrinsic rewards based on the novelty of situations an agent encounters. These intrinsic rewards are typically combined with extrinsic rewards using a weighted sum with the parameter λ. However, fine-tuning λ for each task across multiple environments can become computationally expensive. We propose a meta-learning approach for automatic tuning of λ using a recurrent neural network (RNN) that dynamically outputs λ values. We call this RNN the reward combiner. The reward combiner was trained using evolutionary strategies on XLand-MiniGrid environments, where feedback is sparse. The fitness function was the total extrinsic reward obtained during the training phase of an agent. We used BYOL-Explore, a curiosity-driven exploration algorithm, for intrinsic reward generation. The reward combiner takes normalised extrinsic and intrinsic rewards as input, along with actions that provide task-specific context for λ selection. Trained on `Unlock` and `Empty-16x16` environments, the reward combiner generalises across different grid sizes of the same task, outperforming baselines when tested on `DoorKey` environments. It also generalises across different tasks when tested on `UnlockPickUp`, where the objective differs from the training environments. Our approach achieves higher extrinsic returns at the end of training than curiosity-driven baselines across all test environments. Despite being tested only within XLand-MiniGrid environments, our results indicate this approach has potential to eliminate costly hyperparameter sweeps when switching to new tasks with similar mechanics.

Contents

List of Figures

Chapter 1

Introduction

The field of artificial intelligence (AI) has seen rapid growth in recent years, transforming numerous real-world domains [11], including healthcare [12], weather prediction [13] and video games [14, 15]. One focus of AI is in developing general-purpose agents as it has the potential to significantly grow the world economy and improve the standard of living [11]. A general-purpose agent is informally defined as an agent capable of achieving its goals across a wide range of environments [16, p. 402]. In the field of AI, machine learning (ML) has become a key approach for enabling such general-purpose capabilities. ML focuses on designing computational systems that can learn patterns from data, where the rules are inferred rather than explicitly programmed.

Within ML, there are three main branches: supervised learning, unsupervised learning, and reinforcement learning (RL). Supervised learning involves training models on labelled datasets to learn patterns and make accurate predictions on unseen data. In contrast, unsupervised learning involves training models on unlabelled data to uncover hidden structures or patterns, often resulting in clustering. RL is about learning from interactions with an environment. RL agents rely on feedback, or rewards, to determine which actions they took were beneficial and which were not. An agent's goal is to maximise the total discounted cumulative reward, known as the return.

To illustrate these different ML categories, consider the following examples. In supervised learning, one application is spam email detection. This is where you train a model on emails labelled as spam or not spam. The model can then be deployed to detect new spam emails and filter them out. In unsupervised learning, a common application is customer segmentation, where the model groups customers based on purchasing behaviour or other attributes without any predefined labels. For example, a video game studio can use an unsupervised learning algorithm to group customers by behavioural parameters. For example, people with high engagement tend to buy more cosmetics from the video game store than people who hardly play. Therefore, the company looks for ways to

boost engagement. Finally, in RL, consider teaching an agent to play a video game such as Space Invaders. In Space Invaders, the goal is to defeat the enemies from space before they kill you or make their way down to Earth. The agent receives a reward for every enemy killed, and the objective of the game is to kill as many enemies as possible. Figure 1.1 shows a screenshot of the game, where the agent controls the green laser cannon. Through repeated interactions with the Space Invaders environment, the agent eventually learns that killing the enemies before they harm it or make their way down maximises the return.

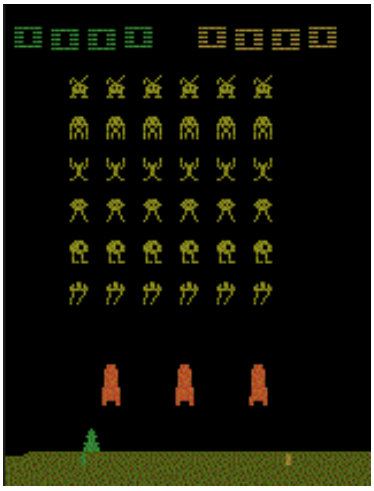


Figure 1.1: A screenshot of Space Invaders. Taken from [1].

It is important to notice that in the RL example above, we described the objective of Space Invaders as maximising the return. To do so, we made use of the reward hypothesis. The reward hypothesis states that goals can be described as maximising the expected value of the cumulative sum of a scalar signal (reward) that the agent receives [17, p. 53] and serves as the cornerstone of RL. This reward hypothesis, together with the informal definition of general-purpose agents presented earlier, explains why RL, the primary focus of this thesis, is considered a promising approach to developing general-purpose AI agents. One challenge in RL is the control problem. Control in the RL context refers to finding the optimal strategy to maximise cumulative rewards when interacting with an environment.

In order to solve the control problem, RL relies on designing a reward function that outputs a scalar signal, which is typically set up in such a way as to encourage the agent to learn to act in a desired manner. In RL, agents must often explore their environment to increase their knowledge and discover more rewarding actions over time [17]. Exploration enables agents to expand their knowledge about the environment and avoid suboptimal behaviours [18]. If the reward function is dense, i.e., it provides frequent feedback, the agent is often

able to solve the task and achieve the goal. For certain tasks, it is straightforward to design a dense reward function. The video game Space Invaders is one such task. In Space Invaders, the reward function is simply +1 for every alien killed.

However, in some cases, designing a dense reward function is challenging. Certain tasks or games only reward the agent in critical states that are far apart or rare, or in terminal states of the environment. Such reward functions are known as sparse. This means feedback comes at a low frequency. A simple example is the game of chess where the agent only obtains feedback once the game is over: +1 for winning, -1 for losing and 0 if it is a draw. But this feedback comes after many moves have been performed, and so the agent does not know which of its moves were beneficial, known as the credit assignment problem (CAP).

The CAP concerns learning to associate actions with distant future outcomes [19]. Tasks involving the CAP have two important characteristics: they take a long time to complete and the rewards are sparse. Solving the CAP requires agents to learn efficiently in environments with sparse rewards, which is critical for deploying RL in real-world applications where sparse rewards are common [19].

The video game Montezuma's Revenge is a more extreme example of a sparse reward setting [20]. Figure 1.2 shows a screenshot of Montezuma's Revenge. In this game, the objective is to collect treasures scattered across multiple rooms in a temple. To achieve this, the player must navigate through different rooms, collect keys to unlock doors, and avoid traps and enemies while delving deeper into the temple. In Montezuma's Revenge, critical states include unlocking doors and picking up keys. The agent receives rewards in these states. However, obtaining a key often requires performing a sequence of actions, such as climbing ladders and jumping over obstacles. Furthermore, reaching a door and unlocking it requires the agent to perform a complex sequence of actions. As a result, rewards in this environment can be spaced hundreds of steps apart, even when the agent follows an optimal policy [20].

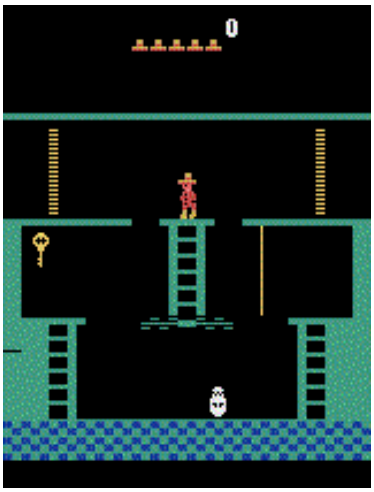


Figure 1.2: A screenshot of Montezuma's Revenge. Taken from [1].

When the reward function is sparse, the agent can struggle to achieve its goal. In such sparse reward environments, agents particularly need efficient exploration techniques, as the absence of rewards makes directed exploration essential [20]. Intrinsically motivated reinforcement learning (IMRL) is one approach that promotes exploration by instilling intrinsic motivation into the agents. These methods are generally applicable across different tasks without requiring domain-specific knowledge, making them more generalisable than hand-crafted reward shaping techniques.

One way to instil intrinsic motivation is to use curiosity-driven exploration algorithms. These algorithms reward the agent for being curious, which encourages it to explore the unknown states of the environment. Such techniques have led to significant progress in solving challenging sparse-reward environments, including improved performance on Montezuma's Revenge [6, 20], where agents have managed to explore more rooms and achieve higher scores compared to earlier approaches.

In curiosity-driven exploration, the agent receives two streams of rewards: the intrinsic reward, r_i, and the extrinsic reward, r_e. The intrinsic reward is typically provided when the agent reaches a novel state in the environment, with higher intrinsic rewards for more novel states. Extrinsic rewards, on the other hand, come from the environment itself. The two rewards are usually combined using a weighted sum:

$$\hat{r} = r_e + \lambda r_i.$$

where λ is a hyperparameter representing the relative weighting of the intrinsic reward.

Finding a suitable value for λ that works well across different tasks is challenging, as λ typically remains fixed during the agent's training. When switching to a new environment, a hyperparameter sweep is usually required to determine an appropriate λ value for that environment. This process is time-consuming and computationally expensive, especially when training agents on multiple tasks, each requiring different λ values for optimal performance.

This is even true for environments within the same domain such as grid-world environments. Grid-world environments are discrete navigation tasks with sparse rewards where the agent must explore to achieve its objective. These environments are sparse because the agent only receives extrinsic rewards upon completing the task objective. Each environment can have different objectives such as navigating to reach a goal position, navigating to pick up a key and unlock a door, or navigating to pick up an object. Even within grid-world tasks, tasks with similar mechanics but different objectives, such as one task requiring unlocking a door to reach a goal position versus another requiring unlocking to pick up an object, still require different λ values for optimal performance.

Meta-learning offers a potential solution to this hyperparameter tuning challenge. Meta-learning involves analysing how ML models perform across a variety of tasks and leveraging that experience to enable faster learning on new, unseen tasks [21]. Performing meta-learning in RL is known as meta-reinforcement learning (meta-RL). Meta-RL focuses on training agents to learn components of the RL algorithm, enabling generalisation to unseen tasks [22]. When the meta-learner is learning part of the RL algorithm, for example learning intrinsic reward weighting, it is common to provide the meta-learner with many episodes or time-steps to adapt to unseen tasks [22]. This is known as many-shot meta-RL.

Our research question is thus as follows:

> **Research Question:** How does the performance of an agent that learns to dynamically output intrinsic reward weightings (λ) compare to manually fine-tuned curiosity-driven algorithms and standard PPO across grid-world tasks involving navigation and/or door-key interaction mechanics?

To answer this research question, we focus on XLand-MiniGrid environments [8], which are JAX implementations of MiniGrid environments [23], discrete grid-world environments. These environments are well-suited for studying λ generalisation as they are sparse-rewarded, requiring curiosity-driven exploration, while being simple and quick to run. In particular, we focus on environments involving navigation and/or door-key interaction mechanics. Our objective is for the agent to generalise across different test tasks with navigation and/or door-key interaction mechanics that were unseen during training, since various environments within XLand-MiniGrid with similar mechanics require different λ values for optimal performance. By generalisation, we mean performing as

well as the fine-tuned curiosity-driven baselines (in terms of episode return at the end of training) in environments not seen during training.

We divide the research question into two key aspects of generalisation: across tasks with varying grid-sizes, and across environments where the objective differs from those seen during training. Therefore, we end up with the following sub-questions:

Sub-question 1: How does the meta-learning agent's performance compare with fine-tuned curiosity-driven algorithms and standard PPO when evaluating across different grid-world sizes on an unseen task?

Sub-question 2: How does the meta-learning agent's performance compare with fine-tuned curiosity-driven algorithms and standard PPO when evaluating on an unseen task where the objective differs from those seen during training?

We hypothesise that a neural network (which we refer to as the reward combiner) can learn to dynamically output λ values by observing normalised intrinsic and extrinsic rewards along with agent actions during the training of the RL agent. The reward combiner's objective would be to maximise the cumulative extrinsic rewards obtained across the RL agent's training lifetime. Formally, our hypothesis for each sub-question is as follows:

Hypothesis 1: The meta-learning agent will perform as well as the baselines when evaluating across different grid-world sizes on an unseen task with navigation and/or door-key interaction mechanics.

Hypothesis 2: The meta-learning agent will perform as well as the baselines when evaluating on an unseen task with navigation and/or door-key interaction mechanics where the objective differs from those seen during training.

We will evaluate whether the agent generalises by comparing the mean episode return at the end of training between agents trained with the reward combiner and those trained with two curiosity-driven algorithms: random network distillation (RND) [20] and BYOL-Explore [6], as well as a baseline agent without intrinsic rewards.

Our contribution is demonstrating that a reward combiner can generalise across tasks with similar mechanics within grid-world environments without retraining, showing the potential for eliminating hyperparameter sweeps in new environments.

The roadmap for testing our hypotheses is outlined as follows: Chapter 2 covers the background needed to understand our methodology, followed by Chapter

3 which reviews related work. Chapter 4 details our methodology, Chapter 5 presents the experimental results, and discusses the implications of our findings and limitations of our approach. Finally, Chapter 6 presents our conclusions and future work.

Chapter 2

Background

The foundations needed to understand our methodology and the results are presented in this chapter. The chapter begins by covering tabular reinforcement learning (RL) methods, concluding with Q-learning, as modern RL algorithms build upon the ideas from tabular RL. The RL algorithm we make use of in this project is the gated recurrent unit (GRU) [24] implementation of Proximal Policy Optimisation (PPO) [25], a deep RL algorithm. Therefore, in the next section we provide the necessary background to understand it by covering deep RL. We start with value-based methods, beginning with the Deep Q-Network (DQN) [26]. This is followed by a discussion of policy-based methods. The section concludes with actor-critic methods, which combine value-based and policy-based approaches. Our work involves curiosity-driven exploration algorithms, which aim to improve the exploration capabilities of RL agents. Therefore, we next cover intrinsically motivated RL and examine how researchers instil intrinsic motivation into agents. This is followed by a section on meta-RL, where algorithms learn how to perform reinforcement learning. Since our approach involves learning a component of an intrinsically motivated RL system, we rely on ideas from meta-RL. Finally, we make use of evolutionary strategies (ES) [27] to optimise our reward combiner. Therefore, we conclude with a discussion of ES.

2.1 Reinforcement Learning

As mentioned earlier, RL is about learning through interaction with an environment. It is based on the reward hypothesis, meaning the goal of RL is to maximise the cumulative sum of rewards that the agent receives. RL differs from other fields of machine learning, such as supervised learning, in that the agent generates its own data as it learns from experience.

In this section, we begin by explaining the Markov Decision Process (MDP), which provides a mathematical framework for agent-environment interaction in

RL. Finally, we conclude with Temporal-Difference Learning, where we explore how to approach the RL problem when the dynamics of the environment are unknown.

2.1.1 Markov Decision Processes

Agent-Environment Interaction

Markov Decision Processes provide a mathematical framework for sequential decision-making scenarios, whereby actions of the decision maker or agent influence not just the rewards but also the next states, and these next states influence the future rewards the agent receives. MDPs are the mathematical form of the RL problem where we are able to make precise mathematical statements. In MDPs agents interact with a environment, which is defined as everything outside the agent itself. The agent-environment interaction occurs at discrete time-steps, t. The agent interacts with an environment through actions and the environment responds by providing new states. The environment also contains a reward function which takes in the current state and action. This reward function assigns numerical values, known as rewards, which the agent aims to maximise as it interacts with the environment. At time-step t, let S_t represent the current state and A_t the action the agent takes in that state. In the next time-step $t+1$, the agent receives a reward $r(S_t, A_t)$ as a result of its action in time-step t and finds itself in a new state S_{t+1}, where it then selects its next action A_{t+1}. This cycle of observation, action, and feedback repeats throughout the agent's interaction with the environment. This sequential process allows us to define the trajectory up to time-step t, τ_t, as

$$\tau_t = \{s_i, a_i, r(s_i, a_i)\}_{i=0}^{t}.$$

The trajectory, τ_t, is all the agent-environment interactions up to and including the interactions at time-step t. Throughout this work, capital letters represent random variables whilst lowercase letters indicate their outcomes. Figure 2.1 illustrates the agent-environment interaction.

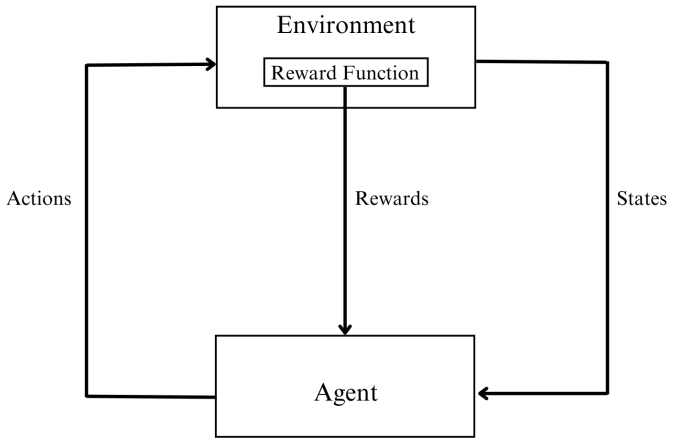


Figure 2.1: An illustration of an MDP. Adapted from [2].

Next we will define an MDP more formally. A MDP consist of the following tuple, $(\mathcal{S}, \mathcal{A}, P, r, \gamma)$. We now describe each element of this tuple in more detail:

- The state space is denoted by $\mathcal{S}$. It is the set of all possible states that an agent can occupy within the environment.
- The action space is denoted by $\mathcal{A}$. This is the set of all possible actions available to the agent in the environment. We assume that the available actions do not vary by state.
- The transition probability function is denoted by $P : \mathcal{S} \times \mathcal{A} \to \Delta(\mathcal{S})$. Here $\Delta(S)$ denotes the probability distribution over $\mathcal{S}$. This function $P(s'|s, a)$ denotes the probability of transitioning to state s' given the previous state, s and action a. It is important to note that the probability of s' only depends on s and a and not the states and actions that appear before them. This is a restriction not on the action but rather the state. The state should contain all the information needed to make a decision. In other words, if we know the transition probability function then we only need the current state to predict the next state and expected next reward. This is known as the Markov property [17].
- The reward function is denoted by $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$. The reward function $r(s, a)$ gives the immediate reward for taking action a in state s. The reward function can output a random variable whose distribution depends on a and s. In our case we assume that the reward function is deterministic.
- The discount rate or discount factor is denoted by $\gamma \in [0, 1]$. This discount factor determines how much the agent values future rewards compared to

the immediate ones. In other words, a reward received k time-steps in the future is weighted by γ^k, meaning its contribution diminishes with each additional time-step into the future. The closer γ is to zero, the more myopic the agent is, i.e., the agent primarily considers only the immediate reward and largely ignores future rewards. On the other hand, as γ approaches 1, the agent becomes more far-sighted, with future rewards being valued almost as much as immediate ones.

In particular we focus on finite MDPs where $\mathcal{S}$ and $\mathcal{A}$ are finite.

Returns, Policies and Value Functions

Usually in RL, the agent-environment interaction breaks into sequences known as episodes. Each episode has a standard starting state and a terminal state in which the episode ends. It is important to note that the next episode begins independently of how the previous episode ended. Each episode has a final time-step T. The final time-step T can vary from episode to episode. It is possible that within an episode future rewards received may not be valued equally to the reward obtained at the current time-step. Therefore, the future rewards can be discounted using the discount factor, γ.

An agent's objective in an episode is to maximise the return. Informally, the return is simply the sum of future rewards possibly discounted. By doing so should yield the desired behaviour. It is important that the reward function truly captures what we wish for the agent to achieve.

With this we can formally define the return as,

$$G_t := \sum_{k=t}^{T} \gamma^{k-t} r(S_k, A_k). \tag{2.1.1}$$

Note that γ indicates how much the future rewards are discounted at the current time-step. Now the returns satisfy an important property, where we can relate G_t to G_{t+1} as follows,

$$\begin{aligned} G_t &= \sum_{k=t}^{T} \gamma^{k-t} r(S_k, A_k) \\ &= r(S_t, A_t) + \gamma r(S_{t+1}, A_{t+1}) + \cdots + \gamma^{T-t} r(S_T, A_T) \\ &= r(S_t, A_t) + \gamma \left(\sum_{k=t+1}^{T} \gamma^{k-t} r(S_k, A_k) \right) \\ &= r(S_t, A_t) + \gamma G_{t+1}. \end{aligned} \tag{2.1.2}$$

In order for the agent to maximise the episode return, it makes use of a value function. A value function tells us the expected return given a state. Value functions are defined with respect to policies. Policies are essentially the agent's

method for optimising its path through the MDP. It is a mapping from states to a probability distribution over the action space and it is of the form:

$$\pi : \mathcal{S} \to \Delta(\mathcal{A}).$$

Hence the action taken at time-step t is sampled from the policy. We can now define value function more formally. The state value function V_π defines the value of state s following policy π and it is given by,

$$V_\pi(s) := \mathbb{E}_\pi \left[G_t | S_t = s \right]. \tag{2.1.3}$$

Similarly we have the state-action value function Q_π which defines the value of selecting action a in state s and thereafter follow policy π:

$$Q_\pi(s, a) := \mathbb{E}_\pi \left[G_t | S_t = s, A_t = a \right]. \tag{2.1.4}$$

The value functions also satisfy a recursive relationship similar to the return as shown in (2.1.2). We start with the state value function,

$$\begin{aligned} V_\pi(s) &:= \mathbb{E}_\pi \left[G_t | S_t = s \right] \\ &= \sum_{a \in \mathcal{A}} \pi(a|s) \sum_{s' \in \mathcal{S}} P(s'|s, a) \left[r(s, a) + \gamma \mathbb{E}_\pi \left[G_{t+1} | S_{t+1} = s' \right] \right] \\ &= \sum_{a \in \mathcal{A}} \pi(a|s) \sum_{s' \in \mathcal{S}} P(s'|s, a) \left[r(s, a) + \gamma V_\pi(s') \right]. \end{aligned} \tag{2.1.5}$$

Equation (2.1.5) is the Bellman equation for the value-function [17]. The above formula shows that the value of the current state is equal to the discounted value of the next state plus the reward. Similarly we can obtain the Bellman equation for the action-value function,

$$\begin{aligned} Q_\pi(s, a) &= \mathbb{E}_\pi \left[G_t | S_t = s, A_t = a \right] \\ &= \sum_{s' \in \mathcal{S}} P(s'|s, a) \left[r(s, a) + \gamma Q_\pi(s', a') \right]. \end{aligned} \tag{2.1.6}$$

Value functions allow us to compare different policies. They enable us to determine whether policy π is better than policy $\hat{\pi}$. More concretely, a policy π is considered better than or equal to policy $\hat{\pi}$ if and only if $V_\pi(s) \geq V_{\hat{\pi}}(s)$ for all possible states in the state space. An important property of finite MDPs is that there exists a policy that is at least as good as, if not better than, all other policies. This is known as the optimal policy, denoted as π^*. Multiple optimal policies may exist, but all optimal policies share the same state-value function, V^*, and action-value function, Q^*. These are defined as follows:

$$V^*(s) := \max_\pi V_\pi(s), \tag{2.1.7}$$

and

$$Q^*(s, a) := \max_\pi Q_\pi(s, a), \tag{2.1.8}$$

for all possible states in the state space. We can also write the value functions Q_π and V_π in terms of each other. We express V_π in terms of Q_π as,

$$V_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) Q_\pi(s, a). \tag{2.1.9}$$

The value function $V_\pi(s)$ is the weighted sum of the action-value function $Q_\pi(s, a)$, where the weights are the probabilities of taking each action a in state s according to the policy π. The action-value function, $Q_\pi(s, a)$, in terms of $V_\pi(s)$ is,

$$Q_\pi(s, a) = r(s, a) + \sum_{s' \in \mathcal{S}} P(s'|s, a) \gamma V_\pi(s'). \tag{2.1.10}$$

We can also do the same for optimal value functions. The optimal state-value function in terms of the optimal action-value function is,

$$V^*(s) = \max_{a \in \mathcal{A}} Q^*(s, a). \tag{2.1.11}$$

And the optimal action-value function in terms of the optimal state-value function is,

$$Q^*(s, a) = r(s, a) + \sum_{s' \in \mathcal{S}} P(s'|s, a) \gamma V^*(s'). \tag{2.1.12}$$

2.1.2 Temporal-Difference Learning

> *"If one had to identify one idea as central and novel to reinforcement learning, it would undoubtedly be temporal-difference (TD) learning."*
>
> — *Sutton and Barto, Reinforcement Learning: An Introduction*

Let us consider a situation where our RL agent is given an MDP with unknown state-transition functions. The agent must solve the MDP based solely on its own experiences in the environment.

Temporal-Difference (TD) learning algorithms are specifically designed for this scenario and are categorised as model-free methods. This means they do not require knowledge of the state-transition function to evaluate the policy's state-value function or to improve it. TD learning allows agents to learn directly from raw experience, making it highly practical in situations where the agent cannot access a model of the environment.

Another important property of TD methods is that they can learn from incomplete episodes. This makes them suitable for environments where episodes may not have a clear ending or are non-episodic. In this subsection, we focus on how we can estimate or evaluate our value function and then improve upon our policy.

Temporal-Difference Prediction

Given that our agent is interacting with an environment where the state-transition function is unknown, we now explore how it can evaluate the policy it is currently following. Suppose the agent is following a policy π, and let V represent our current estimate of the true value function V_π. The simplest TD update rule for V is given by:

$$V(S_t) \leftarrow V(S_t) + \alpha \left[r(S_t, A_t) + \gamma V(S_{t+1}) - V(S_t) \right]. \tag{2.1.13}$$

In this equation, the term $r(S_t, A_t) + \gamma V(S_{t+1})$ represents the new estimate of the value of state S_t, or the return for the agent after taking action A_t. The parameter α is the learning rate, a constant that determines how much we adjust our estimate of $V(S_t)$ based on new information. The difference between the new estimate and the old estimate of $V(S_t)$ is known as the TD-error, which is defined as:

$$\delta_t := r(S_t, A_t) + \gamma V(S_{t+1}) - V(S_t). \tag{2.1.14}$$

The TD-error measures how much the agent's prediction of the value of S_t changes after moving to state S_{t+1} and observing the immediate reward $r(S_t, A_t)$. This error only becomes available once the agent moves to the next state S_{t+1}.

From equation (2.1.13), we see that the value function is updated immediately after the agent takes action A_t and transitions to the next state. This immediate update process, based on new estimations of the value function, is referred to as bootstrapping. Bootstrapping allows the agent to make continuous updates to its value estimates without having to wait until the end of an episode or task.

This ability to update the value function after every step, based on partial observations of the environment, makes TD methods highly efficient in practice. TD learning enables the agent to learn as it interacts with the environment, making it particularly useful in situations where it is not feasible to wait until the episode ends to learn.

Temporal Difference Control: Q-learning

Now that we have seen how to improve estimate of the state-value function we move onto how one can improve their policy using TD ideas. The algorithm that we focus on in this subsection is known as Q-learning [28]. Q-learning has a similar update rule to (2.1.13) except with the estimated action-value function Q,

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[r(S_t, A_t) + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right]. \tag{2.1.15}$$

Another difference is that in the update formula we update the Q value using the greedy action, i.e., the action that maximises the Q in state S_t. We are estimating the return of a greedy policy despite us not following that greedy

policy. This is known as off-policy learning where we are learning about some policy π while following some other policy π'. Let π be our target policy and π' be our behaviour policy, the policy we are actually following. Q-learning allows us to improve both the behaviour policy and the target policy. This allows us to learn the optimal policy while following some exploratory policy. This is what makes Q-learning so powerful. We are able to estimate the optimal action-value function, Q^*, independent of the behaviour policy. Algorithm 1 presents the Q-learning algorithm, taken from [17, p. 132],

Algorithm 1 Q-learning from [17]

Require: step-size $\alpha \in (0, 1]$ and an MDP
Ensure: The optimal policy π^*
1: Initialise Q for all states and actions in state and action spaces respectively. Ensure that Q value for the terminal state is zero.
2: **for** each episode **do**
3: Initialise S, the state
4: **repeat**
5: Pick a from s using policy derived from Q
6: Execute a and observe the reward $r(s, a)$ and the next state s'
7: $Q(s, a) \leftarrow Q(s, a) + \alpha \left[r(s, a) + \gamma \max_{a'} Q(s', a') - Q(s, a) \right]$
8: $s \leftarrow s'$
9: **until** s is terminal
10: **end for**

With this, we have covered the essential elements of tabular RL methods. Modern RL algorithms make use of deep learning to handle large state spaces. The algorithms in this project are examples of modern RL algorithms. We next cover modern RL algorithms, including Deep Q-Network (DQN) and Proximal Policy Optimisation (PPO).

2.2 Deep Reinforcement Learning

We are now ready to explore modern RL algorithms. Most of the problems we wish to tackle using RL involve large state-spaces. Using the tabular methods discussed in section 2.1, such as Q-learning, is not ideal. When dealing with large state-spaces, our Q-table, $Q(s, a)$, becomes extremely large. As a result, the memory requirements and the amount of data needed to populate the table are enormous. Another issue is that most of the states we encounter in such problems will likely be states we have not seen before. There's also the issue of dealing with continuous spaces. One could discretise the continuous spaces to apply the Q-learning algorithm, but binning the continuous values can lead to loss of precision.

As an example, consider the cart-pole environment described by Barto et al.

[29]. A pole is fixed to a cart travelling on a frictionless track. The objective is to maintain the pole's balance by applying forces in the left or right direction onto the cart. Figure 2.2 shows a screenshot of the environment.

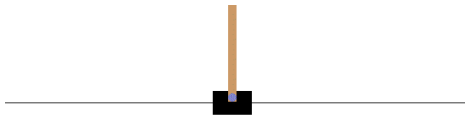


Figure 2.2: A screenshot of the cart-pole environment. Taken from [1].

The action-space is discrete; however, the state-space is continuous. The state-space includes the cart position, the velocity of the cart, the pole angle and the pole angular velocity. It would be beneficial to work with the continuous state-space directly. Deep learning (DL) involves approximating functions using neural networks in high-dimensional problems, e.g., problems with large state-spaces. Tabular RL methods can no longer find exact solutions for such problems. Hence, deep reinforcement learning (Deep-RL) involves combining Deep Learning and Reinforcement Learning.

Recall that in RL, we have a policy and a value function. In Deep-RL, neural networks are used to approximate the policy and value functions in the hopes of achieving generalisation. This combination gives rise to three main types of algorithms in Deep-RL: value-based methods, policy-based methods, and actor-critic methods (a combination of the first two).

2.2.1 Value-Based Methods: Deep Q Network

> *"The potential for off-policy learning remains tantalizing, the best way to achieve it still a mystery."*
>
> — *Sutton and Barto, Reinforcement Learning: An Introduction*

The Deadly Triad

Value-based TD methods, such as Q-learning, involve the agent learning either the state-value function or the action-value function to generate a policy. In this subsection, we will take a look at the Deep Q-Network (DQN) [26]. In simple terms, DQN is an algorithm where we approximate the action-value function, $Q(s, a)$, with a neural network.

However, by doing this, there is no guarantee that our policy will converge and it could cause the parameters of the function to diverge. There are three reasons for this: function approximation, bootstrapping, and off-policy learning [17]. Sutton and Barto [17] note that an RL algorithm could diverge if all three features are present in an algorithm, as they are in DQN. These reasons are collectively known as the deadly triad.

Let us now see why these three features are known as the deadly triad. Function approximation, such as neural networks, uses parameter sharing to output values of all the states. Therefore, when we update the value of one state, we update the parameters. This, however, changes the values of other states as the parameter used to find the value of one state is used to determine the value of another state. This is due to generalisation from function approximation [30]. This parameter sharing can lead to divergence as these changes accumulate.

Bootstrapping is when we update our target Q-function with existing estimates. This can lead to errors from previously calculated values carrying over as we update our Q-function. In off-policy learning, we have a behaviour policy and a target policy, the policy we are trying to optimise. Training is done on data generated by the behaviour policy; however, if the difference between the target policy and the behaviour policy is too large, the algorithm may diverge.

To illustrate how divergence can occur if we have the deadly triad, let us consider a simple example. This example is taken from Tsitsiklis and Van Roy [31]. Assume we have two states: s_1 and s_2, described by a simple scalar feature ϕ. We have that $\phi(s_1) = 1$ and $\phi(s_2) = 2$. In state s_1, we have only one action available, action a. In s_1, once we have taken action a, the state will deterministically transition to state s_2 with a reward of 0. Since we have only one action in state s_1, the state-value function and the action-value function are the same.

Therefore, we will use the state-action value function in this example. We will also use linear function approximation for the state-value function V, which has parameters θ. The value function V is given by,

$$V(s;\theta) = \phi(s) \cdot \theta.$$

Recall equation (2.1.15), the update equation in Q-learning,

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[r(S_t, A_t) + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right].$$

Therefore, we define the loss function as $Q_L(\theta)$,

$$Q_L(\theta) = \frac{1}{2} \left[r(s_1, a) + \gamma V(s_2;\theta) - V(s_1;\theta) \right]^2 \tag{2.2.1}$$

The gradient of the loss with respect to θ is,

$$\nabla_\theta Q_L = \left[r(s_1, a) + \gamma V(s_2;\theta) - V(s_1;\theta) \right] \cdot \nabla_\theta V(s_1;\theta).$$

Substituting $V(s_1;\theta) = \phi(s_1)\cdot\theta = \theta$, $V(s_2;\theta) = \phi(s_2)\cdot\theta = 2\theta$, and $r(s_1, a) = 0$, we obtain,

$$\nabla_\theta Q_L = \gamma 2\theta - \theta. \tag{2.2.2}$$

When we update our parameter θ, we use the following formula:

$$\theta \leftarrow \theta + \alpha \nabla_\theta Q_L, \tag{2.2.3}$$

which in this case is,

$$\theta \leftarrow \theta + \alpha \left[\gamma 2\theta - \theta\right]. \tag{2.2.4}$$

Assume that the initial value is $\theta = 10$. Then the value of s_1 will be 10, and the value of s_2 will be 20, so the transition to s_2 will look good. However, if γ is near one, e.g., 0.999, then our parameter θ will diverge and continue to grow as we perform more updates. Figure 2.3 illustrates this example.

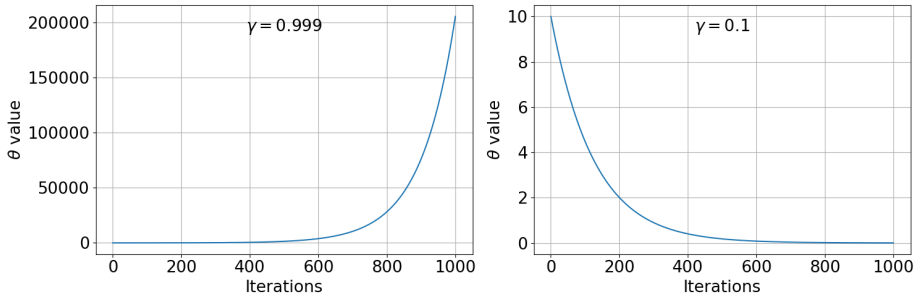


Figure 2.3: Illustration of parameter divergence in the presence of the deadly triad. The plots show the evolution of θ over 1,000 iterations using equation (2.2.4), with initial $\theta = 10$ and learning rate $\alpha = 0.01$. Left: With discount factor $\gamma = 0.999$, θ diverges exponentially. Right: With $\gamma = 0.1$, θ converges to zero.

Now recall that DQN has all three members of the deadly triad, but in practice, it seems to perform well. This is because of two features of the algorithm: experience replay and the target Q network which we discuss next.

Solutions to the Deadly Triad

The first feature that helps DQN perform well is known as experience replay. In RL, an agent's actions and the current environment state influence which future states it will encounter, making the states and rewards at each time-step dependent on what happened in previous time-steps. Experience replay simply stores past experiences in a memory buffer and selects experiences at random for training. It stores the n most recent experiences the RL agent has gathered. Once the memory is full, it discards the oldest experience. Experience replay improves data efficiency, as every transition stored in memory can be used multiple times when updating the network. Since the experiences are selected at random, this reduces the correlation between experiences when they

are sampled. This is needed as gradient descent optimisation techniques generally perform better and have stronger convergence guarantees when training data is independent and identically distributed [32, 33]. This also avoids the situation where the agent trains too much on a certain scenario, which could lead to poor exploration of its environment [34, p. 77].

The second feature that helps with the deadly triad is the periodically updated target network, $\hat{Q}$. Our Q function, which we update, is optimised based on the target values output by $\hat{Q}$. If we update $\hat{Q}$ at every time-step, we end up with a situation where the Q network is chasing a moving target. This can cause the algorithm to diverge. The target network, $\hat{Q}$, has the same architecture as our Q network, and we update it every C steps.

With all the elements presented, we now introduce the DQN algorithm [26].

Algorithm 2 Deep Q-Network (DQN) Algorithm taken from [26]

Require: Replay memory size N, minibatch size, step-size $\alpha \in (0, 1]$, discount factor γ, target network update frequency C, and an environment.
Ensure: The optimal policy π^*

Initialise replay memory D to capacity N
Initialise action-value function Q with random weights θ
Initialise target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
for episode = 1 to M **do**
    Initialise the environment and observe the initial state s_1
    for $t = 1$ to T **do**
        Select an action a_t based on $Q(s_t, a; \theta)$
        Take action a_t and observe reward r_t and next state s_{t+1}
        Store transition (s_t, a_t, r_t, s_{t+1}) in D
        Sample random minibatch of transitions from D
        Set $y_j \leftarrow \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(s_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(s_j, a_j; \theta))^2$ with respect to θ
        if $t \mod C == 0$ **then**
            Update target network: $\theta^- \leftarrow \theta$
        end if
    end for
end for
return θ

2.2.2 Policy-Based Methods

We now tackle methods known as policy gradient methods. These are methods where the policy does not consult the action-value when deciding actions. For

example Q-learning and DQN involves looking at the estimated action value to select an action.

In policy gradient methods, the policy is parametrised and the agent is defined by a policy π_θ, parametrised by θ, which picks allows us to sample actions. The policy is,

$$\pi_\theta(a|s,\theta) = \Pr\{\mathrm{A_t} = \mathrm{a}|\mathrm{S_t} = \mathrm{s}, \theta_\mathrm{t} = \theta\}.$$

The policy is never deterministic so that exploration can take place, i.e., $\pi_\theta \in (0,1)$ for all states and actions. To learn the policy we use the gradient of a scalar objective $J(\theta)$. This scalar objective is given by,

$$J(\theta) := \mathbb{E}_\theta \left[G_0|s_0\right].$$

The gradient of $J(\theta)$ are with respect to the policy's parameters. For policy-based methods we make use of the Policy Gradient Theorem (see Appendix A.1 for more details) to find the gradient of $J(\theta)$. From the Policy Gradient Thereom the gradient of the objective is,

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T} \gamma^t Q_\pi(S_t, A_t) \nabla_\theta \log \pi_\theta(A_t|S_t)|s_0.\right] \tag{2.2.5}$$

Policy parametrisation may take any form provided it remains differentiable with respect to the policy parameters. In this subsection we focus on one policy gradient method known as REINFORCE [35].

REINFORCE

REINFORCE [35] is the first algorithm we will examine that implements the Policy Gradient Theorem. It makes use of Monte Carlo sampling to estimate the state-action value function. We essentially sample trajectories and average their returns to estimate Q_π, therefore in practice we use the return G_t as an unbiased sample-based estimator of Q_π. Algorithm 3 is the REINFORCE algorithm taken from [32].

Algorithm 3 REINFORCE Algorithm taken from [32]

Require: Learning rate α, maximum number of episodes $MAX\_EPISODE$, the discount factor $\gamma \in (0, 1]$,and a policy network π_θ with initial weights θ.
Ensure: Optimised policy π^*.

for episode = 0 to $MAX\_EPISODE$ **do**
  Sample a trajectory $\tau_T = \{s_0, a_0, r(s_0, a_0), \ldots, s_T, a_T, r(s_T, a_T)\}$ by following π_θ.
  Set $\nabla_\theta J(\theta) \leftarrow 0$
  for $t = 0$ to T **do**
    Compute return using:

$$G_t := \sum_{k=t}^{T} \gamma^{k-t} r(s_k, a_k)$$

    Note: G_t serves as an unbiased Monte Carlo estimate of $Q_\pi(s_t, a_t)$
    Compute gradient estimate using:

$$\nabla_\theta J(\theta) \leftarrow \nabla_\theta J(\theta) + \gamma^t G_t \nabla_\theta \log \pi_\theta(a_t|s_t)$$

  end for
  Update policy parameters: $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$
end for
return θ

2.2.3 Actor-Critic Methods

Actor-Critic Methods combine both policy-based and value-based approaches. We have an actor that learns the parametrised policy, π_θ, and a critic that learns the state-value function, parametrised by ϕ. In this subsection, we first discuss why the need to learn the state-value function arises in policy gradient methods by taking a look at the advantage function. Next, we cover the Advantage Actor-Critic (A2C) method. Finally, we conclude with Proximal Policy Optimisation (PPO), which improves the stability of training by ensuring that the new policy does not deviate significantly from the old policy.

The Advantage Function

REINFORCE has high variance in its gradient estimates since it requires a full trajectory for gradient estimation. Furthermore, because it depends on generating full trajectories, REINFORCE exhibits poor sample efficiency compared to value-based methods such as DQN. However, one can reduce the variance by introducing a baseline $b(s)$ such that:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T-1} \gamma^t \left(Q_\pi(S_t, A_t) - b(S_t)\right) \nabla_\theta \left(\log \pi_\theta(A_t|S_t)\right) \middle| s_0\right].$$

Subtracting the baseline reduces the variance while leaving the expectation with respect to the action unaffected, i.e., no bias is introduced to the gradient estimate. For this to hold, b must not depend on the action, although it may depend on the states.

Recall that $\mathbb{E}_{\pi_\theta}$ represents an expectation over all possible trajectories τ_t sampled from the policy π_θ. However, this expectation is actually composed of two expectations. For instance:

$$\mathbb{E}_{\pi_\theta}\left[Q(s,a)\right] = \int_s d_\pi(s) \int_a \pi_\theta(a|s) Q_\pi(s,a)\, da\, ds.$$

Where $d_\pi(s)$ is the state distribution under policy π and is stationary. Following the approach by Lehmann [36], we now demonstrate that if b satisfies this condition, no bias is introduced to the gradient estimate:

$$\begin{aligned}
\mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T-1} \gamma^t \left(b(S_t)\right) \nabla_\theta \left(\log \pi_\theta(A_t|S_t)\right) \middle| s_0\right] &= \sum_{t=0}^{T-1} \gamma^t \int_s d_\pi(s) b(s) \int_a \pi_\theta(a|s) \nabla_\theta \log \pi_\theta(a|s)\, da\, ds \\
&= \sum_{t=0}^{T-1} \gamma^t \int_s d_\pi(s) b(s) \int_a \pi_\theta(a|s) \frac{\nabla_\theta \pi_\theta(a|s)}{\pi_\theta(a|s)}\, da\, ds \\
&= \sum_{t=0}^{T-1} \gamma^t \int_s d_\pi(s) b(s) \int_a \nabla_\theta \pi_\theta(a|s)\, da\, ds \\
&= \sum_{t=0}^{T-1} \gamma^t \int_s d_\pi(s) b(s) \nabla_\theta \int_a \pi_\theta(a|s)\, da\, ds \\
&= \sum_{t=0}^{T-1} \gamma^t \int_s d_\pi(s) b(s) \nabla_\theta 1\, ds \\
&= 0.
\end{aligned}$$

What is a good choice for the baseline? The variance of a random variable X is given by:

$$\mathrm{Var}[X] = \mathbb{E}[X^2] - \mathbb{E}[X]^2.$$

The term $\mathbb{E}[X]^2$ in the above equation is independent of our baseline. Therefore, we will focus on the $\mathbb{E}[X^2]$ term. We have:

$$\begin{aligned}
&\arg\min_b \operatorname{Var}\left[\sum_{t=0}^{T-1}\gamma^t\left(Q_\pi(S_t,A_t)-b(S_t)\right)\nabla_\theta\left(\log\pi_\theta(A_t|S_t)\right)\right]\\
&=\arg\min_b \mathbb{E}_{\pi_\theta}\left[\left(\sum_{t=0}^{T-1}\gamma^t\left(Q_\pi(S_t,A_t)-b(S_t)\right)\nabla_\theta\left(\log\pi_\theta(A_t|S_t)\right)\right)^2\right]\\
&=\arg\min_b \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T-1}\gamma^{2t}\left(Q_\pi(S_t,A_t)-b(S_t)\right)^2\cdot\left(\nabla_\theta\left(\log\pi_\theta(A_t|S_t)\right)\right)^2\right].
\end{aligned}$$

In practice, it is often assumed that $(Q_\pi(S_t,A_t)-b(S_t))$ and $\nabla_\theta\left(\log\pi_\theta(A_t|S_t)\right)$ are independent, or that their dependence is negligible. Therefore:

$$\begin{aligned}
&\arg\min_b \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T-1}\gamma^{2t}\left(Q_\pi(S_t,A_t)-b(S_t)\right)^2\cdot\left(\nabla_\theta\left(\log\pi_\theta(A_t|S_t)\right)\right)^2\right]\\
&\approx\arg\min_b \sum_{t=0}^{T-1}\gamma^{2t}\mathbb{E}_{\pi_\theta}\left[\left(Q_\pi(S_t,A_t)-b(S_t)\right)^2\right]\cdot\mathbb{E}_{\pi_\theta}\left[\left(\nabla_\theta\left(\log\pi_\theta(A_t|S_t)\right)\right)^2\right].
\end{aligned}$$

Thus, we should minimise the term:

$$\mathbb{E}_{\pi_\theta}\left[\left(Q_\pi(S_t,A_t)-b(S_t)\right)^2\right].$$

To minimise the above equation, we use the following result [36]:

$$\min_a \mathbb{E}\left[(X-a)^2\right]=\mathbb{E}[X].$$

This implies that $b(s)=\mathbb{E}_{\pi_\theta}\left[Q_\pi(s,A)\right]=V_\pi(s)$. With this, we are now ready to define the advantage function, $A_\pi(s,a)$, as:

$$A_\pi(s,a):=Q_\pi(s,a)-V_\pi(s). \tag{2.2.6}$$

The advantage function yields a low variance, even with our assumption of independence [37]. The advantage function indicates that the action a is better than the average action when $A_\pi(s,a)>0$, and indicates that the action is worse than average when $A_\pi(s,a)<0$. This allows us to adjust the policy by increasing the probability of actions that are better than average and decreasing the probability of actions that are worse than average. We can therefore rewrite equation (2.2.5) as:

$$\nabla_\theta J(\theta)=\mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T}\gamma^t A_\pi(S_t,A_t)\nabla_\theta\log\pi_\theta(A_t|S_t)\middle|s_0\right]. \tag{2.2.7}$$

Generalised Advantage Estimation (GAE)

Actor-Critic methods make use of equation (2.2.7) to calculate the gradient of the objective. In order to obtain this gradient, we need to estimate the advantage function, which requires learning the state-value function. One popular method for estimating the advantage function is Generalised Advantage Estimation (GAE) [37].

Recall that in Q-learning, we focused on one-step estimates of the Q function. We can adopt a similar approach for the advantage function:

$$A_t^{(1)} = r(s_t, a_t) + \gamma V_\pi(s_{t+1}) - V_\pi(s_t).$$

However, we are not restricted to just one-step estimates; in general, we can use n-step estimates of the advantage function:

$$A_t^{(n)} = r(s_t, a_t) + \gamma r(s_{t+1}, a_{t+1}) + \cdots + \gamma^{n-1} r(s_{t+n-1}, a_{t+n-1}) + \gamma^n V_\pi(s_{t+n}) - V_\pi(s_t).$$

As $n \to \infty$, our estimate of the advantage function becomes unbiased but exhibits high variance. Conversely, as $n \to 1$, the estimate becomes more biased but has lower variance [36]. Instead of choosing a single value of n, we can take an exponentially weighted average of all n-step estimates of the advantage.

First, let:

$$\delta_t = A_t^{(1)} = r(s_t, a_t) + \gamma V_\pi(s_{t+1}) - V_\pi(s_t).$$

Next, observe that:

$$\begin{aligned} A_t^{(2)} &= r_t + \gamma r_{t+1} + \gamma^2 V(s_{t+2}) - V(s_t) \\ &= r_t + \gamma r_{t+1} + \gamma^2 V(s_{t+2}) - V(s_t) + \gamma V(s_{t+1}) - \gamma V(s_{t+1}) \\ &= \big(r_t + \gamma V(s_{t+1}) - V(s_t)\big) + \gamma\big(r_{t+1} + \gamma V(s_{t+2}) - V(s_{t+1})\big) \\ &= A_t^{(1)} + \gamma A_{t+1}^{(1)}. \end{aligned}$$

We can generalise this to say:

$$A_t^{(n)} = A_t^{(1)} + \gamma A_{t+1}^{(n-1)} = \delta_t + \gamma \delta_{t+1}. \tag{2.2.8}$$

This allows us to derive a simple expression for GAE, the exponentially weighted average:

$$A_{GAE}^\pi(s_t, a_t) = \sum_{l=0}^{T-1} (\gamma\lambda)^l \delta_{t+l}. \tag{2.2.9}$$

Here, λ is the exponential decay coefficient, with $\lambda \in [0, 1]$. The closer λ is to 0, the more biased the estimate becomes, while the closer λ is to 1, the higher

the variance of the estimate. The variable λ allows us to control how much variance and bias we want in our estimates. The advantage with the GAE is that it allows contributions from both high-variance estimates and low-variance estimates.

Advantage Actor-Critic (A2C) Method

Advantage Actor-Critic (A2C) is a foundational actor-critic algorithm. The A2C method makes use of entropy regularisation, which is commonly used in actor-critic methods. The purpose of entropy regularisation is to encourage exploration. Higher policy entropy is associated with more uniform policy distributions [32]. If the policy is more uniform, the actions it selects will be more diverse. The entropy term is added to the loss function of the RL agent. A constant $\beta > 0$ determines the weight assigned to this regularisation term. The algorithm uses separate policy (actor) and value (critic) networks. The critic evaluates the current policy, and the actor updates it. It improves the policy using estimated advantage values. The complete A2C algorithm, as described in [32], is provided in Appendix B.2.

2.2.4 Proximal Policy Optimisation (PPO)

One of the main issues with A2C is that it suffers from performance collapse. The gradient of the objective tells us which direction we should move in order to update our parameters. However, this is only local. If we step too far in a direction, i.e., if the step size is too large, we could end up with a suboptimal policy.

This is not ideal, as the data we now use to update our parameters comes from a suboptimal policy, resulting in poor data. If the new policy is very different from the old policy, this can destabilise training and lead to performance collapse. Unlike supervised learning, the data and the performance are dependent on each other.

Now, if we examine how A2C works, we: 1) estimate the advantage of our current policy, and 2) use that advantage to derive an improved policy. This somewhat resembles the steps we perform for Generalised Policy Iteration Algorithm (see Appendix B.1 for more information): 1) evaluate the value function, and 2) use that value function to update our policy.

However, in GPI, we are guaranteed that our new policy will match or exceed the performance of our current policy, thanks to Theorem B.1.2. We aim to achieve something similar for Actor-Critic methods by ensuring that the new policy does not deviate significantly from the old policy, and by having some assurance that the new policy will at least match the performance of the old policy.

Proximal Policy Optimisation (PPO) [25] achieves this. To see how, we first

define the PPO objective, demonstrate how it enforces stability, and conclude by examining the algorithm in detail.

The Surrogate Clipped Objective

In order to see whether the new policy, π', we obtain after the update is better than our current policy π, we need to change our objective. Assume that π has parameters θ and that π' has parameters θ'. Now let us take a look at the following expression:

$$J(\theta') - J(\theta).$$

If we have $J(\theta') > J(\theta)$, then this indicates that our new policy π' is better than our old policy π. This expression will help us obtain monotonic performance improvement during optimisation. Now let us see what this expression is equal to:

$$J(\theta') - J(\theta) = \mathbb{E}_{\pi'_{\theta'}} \left[\sum_{t=0}^{T} \gamma^t A^{\pi}(s_t, a_t) \right]. \tag{2.2.10}$$

We will now show that equation (2.2.10) is true following the approach by Graesser and Keng [32]:

$$\begin{aligned}
&\mathbb{E}_{\pi'_{\theta'}} \left[\sum_{t=0}^{T} \gamma^t A_{\pi_\theta}(s_t, a_t) \right] \\
&\quad = \mathbb{E}_{\pi'_{\theta'}} \left[\sum_{t=0}^{T} \gamma^t ([r(s_t, a_t) + \gamma V_{\pi}(s_{t+1})] - V_{\pi}(s_t)) \right] \quad \text{(the definition of the advantage function)} \\
&\quad = \mathbb{E}_{\pi'_{\theta'}} \left[\sum_{t=0}^{T} \gamma^t r(s_t, a_t) + \sum_{t=0}^{T} \gamma^{t+1} V_{\pi}(s_{t+1}) - \sum_{t=0}^{T} \gamma^t V_{\pi}(s_t) \right] \\
&\quad = \mathbb{E}_{\pi'_{\theta'}} \left[\sum_{t=0}^{T} \gamma^t r(s_t, a_t) \right] + \mathbb{E}_{\pi'_{\theta'}} \left[\sum_{t=0}^{T} \gamma^{t+1} V_{\pi}(s_t) - \sum_{t=0}^{T} \gamma^t V_{\pi}(s_t) \right] \\
&\quad = J(\theta') + \mathbb{E}_{\pi'_{\theta'}} \left[\sum_{t=0}^{T} \gamma^{t+1} V_{\pi}(s_{t+1}) - \sum_{t=0}^{T} \gamma^t V_{\pi}(s_t) \right] \\
&\quad = J(\theta') - \mathbb{E}_{\pi'_{\theta'}} \left[V_{\pi}(s_0) \right] \\
&\quad = J(\theta') - J(\theta).
\end{aligned}$$

One issue with equation (2.2.10) is that it requires us to take trajectories from the new policy, π'. However, the new policy is only available after we perform the update. To deal with this, we make use of Importance Sampling. Importance Sampling is when we estimate an unknown distribution using samples drawn from a known distribution [32]. Using Importance Sampling, we obtain the

surrogate objective::

$$J_{\pi}^{\text{CPI}}(\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T}\gamma^t A_{\pi_\theta}(s_t,a_t)\frac{\pi'_{\theta'}(a_t|s_t)}{\pi_\theta(a_t|s_t)}\right] \tag{2.2.11}$$
$$\approx J(\theta') - J(\theta) = \mathbb{E}_{\pi'_{\theta'}}\left[\sum_{t=0}^{T}\gamma^t A^{\pi}(s_t,a_t)\right].$$

The superscript CPI stands for conservative policy iteration. However, $J_{\pi}^{\text{CPI}} \approx J(\theta') - J(\theta)$ only if π' is close to π. One can use Total Variation to measure the difference between the old policy and the new policy; however, it is difficult to work with in practice. It is computationally expensive. The KL divergence is a more popular choice, as it has tractable expressions, to measure how different and how similar two distributions are. The KL divergence allows us to place a constraint. We therefore have the following optimisation problem:

$$\max_{\theta} J_{\pi}^{\text{CPI}}(\theta) \tag{2.2.12}$$
$$\text{subject to } \mathbb{E}_{\pi_\theta}\sum_{t=0}^{T}\left[KL(\pi'_{\theta'}(a_t|s_t)\,||\,\pi_\theta(s_t,a_t))\right] \leq \epsilon.$$

Only policies in the neighbourhood around policy π are considered. This neighbourhood is called the trust region [32]. The parameter ϵ limits how large the KL divergence can be and must be tuned when performing experiments.

One issue with the constraint is that the KL divergence can be difficult to work with on complex problems. In practice, one often uses the Taylor Expansion of the KL divergence, and this leads to us calculating second-order derivatives, which can be computationally expensive. If we take a look at the ratio between the old policy and the new policy, one can see that if $\pi' = \pi$, then the ratio is one. Once we start to deviate from the old policy, the ratio changes; it can either be greater than one or less than one. For instance, if the new policy assigns a higher probability to an action than the old policy did, the ratio will be greater than one; if it assigns a lower probability, the ratio will be less than one. Perhaps we can do something simpler to measure how different the two policies are by simply looking at the ratio between the old policy and the new policy.

Proximal Policy Optimisation [25] (PPO) does this by using a simpler objective and takes the expectation over a single time-step. This objective is known as the surrogate clipped objective and is given by equation (2.2.13):

$$J^{\text{CLIP}} = \mathbb{E}_t\left[\min\left(\frac{\pi'_{\theta'}(a_t|s_t)}{\pi_\theta(a_t|s_t)}A_t,\ \text{clip}\left(\frac{\pi'_{\theta'}(\mathrm{a_t}|\mathrm{s_t})}{\pi_\theta(\mathrm{a_t}|\mathrm{s_t})}, 1-\epsilon, 1+\epsilon\right)\mathrm{A_t}\right)\right]. \tag{2.2.13}$$

The hyperparameter ϵ defines the clipping neighbourhood. This neighbourhood is,

$$\left|\frac{\pi'_{\theta'}(a_t|s_t)}{\pi_\theta(a_t|s_t)} - 1\right| \leq \epsilon.$$

The clipped objective restricts the ratio between the old policy and the new policy, ensuring that the new policy does not deviate too much from the old one. This restriction also limits how much influence a single state-action pair can have when updating the policy's parameters. Notice how we take the minimum between the original value of the ratio multiplied by the advantage and the clipped version. This approach is often described as being pessimistic in the policy update. PPO prioritises stable policy updates to avoid overestimating the potential benefit of large policy changes, which could otherwise lead to performance collapse.

The objective also does not involve second-order derivative calculations and is thus easy to implement in practice and more efficient. With this, we can now introduce the PPO algorithm, which prevents performance collapse and serves as our algorithm of choice in this work.

PPO Algorithm

Algorithm 4 is the PPO algorithm taken from [32, p. 177]:

Algorithm 4 Proximal Policy Optimisation (PPO) Algorithm taken from [32, p. 177]

Require: Entropy regularisation weight $\beta \geq 0$, clipping parameter $\epsilon > 0$, number of epochs K, number of actors N, time horizon T, minibatch size $M \leq NT$, actor learning rate $\alpha_A \geq 0$, critic learning rate $\alpha_C \geq 0$, initial parameters θ_A (actor), θ_C (critic).

Ensure: Optimised actor π_{θ_A} and critic V_{θ_C}.

Randomly initialise actor and critic parameters θ_A, θ_C.
Initialise the "old" actor network $\theta_{A_{\text{old}}} \leftarrow \theta_A$.
for $i = 1, 2, \ldots$ **do**
    Set $\theta_{A_{\text{old}}} \leftarrow \theta_A$.
    for each actor $n = 1, 2, \ldots, N$ **do**
        Run policy $\pi_{\theta_{A_{\text{old}}}}$ in the environment for T time-steps and collect trajectories.
        Compute advantages $A_1, \ldots, A_T$ using $\theta_{A_{\text{old}}}$.
        Calculate $V^{\text{target}}_{t,t+1,\ldots,T}$ using θ_C and/or trajectory data.
    end for
    Combine collected trajectories, advantages, and target V-values into a batch of size NT.
    for epoch $= 1, 2, \ldots, K$ **do**
        for minibatch m in batch **do**
            Compute probabilities ratio $r_m(\theta_A)$:

$$r_m(\theta_A) = \frac{\pi_{\theta_A}(a_m|s_m)}{\pi_{\theta_{A_{\text{old}}}}(a_m|s_m)}.$$

            Compute clipped surrogate objective $J^{\text{CLIP}}_m(\theta_A)$:

$$J^{\text{CLIP}}_m(\theta_A) = \mathbb{E}_m\left[\min\left(r_m(\theta_A)A_m,\, \text{clip}(r_m(\theta_A), 1-\epsilon, 1+\epsilon)A_m\right)\right].$$

            Compute entropy regularisation term H_m using actor network θ_A.
            Compute policy loss:

$$\mathcal{L}_{\text{policy}}(\theta_A) = -J^{\text{CLIP}}_m(\theta_A) + \beta H_m.$$

            Compute predicted value $\hat{V}(s_m)$ using critic network θ_C.
            Compute value loss using the MSE loss:

$$\mathcal{L}_{\text{value}}(\theta_C) = \frac{1}{M}\sum_{m=1}^{M}\left(V^{\text{target}}_m - \hat{V}(s_m;\theta_C)\right)^2.$$

            Update actor parameters using gradient descent:

$$\theta_A \leftarrow \theta_A - \alpha_A \nabla_{\theta_A}\mathcal{L}_{\text{policy}}(\theta_A).$$

            Update critic parameters using gradient descent:

$$\theta_C \leftarrow \theta_C - \alpha_C \nabla_{\theta_C}\mathcal{L}_{\text{value}}(\theta_C).$$

        end for
    end for
end for
return θ_A, θ_C

From Algorithm 4, there are key features we should mention. The first is that we can see that PPO uses N actors. In our case, PPO makes use of a vectorised architecture. This means that it takes a step in N independent, identical environments simultaneously.

Another key element is that we do not train the agent based on the number of episodes. This is because episodes can have varying numbers of time-steps. This means that in some environments it can take a long time for an agent to reach a certain objective, which can be computationally expensive.

Having covered the main categories of deep RL algorithms, we now examine a challenge fundamental to all of them: exploration. Exploration is essential for agents to expand their knowledge about the environment and avoid suboptimal policies [18]. Without effective exploration, agents risk converging to local optima and failing to discover better strategies. In the following subsection, we discuss the exploration challenge in RL.

2.2.5 Exploration in RL

RL agents often face a dilemma at every time-step: whether to take the best action given current information (exploitation) or to gather more information about the environment (exploration). This is known as the exploration-exploitation dilemma [17]. The dilemma arises because neither exploration nor exploitation can be pursued excessively without failing at the task.

As an example, assume the RL agent is playing a first-person shooter game on the PlayStation 5 where it is given random weapons but allowed to switch weapons. Exploiting would mean using its current weapon to kill every enemy it encounters. Exploring would mean dropping the weapon and picking up another weapon that could potentially cause more damage, allowing the agent to kill enemies faster and therefore obtain more rewards. Essentially, exploration is needed to escape suboptimal policies and search for more optimal ones. Of course, it is entirely possible that in this situation the weapon the agent switches to is worse than its current weapon. However, exploration can allow the agent to form long-term strategies to maximise rewards in the long run.

The most basic exploration strategy is known as ϵ-greedy. We simply add noise to a greedy policy, i.e., a policy that always picks the action that maximises its value function. For example, Algorithm 2, which shows the DQN algorithm, can be modified such that there is some probability ϵ that the agent takes a random action. This probability can decay as the agent explores more.

Actor-critic methods such as PPO, whereby the output is a probability distribution over the possible actions, make use of entropy regularisation to encourage exploration. At the start of training, one would expect high entropy, where all actions have an equal chance of being selected. But as the agent interacts with

the environment, it should gradually reduce exploration and increase exploitation.

In sparse-reward settings, exploration is particularly critical, as without feedback the agent has nothing to exploit [38]. The issue with exploration methods such as ϵ-greedy and entropy regularisation is that they do not work well in environments with sparse rewards such as Montezuma's Revenge [20]. More effective exploration is needed in such cases. Our next section, intrinsically motivated RL, will cover how one can incorporate intrinsic motivation into RL agents in order to help them explore their environments more efficiently.

2.3 Intrinsically Motivated Reinforcement Learning

Humans are motivated to explore their environment and increase their knowledge due to curiosity. This type of motivation, known as intrinsic motivation, can be formally defined as follows [39, p. 56]:

> "Intrinsic motivation is defined as the doing of an activity for its inherent satisfactions rather than for some separable consequence. When intrinsically motivated, a person is moved to act for the fun or challenge entailed rather than because of external prods, pressures, or rewards."

For example, one might enjoy playing video games with their siblings simply because they find it fun. In this case, the individual is intrinsically motivated.

On the other hand, humans are also motivated by external rewards or outcomes, known as extrinsic motivation. We use the definition provided by Ryan and Deci [39, p. 60]:

> "Extrinsic motivation is a construct that pertains whenever an activity is done in order to attain some separable outcome. Extrinsic motivation thus contrasts with intrinsic motivation, which refers to doing an activity simply for the enjoyment of the activity itself, rather than its instrumental value."

For example, a child might play video games with their sibling to avoid being scolded by their parents for not spending enough time together. In this second example, the individual is motivated by external factors, such as the desire to avoid parental disapproval.

Intrinsically motivated behaviour is essential for intellectual growth, as it enables individuals to expand their knowledge and develop new skills [39]. It is because of this that researchers have attempted to instil intrinsic motivation into RL agents.

2.3.1 The extended view of RL

Figure 2.1 is our standard view of RL, but as indicated by Sutton and Barto [17], this view is misleading. This view implies that the boundary between the agent and environment is the same as the boundary between an organism's body and the outside world. This may not be the case. In RL, anything that the agent cannot change arbitrarily is considered the environment.

For example, if a human were an agent, then one could consider the skeleton and muscles of the individual as part of the environment. This means that in RL, there two types of environments: the internal environment and the external environment. This allows us to expand our standard view of RL.

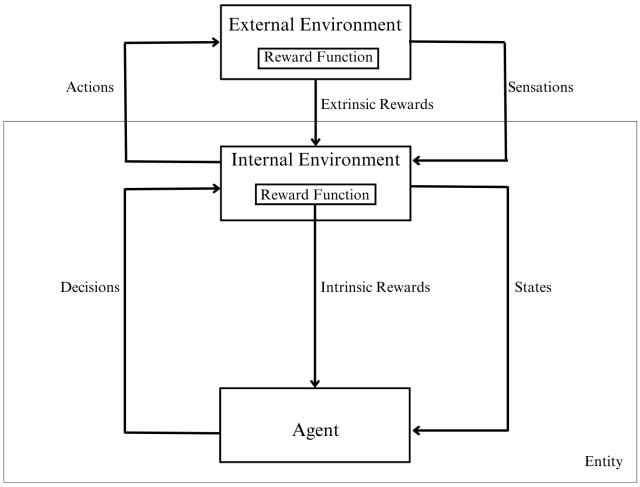


Figure 2.4: An illustration of the extended MDP. Adapted from [3].

Figure 2.4 illustrates the extended view of the agent-environment interaction. Note that we now have a reward function inside the agent's internal environment as well. The internal environment contains the agent's motivational system, which should ideally not be redesigned for different problems. The standard view of RL implies that the rewards the agent obtains are objects or events that encourage returning to the states where those rewards were obtained. However, rewards in RL can be viewed as reward signals.

For humans, these reward signals are produced by neurons in the brain. Furthermore, in Figure 2.4, the agent's outputs are labelled as "Decisions" indicating that the agent's actions, which affect the external environment, are made by the agent's decisions in its internal environment, such as moving a particular muscle. Note that when we refer to the environment, we refer to both the external and the internal environment.

We have two reward streams in RL, one from the internal and one from the external environment. In RL, we define extrinsic rewards as rewards provided by the external environment. For example, the agent obtains a reward of +1 for completing a maze. We define intrinsic rewards as rewards from the internal environment. These are rewards that are not tied to any external goal or reward. For example, the agent receiving an intrinsic reward of +1 for satisfying its curiosity by exploring its environment. We now dive into how one can instil intrinsic motivation into RL agents by using curiosity.

2.3.2 Curiosity-Driven Exploration

In this subsection, we address the question of how one might approach this concept. Jürgen Schmidhuber [40] highlighted that one becomes curious when encountering unknown aspects of their environment. Therefore, curiosity or intrinsic rewards are often linked to predictive errors. For example, when an RL agent's world model predicts the next state s_{t+1} from s_t, the prediction error is the intrinsic reward.

This approach is known as curiosity-driven exploration. The intrinsic rewards diminish as the agent gains more knowledge of its environment. As the world model becomes proficient at predicting what will happen in a specific state, the agent becomes less inclined to visit that state again. We describe this as the agent becoming "bored" of that state, as the intrinsic reward there is low. The agent will only return to that state if the extrinsic rewards are sufficiently high.

To illustrate how curiosity can be implemented in practice, we will now examine three notable algorithms. The remainder of this section on Curiosity focuses on three specific curiosity algorithms: Intrinsic Curiosity Module (ICM) [4], Random Network Distillation (RND) [20] and BYOL-Explore [6].

Intrinsic Curiosity Module

Intrinsic Curiosity Module (ICM) [4] is the first modern curiosity algorithm we will discuss. It has two subsystems: the RL agent and the intrinsic reward generator, known as ICM.

The RL agent's task remains the same as always: to maximise the sum of rewards it obtains. The ICM's task is to output the intrinsic reward. ICM comprises two submodules, both of which are neural networks: the inverse dynamics model, g with parameters θ_I, and the forward dynamics model, f, with parameters θ_F.

Since the intrinsic reward is a prediction error, it is generally preferable to avoid making predictions based on raw observations. This is due to the increased difficulty in identifying which aspects of the environment can be modelled effectively. To illustrate this challenge, consider the following example. An RL agent is attempting to make a stew that requires boiling. The agent observes bubbles

forming during the cooking process. Predicting the exact location of bubbles after they pop is highly challenging and irrelevant for the agent, as this detail does not impact its task. We do not want the agent to become curious about such inconsequential details.

To address this, Pathak et al. [4] divided the agent's observations into three cases: 1) aspects that the agent can control; 2) aspects the agent cannot control but which affect the agent; and 3) aspects the agent cannot control and which do not affect the agent. The formation of bubbles in the earlier example falls into case 3. To ensure that ICM focuses on generating rewards based on cases 1 and 2, it must transform raw observations into a feature representation that emphasises these aspects.

ICM takes observations s_t and s_{t+1} and outputs feature representations $\phi(s_t)$ and $\phi(s_{t+1})$. These feature representations are then inputted into the inverse dynamics model, g, which predicts the action the agent took to transition from $\phi(s_t)$ to $\phi(s_{t+1})$. We aim to find a function g defined as:

$$\hat{a_t} = g(\phi(s_t), \phi(s_{t+1}); \theta_I). \tag{2.3.1}$$

where $\hat{a_t}$ is the predicted action. The parameters θ_I are trained to minimise the loss:

$$\min_{\theta_I} L_I(\hat{a_t}, a_t). \tag{2.3.2}$$

Here, L_I measures the discrepancy between the actual action a_t and the predicted action $\hat{a_t}$. For instance, if the action space were discrete, the loss would be the Negative Log-Likelihood (NLL).

We now turn to the forward dynamics model f. This model takes $\phi(s_t)$ and the actual action a_t as input and predicts the feature representation of the next state, $\hat{\phi}(s_{t+1})$:

$$\hat{\phi}(s_{t+1}) = f(\phi(s_t), a_t; \theta_F). \tag{2.3.3}$$

The forward dynamics model aims to minimise the following loss L_F:

$$L_F(\phi(s_t), \hat{\phi}(s_{t+1})) = \frac{1}{2} \left\| \hat{\phi}(s_{t+1}) - \phi(s_{t+1}) \right\|_2^2. \tag{2.3.4}$$

The intrinsic reward that the agent receives is given by:

$$r_t^i = \nu L_F(\phi(s_t), \hat{\phi}(s_{t+1})), \tag{2.3.5}$$

where ν is a positive scaling factor and r_t^i is the intrinsic reward at time-step t. Figure 2.5, taken from [4], illustrates the ICM procedure.

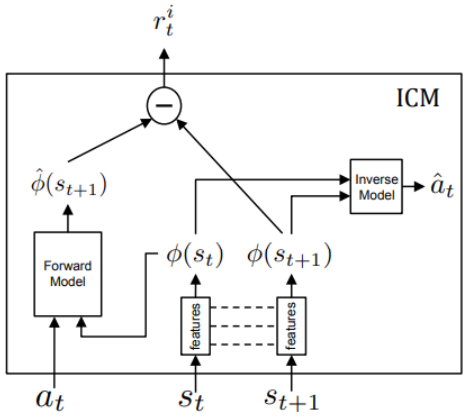


Figure 2.5: An illustration of the ICM procedure. The algorithm encodes states s_t and s_{t+1} into feature representations $\phi(s_t)$ and $\phi(s_{t+1})$, which are fed into the inverse dynamics model to predict action $\hat{a}_t$. The feature representation $\phi(s_t)$ and the actual action a_t are then fed to the forward dynamics model to predict the next state features $\hat{\phi}(s_{t+1})$. The difference between predicted features $\hat{\phi}(s_{t+1})$ and actual features $\phi(s_{t+1})$ generates the intrinsic reward. Taken from [4].

Looking at Figure 2.5, we see that ICM encodes s_t and s_{t+1} into their feature representations $\phi(s_t)$ and $\phi(s_{t+1})$, respectively. These feature representations are then fed into the inverse dynamics model, which predicts the action taken by the agent. The feature representation $\phi(s_t)$ and the action a_t are inputted into the forward dynamics model, which attempts to predict $\phi(s_{t+1})$. The scaled prediction error between $\phi(s_{t+1})$ and $\hat{\phi}(s_{t+1})$ serves as the intrinsic reward. The total reward that the agent receives is:

$$r_t = r_t^e + r_t^i, \tag{2.3.6}$$

where r_t^e is the extrinsic reward. The feature space does not need to encode observations that fall into case 3 (aspects the agent cannot control and that have no effect on the agent), ensuring that the agent does not receive rewards for attempting to predict the unpredictable.

The Noisy-TV Problem

Imagine, for a moment, that we have an agent in a maze attempting to find a particular room. During its path, the agent encounters a room with a noisy TV. This TV can change channels, either due to the agent's actions or randomly, without any intervention from the agent. If the agent is driven by intrinsic curiosity through an Intrinsic Curiosity Module (ICM) acting as its intrinsic reward generator, it is possible for the agent to become overly attracted to the noisy TV and essentially become a "couch potato." The agent continues to receive high intrinsic rewards by attempting to predict the TV's next channel

change. This scenario highlights a significant weakness of curiosity-driven agents based on prediction error: they can become fixated on sources of randomness in their environment.

The issue outlined above is referred to as the Noisy-TV Problem. It describes the tendency of curiosity-driven agents to be drawn towards unpredictable, random elements in their environment. Burda et al. [41, Figure 6] demonstrated that adding sources of randomness to an environment can significantly slow down the learning process for agents using ICM. The Random Network Distillation (RND) method was designed to address this problem.

Random Network Distillation

Random Network Distillation (RND) [20] was designed to overcome the Noisy-TV Problem. Over time, the term Noisy-TV problem has come to encompass any environment with some degree of stochasticity. Traditionally, RND has been combined with Proximal Policy Optimisation (PPO) to address complex exploration challenges, but it can also be used with other deep RL algorithms. In RND, we have two neural networks: a predictor network, $\hat{f}$ with parameters θ, and a randomly initialised neural network called the target network, f, whose parameters, ϕ, stay fixed. The target network's sole purpose is to output a feature representation of a given input state. The predictor network's task is to try and predict the output of the target network, with the prediction error serving as the intrinsic reward to the agent. Importantly, this prediction task is not related to environment dynamics. Figure 2.6 illustrates the RND procedure.

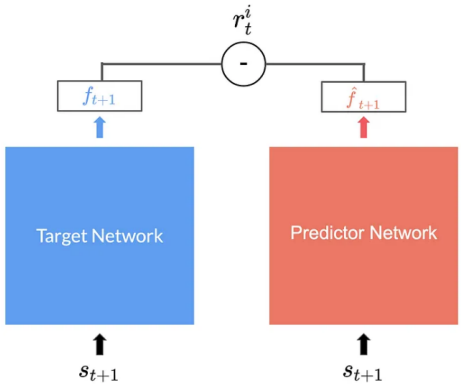


Figure 2.6: An illustration of the RND procedure. The state s_{t+1} is fed into both a fixed Target Network and a trainable Predictor Network. The intrinsic reward is calculated as the squared L2 norm of the difference between their outputs: $r_t^i = \left\| f(s_{t+1}) - \hat{f}(s_{t+1}) \right\|_2^2$. Taken from [5].

Looking at Figure 2.6, we can see that both networks receive state s_{t+1}. The

intrinsic reward, r_t^i, at time-step t is then,

$$r_t^i = \left\| f(s_{t+1}) - \hat{f}(s_{t+1}) \right\|_2^2. \tag{2.3.7}$$

The key idea with RND is that, unlike ICM, we are not predicting the feature representation of the next state. Instead, we aim to predict the output of a fixed, randomly initialised network, i.e., the output of a deterministic function.

In RL, we treat extrinsic rewards as episodic, meaning that the agent focuses on achieving goals within each episode. It does not optimise for goals beyond the episode, as returns are reset upon episode termination. Practically, there is a variable called `done` that signals when an episode has concluded. This variable is used to truncate the return. We aim to treat extrinsic rewards as episodic because, as Burda et al. [41] argue, treating extrinsic rewards as non-episodic could leak information about the true reward function. For example, if a game progressively increases in difficulty, the agent could potentially exploit rewards by restarting the episode deliberately to continue earning easy rewards from the start.

Burda et al. [20] argue that intrinsic rewards should be treated as non-episodic, as this promotes better exploration. For instance, consider an RL agent playing a video game: it may want to explore a risky move to defeat an enemy efficiently. The move could be risky enough to end the episode if it fails, but if intrinsic rewards were episodic, future intrinsic rewards would reset to zero at the end of the episode. Thus, the agent might avoid this potentially rewarding exploration strategy, missing new and valuable tactics for future episodes.

When intrinsic and extrinsic rewards are treated as non-episodic and episodic, respectively, it is not immediately clear how to combine them. The solution is to use two value heads: V_E for extrinsic value and V_I for intrinsic value. Thus, the total value, V, is simply,

$$V = V_E + V_I. \tag{2.3.8}$$

The total return, R, is then,

$$R = R_E + R_I, \tag{2.3.9}$$

where R_E is the extrinsic return and R_I is the intrinsic reward.

However, since we have two value heads, one for the extrinsic reward and one for the intrinsic reward, we actually have two advantages: A_E and A_I. Here, A_E is the advantage of the extrinsic reward, and A_I is the advantage of the intrinsic reward. To compute the total advantage, we use a weighted sum of these two advantages:

$$A = \beta \cdot A_E + \chi \cdot A_I.$$

Here, A is the total advantage, and $\beta > 0$ and $\chi > 0$ are hyperparameters that control the weighting of the extrinsic and intrinsic advantages, respectively.

The final component in RND is observation and intrinsic reward normalisation. Note that only the observations for the target and predictor networks are normalised. We normalise observations by subtracting the running mean, dividing by the running standard deviation, and clipping between -5 and 5. We initialise observation parameters by running a random agent in the environment before training the RL agent. Normalisation ensures the variance of the target network's embeddings isn't excessively low, allowing the predictor network to detect novel states more easily. Low variance in embeddings could make new states appear almost identical to previously visited states, thereby hindering exploration.

Normalising intrinsic rewards is also essential, as they can vary significantly across training stages and environments. To ensure hyperparameters work consistently across environments, intrinsic rewards are normalised at the batch level by dividing them by the running estimate of the standard deviations of the intrinsic returns.

BYOL-Explore

BYOL-Explore is based on a self-supervised representation learning algorithm called Bootstrap Your Own Latent (BYOL) [42]. Similar to RND, BYOL-Explore includes a network that predicts the output of a target network. Specifically, there is an online network with parameters θ and a target network with parameters ϕ. The online network comprises an encoder, a closed-loop recurrent neural network (RNN) cell, an open-loop RNN cell, and a predictor, whereas the target network consists solely of an encoder. We will now discuss the architecture of the online network in more detail. Figure 2.7 [6, Figure 1] shows the architecture of the online network.

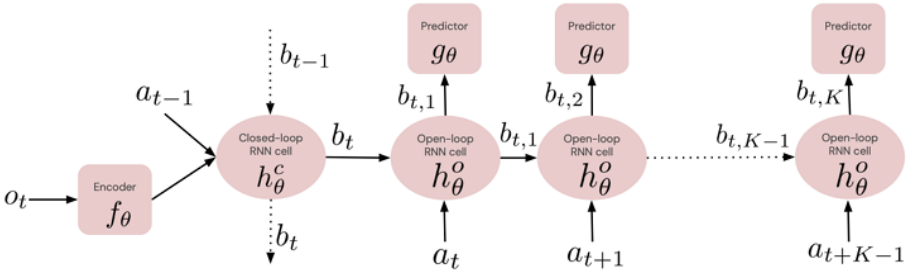


Figure 2.7: The architecture of BYOL-Explore. Taken from [6].

The online network takes in the raw observation, o_t, and passes it through its encoder f_θ to output the feature representation. Next, there is a closed-loop RNN cell, denoted by h^c_θ. Assume that at time t, h^c_θ outputs b_t, which is a representation of the history it has seen so far. The output of f_θ, the agent's previous action, a_{t-1}, and the prior history representation, b_{t-1}, are then fed into h^c_θ to produce the next history representation b_t. In other words,

$b_t = h^c_\theta(b_{t-1}, a_{t-1}, f_\theta(o_t))$.

This history representation b_t is subsequently fed into the open-loop RNN cell, h^o_θ, along with the current action, a_t. The open-loop RNN cell outputs future history representations, $(b_{t,k})_{k=1}^K$, where $b_{t,k} = h^o_\theta\left(b_{t,k-1},\, a_{t+k-1}\right)$, and K is the open-loop horizon. Here, $b_{t,0} = b_t$. The open-loop RNN generates future history representations based on previous representations and forthcoming actions. It learns how the history representation will change in response to the actions taken.

The output from h^o_θ is then fed into the predictor, g_θ, which outputs a prediction of the feature representation of the next state. The output from our predictor, $g_\theta(b_{t,k})$, represents our prediction at time $t + k$.

The target network consists solely of an encoder, f_ϕ, which outputs the feature representation of the observation at time t, o_t. In this case, we focus on predicting only the next state, so $K = 1$. Unlike RND, the target network's parameters are updated via the exponentially moving average (EMA) of the online network's encoder. Thus, the target network's parameters are updated as $\phi \leftarrow \alpha\phi + (1-\alpha)\theta$. The output from the target network, $f_\phi(o_t)$, serves as the target that the online network's predictor aims to match.

We now focus on the loss function for the online network. Assume our agent has obtained a batch of experience, $\left(\left(o^j_t, a^j_t\right)_{t=0}^{T-1}\right)_{j=0}^{B-1}$, where $B \in \mathbb{N}$ is the batch size, and $T \in \mathbb{N}$ is the trajectory length. The loss function of the online network is given by:

$$\mathcal{L} = \frac{1}{B(T-1)} \sum_{j=0}^{B-1} \sum_{t=0}^{T-2} \left\| \frac{g_\theta(b^j_{t,1})}{\left\| g_\theta(b^j_{t,1}) \right\|_2} - \frac{f_\phi(o^j_{t+1})}{\left\| f_\phi(o^j_{t+1}) \right\|_2} \right\|_2^2 . \tag{2.3.10}$$

Equation (2.3.10) represents the average error over the batch acquired by our RL agent. We next discuss how the intrinsic reward is obtained at time-step t. We begin with:

$$l^j_t = \left\| \frac{g_\theta(b^j_{t,1})}{\left\| g_\theta(b_{t,1})^j \right\|_2} - \frac{f_\phi(o^j_{t+1})}{\left\| f_\phi(o^j_{t+1}) \right\|_2} \right\|_2^2 . \tag{2.3.11}$$

This is the prediction error associated with the transition $(o^j_t,\, a^j_t,\, o^j_{t+1})$. We normalise the prediction errors of the batch, $\left(\left(l^j_t\right)_{t=0}^{T-2}\right)_{j=0}^{B-1}$, by dividing them by the EMA estimate of their standard deviation, σ_r. The final step in obtaining intrinsic rewards is to apply reward prioritisation.

Reward prioritisation operates by directing attention towards environment regions where intrinsic rewards are high, whilst neglecting areas with low intrinsic

rewards. This approach allows the agent to focus on environmental features it understands the least. As training progresses, areas previously overlooked due to low intrinsic rewards gradually become prioritised. The implementation uses the EMA mean relative to successive batches of normalised prediction errors, denoted by μ/σ_r, as a clipping threshold to distinguish between low and high prediction errors. The intrinsic rewards in a batch are then given by:

$$r_{i,t}^{j} = \max\left(\frac{l_t^j}{\sigma_r} - \frac{\mu}{\sigma_r}, 0\right). \tag{2.3.12}$$

Here, $r_{i,t}^{j}$ represents the intrinsic rewards. To obtain the total reward, we apply a weighting $\lambda \in (0, 1]$ to the intrinsic rewards and add it to the extrinsic rewards, $r_{e,t}^{j}$. The total reward in a batch is therefore given by,

$$r_t^j = r_{e,t}^{j} + \lambda r_{i,t}^{j}. \tag{2.3.13}$$

After examining ICM, RND, and BYOL-Explore, we've seen how adding intrinsic motivation enables RL agents to explore sparse reward environments by using curiosity to provide an internal signal to guide agents. However, even with these improvements, RL algorithms are still sample inefficient, i.e., the algorithms require thousands if not millions of time-steps to solve a task. This limits their real-world application as the algorithms cannot adapt quickly enough to new tasks. Our next section, meta-reinforcement learning, addresses this challenge.

2.4 Meta-Reinforcement Learning

> *"The upside of reinforcement learning is that if you want to do well in an environment, you're free to overfit like crazy. The downside is that if you want to generalize to any other environment, you're probably going to do poorly, because you overfit like crazy."*
>
> — *Alex Irpan, Deep Reinforcement Learning Doesn't Work Yet*

Meta-reinforcement learning (meta-RL) is about designing RL agents that learn how to reinforcement learn. Meta-RL is attractive to researchers because it is promising to addressing the sample inefficiency issue with RL algorithms [22]. By this we mean that RL algorithms require many time-steps to train on an environment and this often limits their real-world application use especially when the agent is faced with a new task that requires fast adaptation.

As an example, imagine we have an RL agent designed to learn different first-person shooter games on a PlayStation 5. Each game varies in its controls. One game uses the R2 button to shoot, while another uses the R1 button. However, many core skills, such as taking cover or reloading after shooting an enemy, are generalisable across games. Ideally, the RL agent should quickly adapt to

the different controls without needing to re-learn these fundamental skills from scratch each time it encounters a new first-person shooter game.

Training the RL agent from scratch every time it encounters a new game would be too time-consuming and inefficient, especially for skills that are transferable, like taking cover. We don't want to waste resources on relearning behaviours that can be applied across multiple first-person shooter games. Instead, we can train the RL agent on a distribution of first-person shooter games, each with its own unique control scheme. This way, it learns to adjust quickly to new games, even if the controls differ.

The primary trade-off with meta-RL algorithms is that they require more training time initially, as they must learn across multiple tasks or environments. However, this investment pays off because the meta-RL agent is more sample-efficient at test time, allowing it to adapt more quickly and effectively. This approach is especially beneficial when efficient adaptation is frequently needed.

Meta-RL's potential lies in it improving upon the sample efficiency of RL algorithms by learning RL algorithms or components of RL algorithms to allow adaptation to new, unseen environments or tasks. In this subsection we discuss the definition of meta-RL as well as its problem categories and finally conclude with the optimisation issues in meta-RL.

2.4.1 Definition

The Meta-RL Process

The core idea of meta-RL is that the agent learns the entire or parts of an RL algorithm, which we denote as f. In RL, f is designed to output a policy, whereas in meta-RL the agent learns f, which then outputs the policy.

We refer to the algorithm that learns f as the outer-loop, and the learned f as the inner-loop. Whilst learning occurs in both loops, adaptation is handled by the inner-loop, and meta-training is performed by the outer-loop. Once the inner-loop is obtained, it is assessed during meta-testing.

The inner-loop should adapt to a new MDP, $\mathcal{M}$, and so meta-training requires access to a distribution of MDPs or tasks, denoted as $p(\mathcal{M})$. It is common for all tasks or MDPs within the distribution to share the same action space, $\mathcal{A}$, and the same state space, $\mathcal{S}$ [22]. Tasks typically differ in their reward functions and state-transition probability functions.

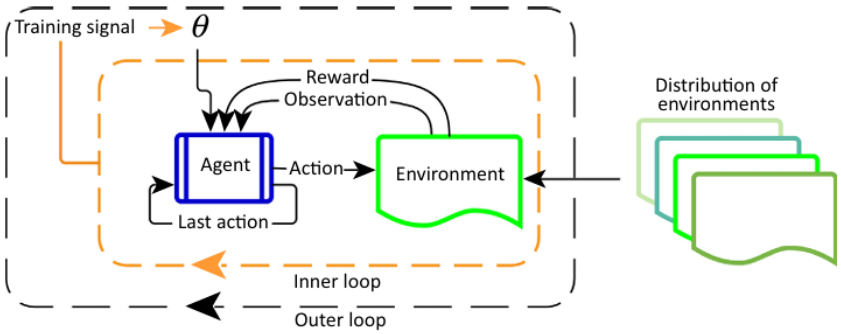


Figure 2.8: The meta-RL process. Taken from [7, Figure 1].

The meta-training process is illustrated in Figure 2.8, taken from [7, Figure 1]. The process involves sampling a task from $p(\mathcal{M})$, allowing the inner-loop to attempt solving it, and optimising the inner-loop such that policy performance increases. The lifetime is the interaction between the inner-loop with the task. Adaptation occurs during this interaction.

The Meta-RL Objective

Next, we define the objective in meta-RL. Recall that a trajectory τ_t is given by:

$$\tau_t = \{S_i, A_i, r(S_i, A_i)\}_{i=0}^{t}.$$

Assume we are working with an MDP where T is the last time-step of an episode. Let $\mathcal{D}$ denote all of the data collected during a lifetime. If the lifetime contains H episodes, we express $\mathcal{D}$ as:

$$\mathcal{D} = \{\tau_T\}_{h=0}^{H}.$$

Let the inner-loop f be parameterised by θ, where θ represents the meta-parameters. In standard RL, the objective is to maximise the episode return. In meta-RL, the performance of the agent is usually measured using the returns obtained over its lifetime on tasks sampled from $p(\mathcal{M})$. The objective can vary depending on what needs to be achieved. In general, we define the meta-RL objective as:

$$\mathcal{J}(\theta) = \mathbb{E}_{\mathcal{M}^i \sim p(\mathcal{M})} \left[\mathbb{E}_{\mathcal{D}} \left[\sum_{\tau_T \in \mathcal{D}} G_T \;\middle|\; f(\tau_T; \theta), \mathcal{M}^i \right] \right]. \tag{2.4.1}$$

This objective represents the performance of the meta-RL agent across all tasks in the task distribution. The outer expectation, $\mathbb{E}_{\mathcal{M}^i \sim p(\mathcal{M})}$, evaluates the performance across tasks, while the inner expectation, $E_{\mathcal{D}}$, considers the adaptation performance of the inner-loop on a specific MDP, $\mathcal{M}^i$.

Problem Categories

Recall that the lifetime contains H episodes, which represents the number of episodes the agent has to adapt within a task. This is also known as the task horizon.

Meta-RL is categorised based on the length of the task horizon H and the training distribution. If H is short (few episodes are needed to solve the task), then it is considered few-shot meta-RL, and if H is long (hundreds of episodes or more are needed to solve the task), it is many-shot meta-RL. If the training distribution contains multiple tasks, it is classified as multi-task meta-RL, and if it contains only a single task, it is classified as single-task meta-RL. Therefore, meta-RL can be categorised into the following categories [22]:

- Multi-Task Many-Shot Meta-RL
- Multi-Task Few-Shot Meta-RL
- Single-Task Many-Shot Meta-RL
- Single-Task Few-Shot Meta-RL

In the Multi-Task Few-Shot setting, the agent has a few episodes to adapt to the new MDP that is sampled from $p(\mathcal{M})$, which was used to train it.

However, sometimes the agent faces a challenge where it needs a long task horizon to solve the problem. The inner-loop may require hundreds or thousands of episodes to adapt to a new task. This is Multi-Task Many-Shot Meta-RL.

Meta-RL also deals with cases where the task distribution contains a single task. For Single-Task Many-Shot Meta-RL, we are concerned with whether we can somehow improve the sample efficiency without relying on training on similar tasks.

The last problem setting is few-shot single-task meta-RL. This describes a hypothetical meta-learner capable of learning a single task more efficiently without transfer from similar tasks. This is unlikely, as the inner-loop will not have enough time to adapt to the task in a data-efficient way [22].

2.4.2 Optimisation Issues in Many-Shot Meta-RL

Recall from Chapter 1 that our objective is to learn a reward combiner that is able to generalise across different tasks in Xland-MiniGrid using many-shot meta-RL. This will therefore be the focus for the rest of this section. Many-shot meta-RL refers to cases where the inner loop has a long task horizon to solve the task.

In many-shot meta-RL, we aim to improve the sample efficiency or maximise the final performance of existing RL algorithms by introducing meta-learned

components. In our case, this involves replacing the static λ value with a neural network that outputs λ at each time-step.

Optimisation in the many-shot setting is difficult because the inner loop performs hundreds or thousands of policy updates during adaptation. As a result, backpropagation is not practical for calculating the gradient, as it requires a large amount of memory, making the computational cost high [43, 22]. The main reason for these costs is that backpropagation needs to compute gradients over the agent's entire lifetime, which is often not feasible. To address this, researchers often truncate the backpropagation window [44], meaning the gradient is only computed over a small segment of the lifetime instead of the full duration. This window is usually much smaller than the total number of steps in the agent's lifetime.

This truncation leads to the emergence of short-horizon bias. Wu et al. [45] define short-horizon bias as the tendency of backpropagation-based optimisation methods in meta-RL to optimise their objective function over a short time horizon.

An alternative to backpropagation is Evolution Strategies (ES), which avoids this issue. ES allows the agent's entire lifetime to be used for optimisation without requiring additional memory [44, 9]. By not truncating the lifetime, ES eliminates short-horizon bias and ensures the optimisation process takes long-term dependencies into account. This makes ES a practical option in many-shot meta-RL settings where the cost of backpropagation is too high. The next chapter focuses on ES, in particular the ES algorithm developed by Salimans et al. [27].

2.5 Evolutionary Strategies

Evolutionary Strategies (ES) are a class of black-box optimisation techniques that are heuristic search procedures often used to optimise non-differentiable functions [27]. In ES, we have some function $F(\theta)$ which we aim to optimise with respect to the parameters of the model θ. ES makes no assumption about the structure of F and is commonly known as the fitness function.

In each generation or iteration, a population of parameters is perturbed and their fitness evaluated. The best performing parameters are recombined to form the next generation, and this process continues until the fitness function is optimised [27]. This process can be completely gradient-free. Despite ES being capable of completely gradient-free operation, researchers also make use of ES to estimate the gradient and then apply gradient descent to optimise the parameters of a neural network instead of using backpropagation to obtain the gradient. The ES algorithm introduced by Salimans et al. [27] is an example of such an ES algorithm.

Salimans et al. [27] approach involves maximising the expected fitness of the parameters, θ. The search distribution of θ is assumed to be an isotropic multivariate Gaussian where the mean is $\hat{\theta}$, the current set of parameters, and the covariance matrix is $\sigma^2 I$. The covariance matrix is fixed and I is the identity matrix and σ is the standard deviation.

The sampled solution θ is then drawn from $\mathcal{N}(\hat{\theta}, \sigma^2 I)$. This is equivalent to [46],

$$\theta = \hat{\theta} + \sigma\epsilon,\ \epsilon \sim \mathcal{N}(0, I).$$

With this we can write our objective $J(\theta)$ as,

$$J(\theta) = \mathbb{E}_{\epsilon\sim\mathcal{N}(0,I)} F(\hat{\theta} + \sigma\epsilon). \tag{2.5.1}$$

Next, we need to determine the expression for the gradient of $J(\theta)$.

$$\begin{aligned}
&\nabla_\theta \mathbb{E}_{\epsilon\sim\mathcal{N}(0,I)} F(\hat{\theta} + \sigma\epsilon) \\
&= \nabla_\theta \int_\epsilon p(\epsilon) F(\hat{\theta} + \sigma\epsilon) d\epsilon
\end{aligned}$$

Here $p(\epsilon)$ is a multivariate Gaussian, $p(\epsilon) = (2\pi)^{-\frac{n}{2}} \exp(-\frac{1}{2}\epsilon^\top \epsilon)$. Continuing,

$$\begin{aligned}
&\nabla_\theta \int_\epsilon p(\epsilon) F(\hat{\theta} + \sigma\epsilon) d\epsilon \\
&= \int_\epsilon \nabla_\theta \left(p(\epsilon) F(\hat{\theta} + \sigma\epsilon) \right) d\epsilon \\
&= \int_\epsilon \nabla_\theta \epsilon \left[\nabla_\epsilon \left(p(\epsilon) \right) \cdot F(\hat{\theta} + \sigma\epsilon) \right] d\epsilon
\end{aligned}$$

Now if we use the log-likelihood trick we can rewrite $\nabla_\epsilon \left(p(\epsilon) \right)$ as $\nabla_\epsilon \log(p(\epsilon)) \cdot p(\epsilon)$. Continuing,

$$\begin{aligned}
&\int_\epsilon \nabla_\theta \epsilon \left[\nabla_\epsilon \left(p(\epsilon) \right) \cdot F(\hat{\theta} + \sigma\epsilon) \right] d\epsilon \\
&= \int_\epsilon \nabla_\theta \epsilon \left[\nabla_\epsilon \log(p(\epsilon)) \cdot p(\epsilon) \cdot F(\hat{\theta} + \sigma\epsilon) \right] d\epsilon \\
&= \int_\epsilon p(\epsilon) \nabla_\theta \epsilon \left[\nabla_\epsilon \log(p(\epsilon)) \cdot F(\hat{\theta} + \sigma\epsilon) \right] d\epsilon \\
&= \mathbb{E}_{\epsilon\sim\mathcal{N}(0,I)} \left[\nabla_\epsilon \left(\log\left((2\pi)^{-\frac{n}{2}} \right) + -\frac{1}{2}\epsilon^\top \epsilon \right) \cdot F(\hat{\theta} + \sigma\epsilon) \cdot \nabla_\theta \epsilon \right] \\
&= \mathbb{E}_{\epsilon\sim\mathcal{N}(0,I)} \left[(-\epsilon) \left(\frac{1}{\sigma} \right) F(\hat{\theta} + \sigma\epsilon) \right]
\end{aligned}$$

Note that $\epsilon = \frac{\theta - \hat{\theta}}{\sigma}$. Now the negative sign in $(-\epsilon)$ can be absorbed into the gradient descent step and we finally obtain,

$$\begin{aligned}
&\nabla_\theta \mathbb{E}_{\epsilon\sim\mathcal{N}(0,I)} F(\hat{\theta}+\sigma\epsilon) \\
&= \mathbb{E}_{\epsilon\sim\mathcal{N}(0,I)} \left[(-\epsilon) \left(\frac{1}{\sigma}\right) F(\hat{\theta}+\sigma\epsilon) \right] \\
&= \mathbb{E}_{\epsilon\sim\mathcal{N}(0,I)} \left[(\epsilon) \left(\frac{1}{\sigma}\right) F(\hat{\theta}+\sigma\epsilon) \right].
\end{aligned}$$

The above expression is estimated with samples. In order to reduce the variance in the gradient estimate it is common to make use of antithetic sampling. Instead of generating samples we generate pairs where one is the negative of the other. For example instead of just sampling $\epsilon \sim \mathcal{N}(0, I)$ we sample $-\epsilon$ as well. With antithetic sampling the gradient estimate becomes,

$$\nabla_\theta J(\theta) = \frac{1}{2}\mathbb{E}_{\epsilon\sim\mathcal{N}(0,I)} \left[(\epsilon) \left(\frac{1}{\sigma}\right) \left[F(\hat{\theta}+\sigma\epsilon) - F(\hat{\theta}-\sigma\epsilon) \right] \right]. \tag{2.5.2}$$

We now present the ES algorithm introduced by Salimans et al. [27]. For the remainder of this work, whenever we refer to ES, we specifically mean the Evolution Strategies algorithm proposed by Salimans et al. [27].

Algorithm 5 Evolution Strategies (ES) [27] with Antithetic Sampling

Require: Learning rate $\alpha \geq 0$, noise standard deviation $\sigma > 0$, number of samples n, initial parameters θ_0 and fitness function F.

Ensure: Optimised parameters θ^*.

1: Initialise parameters θ_0 randomly.
2: **for** $t = 0, 1, 2, \ldots$ **do**
3: Sample perturbations $\epsilon_1, \ldots, \epsilon_n \sim \mathcal{N}(0, I)$.
4: Generate antithetic perturbations: for each ϵ_i, include $-\epsilon_i$.
5: **for** $i = 1$ to n **do**
6: Compute fitness function evaluations:

$$F_i^+ = F(\theta_t + \sigma\epsilon_i), \quad F_i^- = F(\theta_t - \sigma\epsilon_i)$$

7: **end for**
8: Compute the gradient estimate using antithetic pairs:

$$\hat{g} = \frac{1}{2n\sigma} \sum_{i=1}^{n} \epsilon_i (F_i^+ - F_i^-)$$

9: Update parameters:

$$\theta_t \leftarrow \theta_t - \alpha\hat{g}$$

10: **end for**
11: **return** θ_t

In this chapter we have now laid the foundation needed to understand our methodology by discussing traditional RL methods such as GPI and Q-learning. We then proceeded with exploring modern RL algorithms by examining value-based methods, policy-based methods and actor-critic methods, which leads us to PPO, our selected RL algorithm. Next we presented curiosity-driven exploration algorithms and how they are implemented to encourage exploration. To understand how researchers have aimed to improve the sample efficiency of RL algorithms, we looked into meta-RL. Following this, we introduced ES [27], a black-box optimisation technique used to estimate gradients of fitness functions. Finally, we concluded by outlining the LSTM and GRU architecture.

Before proceeding to our methodology, it is important that we examine existing work that has addressed similar questions. This includes research on meta-learning intrinsic rewards and on using meta-RL with ES to develop general-purpose RL algorithms, and how our work departs from these approaches.

Chapter 3

Related Work

Recall from Chapter 1 that our research question examines how well a reward combiner can dynamically output appropriate intrinsic reward weightings (λ) across different environments without requiring retraining or fine-tuning. Our approach falls into the many-shot category of meta-RL and the reward combiner is optimised using ES [27]. Therefore, this chapter reviews literature relevant to our approach, focusing on two main research areas:

1. Meta-learning of intrinsic rewards and reward combination.
2. Applications of ES to optimise meta-learned components in RL algorithms.

We structure this review around these two areas, which most closely relate to our approach. By examining these areas, we contextualise the work done in this thesis within the existing literature and highlight the distinguishing aspects of our approach.

3.1 Meta-learning of intrinsic rewards and reward combination

LIRPG: Learning Intrinsic Rewards for Policy Gradient

As stated in Chapter 1, learning in sparse reward environments is challenging. Intrinsic motivation is seen as a way to improve performance in these environments, but the algorithms that output intrinsic rewards are often hand-designed. Recognising this limitation, Zheng et al. [47] proposed Learning Intrinsic Reward for Policy Gradient (LIRPG), which replaces hand-designed rewards with a learnable intrinsic reward function for a single-task setting.

In their work, the intrinsic rewards are given by a neural network parameterised by ν. The authors denote this neural network by r^i_ν and refer to it as the intrinsic

reward module. This neural network takes in the current state and action and then outputs the intrinsic reward. LIRPG can be used with any policy gradient method or actor-critic method, such as PPO or A2C. The agent's policy is parameterised by θ, and the authors denote its policy by π_θ. In this work, the objective of r_ν^i is given by the expected extrinsic return. The RL agent aims to maximise the expected cumulative sum of extrinsic and weighted intrinsic rewards:

$$J_\theta = \mathbb{E}_\theta \left[\sum_{t=0}^{T} \gamma^t \left(r_t^e + \lambda r_\nu^i(s_t, a_t) \right) \right]. \tag{3.1.1}$$

In the above equation, λ is the relative weight of the intrinsic reward, r_t^e is the extrinsic reward at time-step t, and T denotes the final time-step of the episode. The policy parameters θ are updated via gradient ascent.

The LIRPG agent was tested and trained on a set of 15 Atari games. In the Atari experiments, the LIRPG was used with A2C. The baselines were a standard A2C agent and an A2C-bonus agent. The A2C-bonus agent received a constant live bonus of 0.01 at each time step to encourage the agent to live longer so the chances of it receiving extrinsic rewards increase. For these experiments, they explored values for the intrinsic reward weighting coefficient λ and selected the best-performing value for each game individually. This required separate hyperparameter tuning for each environment. This approach becomes less practical when scaling to larger numbers of tasks. The LIRPG agent outperformed the A2C baseline agent on all 15 games. However, the LIRPG agent only outperformed the A2C-bonus agent in 13 out of 15 games. In particular, it obtained lower episode returns on Asterix and Space Invaders environments. This suggests that in games where extended survival directly correlates with higher scores, as in Asterix and Space Invaders, even a simple constant bonus can be effective.

Zheng et al. [47] conducted experiments on the MuJoCo environments as well. In particular, they conducted experiments on the Ant, Hopper, Walker2d, HalfCheetah, and Humanoid. This time PPO was the algorithm of choice and so the baselines were standard PPO and PPO-bonus. However, for the PPO-bonus they explored the values 0.001, 0.01, 0.1, and 1, and used the domain-specific best-performing choice. These experiments also included another agent that was just trained on the intrinsic reward provided by the intrinsic reward module known as PPO-R_{in}.

For the MuJoCo experiments, they took a different approach to the weighting parameter, fixing $\lambda = 1.0$ across all environments and instead scaling the extrinsic rewards by 0.01 for the LIRPG agent. To create a more challenging learning scenario, they implemented a reward delay mechanism where the extrinsic rewards were accumulated for 20 time-steps before being provided to the agent. This delay created a more sparse reward setting. The LIRPG agent outperformed standard PPO and PPO-bonus in all tasks except Ant. PPO-

R_{in} achieved higher episode returns than the LIRPG agent in the Half-Cheetah environment, and achieved similar performance in the Ant, Walker2d and Humanoid environments. This shows that the intrinsic reward module successfully captures more than the bonus given to the PPO-bonus agent at each time-step due to its objective being the extrinsic episode return.

Learning during Practice: Extending LIRPG

To extend learned intrinsic rewards beyond individual episodes, Rajendran et al. [48] explored how such rewards could guide learning during practice periods separate from actual task performance. During practice, the agent does not have access to extrinsic rewards and must rely on rewards generated by r_ν^i. The RL agent alternates between practice and matches (the actual task) during training. Humans often have practice sessions in sports or video games. These practice sessions can differ significantly from the actual match. For example, football players may focus on specific skills like shooting or free kicks. These practice sessions should ideally improve performance in actual matches. The aim is to replicate this for RL agents.

The objective that agents aim to maximise during a match is the expected extrinsic return. During practice, the RL agent's objective is the expected cumulative sum of the weighted intrinsic rewards. The objective of the intrinsic module is the same as the RL agent's match objective. Its parameters are updated during the practice session.

Rajendran et al. [48] evaluated their method on three environments: 1.) A grid-world containing a corridor of length 8. The objective is for the agent to take the trash from the leftmost side of the corridor to the bin on the rightmost side of the corridor. 2.) The Atari game Pong. 3.) The Atari game Pac-Man.

In the simple grid-world environment, there was no difference between the practice setting and the match setting. This is to isolate the effect of learned intrinsic rewards from environmental differences. The authors employed REINFORCE. The agent was trained for 200 matches. The length of the match and practice session varied between 45 and 50, which were sampled uniformly. Practice occurred after every match. During the training, progressive improvement in episode returns is seen. The agent progressed from achieving single trash deliveries (+1 reward) to completing multiple cycles (+2 reward) per episode.

In the Atari experiments, the match setting differed from the practice setting. In Pong, the practice session involved the agent hitting the ball against a wall instead of an opponent. In Pac-Man, the practice environment involved the same maze but with no ghosts. The authors compared their approach against several baselines, including agents that learned only during matches and agents using either fixed random intrinsic rewards or intrinsic rewards predicted from extrinsic rewards during practice.

The results demonstrated that learning during practice improved sample efficiency compared to match-only learning. Their proposed solution outperformed the baseline approaches, indicating that meta-gradients are important for improving intrinsic rewards.

Our approach differs from these prior works in several key ways. First, both prior works learn intrinsic rewards through neural networks, whereas our method makes use of hand-designed curiosity algorithms (RND and BYOL-Explore) and focuses on learning how to combine them with extrinsic rewards. Second, their approaches operate within single-task meta-RL frameworks, requiring retraining for each new environment, whilst our work addresses multi-task meta-RL within grid-world environments with navigation and/or door-key mechanics, aiming to generalise across different tasks with similar mechanics without retraining. Third, neither prior work investigated meta-learning the combination weights between intrinsic and extrinsic rewards: Zheng et al. [47] required environment-specific hyperparameter tuning for λ, whilst Rajendran et al. [48] relied solely on intrinsic rewards during practice periods. Fourth, whilst Rajendran et al. [48] demonstrated faster convergence in match performance, the overall training time including practice sessions may offset these gains. Finally, both works make use of backpropagation for gradient computation of intrinsic reward modules during training, updating parameters concurrently with policy updates. In contrast, our approach considers the agent's entire training lifetime and makes use of ES to optimise the reward combiner.

Lifetime Intrinsic Reward Learning

Both LIRPG and its practice extension optimised for individual episode returns rather than considering the agent's entire training lifetime. Zheng et al. [49] aimed to see if one can learn intrinsic rewards that encourage exploration across multiple episodes in an agent's lifetime. The authors extend LIRPG by considering the entire lifetime return of the agent across multiple episodes, as this allows exploration across episodes during training. LIRPG considered a single task in a single lifetime using the episode return as the intrinsic reward module's objective. To capture long-term temporal dependencies across episodes, they parametrise their intrinsic reward function using an LSTM architecture. The objective of the reward function is the expected extrinsic return over the agent's lifetime, averaged across the distribution of tasks (MDPs). Another key difference is that the reward function, r_ν, parameterised by ν, takes in the entire trajectory, where:

$$\tau_t = s_0, a_0, r(s_0, a_0), d_0, \ldots, s_t, a_t, r(s_t, a_t), d_t.$$

Here, the variable d_t is a binary value that indicates whether an episode has terminated.

In LIRPG [47], the agent aimed to optimise a weighted sum of intrinsic and extrinsic rewards, with the intrinsic reward being output by the intrinsic reward

module, r^i_ν. In this case, the intrinsic reward function takes in τ_t and outputs an intrinsic reward, and the RL agent aims to maximise the sum of the intrinsic rewards.

Calculating the gradient for the intrinsic reward moduleis infeasible as it requires backpropagating throughout the agent's lifetime. To address this, Zheng et al. [49] update the intrinsic reward function every N policy updates using the estimated lifetime return of the agent. The estimation of the lifetime return is given by a neural network V_ϕ which takes in the same inputs as r_ν. The neural network V_ϕ is updated at the same time as r_ν.

To test their intrinsic reward model r_ν, Zheng et al. [49] trained the model on various grid-world tasks including goal-finding, object selection, and non-stationary reward scenarios. For each environment, a separate intrinsic reward model was trained across multiple lifetimes on variants of that specific task, with each model learning to generalise across the natural variation within its respective environment.

The authors compared their approach against several baselines including standard episode-based and lifetime-based extrinsic reward training, count-based exploration, and ICM. The intrinsic reward approach consistently outperformed all baselines, demonstrating the model's ability to encourage appropriate exploration-exploitation behaviour and adapt to environment changes within a lifetime. This showed that the intrinsic reward model could successfully carry information across episodes during the agent's lifetime.

Like Zheng et al. [49], our work focuses on grid-world environments, specifically using the XLand-MiniGrid framework. However, what distinguishes our approach is that we learn the intrinsic reward weighting λ rather than learning the intrinsic reward function itself. Their learned intrinsic reward model directly outputs reward signals that combine intrinsic and extrinsic components, whereas our approach learns the weighting parameter for combining curiosity-driven intrinsic rewards with extrinsic rewards. Additionally, our work focuses on multi-task learning, where the learned weighting generalises across different grid-world tasks with navigation and/or door-key mechanics within XLand-MiniGrid, whereas they trained separate intrinsic reward models on variants of the same environment. While both approaches optimise over agent lifetimes, a key difference in our methodology lies in our optimisation strategy. We use a black-box approach with ES, whereas they employ backpropagation. Because of the computational cost of backpropagating through entire lifetimes, they cannot use the actual lifetime return and must rely on bootstrapped estimations, whereas our ES approach enables direct optimisation of the actual lifetime return.

Meta-learning Curiosity Algorithms

The works we have discussed have relied on optimising neural network weights to learn the intrinsic reward. Therefore, the learned intrinsic rewards are often used in very similar environments (same state and action space). Alet et al. [50] took a different and unique approach to the works we have discussed so far. Instead of optimising neural network weights, they meta-learned pieces of code. In other words, the meta-learning occurred in the space of programs and operations. These operations are similar to those used in hand-designed curiosity algorithms. They used these operations to form a domain specific language. The reason for this was to design a curiosity module, $\mathcal{C}$, that achieves generalisation across vastly different environments with different action and state spaces.

This module had two components working together. Component $\mathcal{I}$ calculates the intrinsic reward, while component χ combines the extrinsic reward, intrinsic reward, and current normalised timestep of the agent's lifetime to output $\hat{r}_t$, which feeds into the RL agent. Meta-learning these pieces of code resulted in two new interpretable algorithms for component $\mathcal{I}$: Fast Action Space Transition (FAST) and Cycle-Consistency Intrinsic Motivation (CCIM). These algorithms were found using the grid-world environment, where the objective is to find the goal position. The authors conducted a systematic search through their domain-specific language (DSL) in the grid-world environment, using the number of distinct cells visited as their optimisation metric.

However the main flaw of these algorithms is that the intrinsic reward did not decrease during training [51].

The component χ, used to combine the intrinsic reward and extrinsic reward, was found using the Lunar Lander environment. This is an environment with a dense reward function and was the reason why the authors selected it. The intrinsic rewards were provided by an ensemble variant of RND. The authors searched through 2500 possible reward combiners and selected the best performing one in the lunar lander environment. The formula discovered was:

$$\hat{r}_t = \frac{(1 + r_t^i - t/T) \cdot r_t^i + r_t^e \cdot t/T}{1 + r_t^i}. \tag{3.1.2}$$

Here, T is the number of timesteps in the agent's lifetime.

The authors evaluated their meta-learned curiosity algorithms across multiple environments: Acrobot, and two MuJoCo environments (Ant and Hopper). Using PPO as their base RL algorithm, they compared their approaches against several baselines: agents receiving constant intrinsic rewards (-1, 0, 1) and Gaussian noise, as well as published curiosity algorithms including RND, ICM. Importantly, they used the same fixed reward combiner (Equation (3.1.2)) across all environments and algorithms. Their results showed that meta-learned curiosity algorithms performed statistically comparable to the published curiosity methods and statistically outperformed the constant-bonus baselines. This

demonstrated that their automatically discovered algorithms could generalise effectively across environments with vastly different state and action spaces, despite being initially discovered only in the simple grid-world task.

In contrast to our approach, Alet et al. [50] attempted to generalise across vastly different domains (from grid navigation to continuous control tasks like MuJoCo), whereas we focus on generalisation within XLand-MiniGrid environments involving navigation and/or door-key interaction mechanics. Another difference lies in our methodologies: while they search through a space of programs to discover curiosity algorithms and a reward combiner, we optimise neural network weights using ES to learn an adaptive reward combiner. For intrinsic rewards, they meta-learn the curiosity algorithms themselves, whereas we employ established methods like BYOL-Explore. Regarding reward combination, we utilise a simple weighted sum with a learned λ parameter, while they propose Equation (3.1.2). However, as noted by Ziki [51], their discovered curiosity algorithms do not exhibit diminishing intrinsic rewards over time. Furthermore, one of their baselines, the agent receiving zero intrinsic reward but still using their reward combiner formula, results in extrinsic rewards being scaled by normalised time ($t/T \cdot r_t$). This baseline agent receives diminished rewards at the beginning of training, gradually increasing to full rewards only at the end.

The works reviewed in this section demonstrate various approaches to addressing the challenge of sparse rewards through intrinsic motivation, yet each faces computational efficiency limitations when applied across multiple environments. LIRPG required environment-specific hyperparameter searches to determine optimal λ values. This process becomes computationally expensive when scaling to numerous tasks. The practice-based extension addressed some limitations but introduced uncertainty about whether the additional training time in practice sessions justifies the performance improvements. The lifetime learning approach made progress by considering longer horizons but was constrained by truncation issues and focused primarily on variants within the same task rather than generalisation across different environments. Alet et al. [50] demonstrated promising cross-domain generalisation capabilities, but their approach applied the discovered reward combiner across all methods and baselines, making it challenging to isolate the specific contribution of adaptive reward combination. Our work takes a different angle, focusing specifically on learning the intrinsic reward weighting parameter λ using multi-task meta-learning with ES. This approach aims to eliminate the need for hyperparameter sweeps across new grid-world tasks with similar mechanics by training a reward combiner that can dynamically output appropriate λ values, thereby addressing the computational efficiency challenges highlighted in Chapter 1.

3.2 Applications of ES in many-shot meta-RL

Zheng et al. [49] highlight a key challenge: backpropagating through entire agent lifetimes is computationally expensive. This motivates our approach of ES as an alternative optimisation approach that can directly optimise lifetime returns. In this section we review works which make use of ES when the objective involves the lifetime of the RL agent.

Learning Objective Functions

In RL we often distinguish between value-based methods (all of which can be seen as instances of Generalised Policy Iteration) and policy-gradient methods (many of which use trust-region constraints, as in PPO). Q-learning is a classic value-based (i.e. GPI) algorithm. TRL is a framework in which the focus is on restricting policy updates to ensure stability and prevent performance collapse. PPO somewhat falls into the TRL framework, as it constrains policy updates and is designed to prevent performance collapse. However, it uses a clipped surrogate objective, essentially constraining policy updates in an approximate way. Therefore, despite being one of the most successful RL algorithms, it lacks theoretical guarantees [52].

Kuba et al. [52] introduced a new framework called Mirror Learning, which unifies algorithms that fall into GPI and TRL. Algorithms within the Mirror Learning framework come with monotonic improvement properties and convergence guarantees. It even provides theoretical guarantees for PPO!

Mirror Learning achieves this by providing a drift function $\mathcal{D}_{\pi_k}(\pi|s)$, which measures the difference between policy π_k and policy π. The drift function possesses important properties. It should be non-negative, and its derivative with respect to π_k must be zero. With these properties, we achieve monotonic improvement in our policy updates.

It is because of the theoretical guarantees of Mirror Learning that Lu et al. [53] focused on discovering new RL algorithms within this framework. The authors focused on improving PPO by having a neural network with parameters θ learn the drift function. This algorithm is known as Learned Policy Optimisation (LPO). LPO was trained on a single environment, the Ant task from the Brax environments [54]. LPO was trained using ES, where its fitness function, $F(\theta)$, was

$$F(\theta) = \mathbb{E}\left[J(\pi)\right].$$

Here, $J(\pi)$ is the expected discounted return of policy π discovered by LPO. To ease learning, the drift network was also initialised close to PPO's drift function.

By analysing the drift function at the end of training LPO, Lu et al. [53] were able to obtain a closed-form version of LPO. This closed-form variant is known as Discovered Policy Optimisation (DPO).

Lu et al. [53] proceeded to test LPO and DPO on the other Brax environments where standard PPO algorithm served as the baseline. LPO and DPO outperformed PPO on most environments which indicate that LPO and DPO generalise to unseen environments.

Jackson et al. [9] built upon LPO by including a temporal component into the input of the drift network. The authors highlight that humans adapt and change their strategy based on time. For example, a football team behind on the scoreboard may adopt more aggressive tactics as time runs out in an attempt to win. In particular, they included the normalised time step in the input of the drift function. This allows LPO to be conditioned on the agent's lifetime. This version of LPO is known as Temporally-Adaptive LPO (TA-LPO).

Jackson et al. [9] trained TA-LPO on the MinAtar version [55] of the Space Invaders environment. The baselines were LPO (trained on Space Invaders as well) and standard PPO. The authors then proceeded to test TA-LPO on the other MinAtar environments and the Brax environments. TA-LPO was able to outperform the baselines on all the testing MinAtar and Brax environments, highlighting its generalisation capabilities given that it was only trained on Space Invaders.

Similar to the works by Lu et al. [53] and Jackson et al. [9], our method employs ES for meta-optimisation. However, our approach differs in several key aspects. Whilst both LPO and TA-LPO perform single-task meta-RL, our method employs multi-task meta-RL for generalisation across XLand-MiniGrid environments with navigation and/or door-key mechanics. Additionally, rather than meta-learning drift functions, our approach meta-learns intrinsic reward weighting parameters. We adopt the multitask ES approach from [9] (detailed in Chapter 4). Since our objective function evaluates lifetime return, gradient-based meta-optimisation would require truncation due to computational constraints, leading to the short-horizon bias identified by both authors.

Having reviewed the relevant literature, we have covered works that have attempted to meta-learn intrinsic rewards and reward combiners, as well as applications of ES within many-shot meta-RL for optimising meta-parameters. However, none of the existing works have attempted multi-task ES specifically for learning reward combination across different environments. This represents the gap our research addresses: Developing a reward combiner that can dynamically output appropriate λ values across multiple tasks without requiring retraining or hyperparameter tuning for each new environment. In the next chapter, we detail our methodology, including our training process, the XLand-MiniGrid environments used in our experiments, and the specific experimental setup designed to answer each of our research sub-questions.

Chapter 4

Methodology

Having obtained the necessary background, the steps for answering the research question can now be detailed. All code implementations are available at https://github.com/Ziksby/MetaLearnCuriosity. Recall that fine-tuning λ for curiosity-driven baselines can be expensive across many environments. The research question was thus:

> **Research Question:** How does the performance of an agent that learns to dynamically output intrinsic reward weightings (λ) compare to manually fine-tuned curiosity-driven algorithms and standard PPO across grid-world tasks involving navigation and/or door-key interaction mechanics?

We conduct all experiments in the XLand-MiniGrid environments [8]. These environments are discrete grid-world tasks with sparse rewards, therefore requiring exploration to solve the tasks as the extrinsic reward is only provided at the end of the episode. The reward combiner (RC) took in the normalised extrinsic and intrinsic rewards as well as the actions. The actions in XLand-MiniGrid provided context for what the goal of the task is. For example, obtaining the extrinsic reward after unlocking a door to pick up an object in the next room signalled to the RC that the actions unlocking and picking up were important to solving this environment.

The research question was divided into two sub-questions:

> **Sub-question 1:** How does the meta-learning agent's performance compare with fine-tuned curiosity-driven algorithms and standard PPO when evaluating across different grid-world sizes on an unseen task?

Sub-question 2: How does the meta-learning agent's performance compare with fine-tuned curiosity-driven algorithms and standard PPO when evaluating on an unseen task where the objective differs from those seen during training?

Our hypotheses for each sub-question were as follows:

Hypothesis 1: The meta-learning agent will perform as well as the baselines when evaluating across different grid-world sizes on an unseen task with navigation and/or door-key interaction mechanics.

Hypothesis 2: The meta-learning agent will perform as well as the baselines when evaluating on an unseen task with navigation and/or door-key interaction mechanics where the objective differs from those seen during training.

This chapter explains the methods used to answer the sub-questions and test the hypotheses. The RC was trained on `Unlock` and `Empty-16x16` environments using multi-task ES. The fitness function was the cumulative sum of the extrinsic rewards of the RL agent it trained. The RC was then tested on `DoorKey` (on all its grid sizes) and `UnlockPickUp`. The RC was compared to the curiosity-driven baselines: RND and BYOL-Explore, as well as standard PPO. Their episode return at the end of training was compared. In the training environments, the optimal λ value for RND and BYOL-Explore was searched for.

The chapter begins with a discussion of the implementation details of the algorithms, the RC, the hyperparameter search used to fine-tune the curiosity-driven baselines, and the reward normalisation scheme for both the baselines and the RC. The XLand-MiniGrid framework is then described, including its state-space representation, action-space, and the specific environments used for training and testing the RC. Following this, the training procedure of the RC is outlined, with a focus on how multi-task ES is used for training. Finally, we present the experiments conducted to answer the sub-questions and test the hypotheses.

4.1 Implementation Details

The implementation details of the algorithms are now presented. The purpose of this section is to outline the algorithms and the hyperparameter search for the optimal λ, which defines the relative weighting of the intrinsic reward. For specific hyperparameter values, refer to Appendix D.

PPO Implementation The standard PPO agent was based on the implementation provided by Nikulin et al. [8] and Lu et al. [53]. The PPO agent employed an RNN architecture with three main components: an observation

embedding layer, a GRU, and two output heads. Observations, previous rewards, and previous actions were received as input by the PPO agent. For more information about the GRU architecture please see Appendix C.

The observation embedding consisted of a single feed-forward layer with 256 neurons and ReLU activation. Previous actions were embedded using a discrete embedding layer with 16 dimensions for the 6 possible actions. The embedded observations, embedded previous actions, and previous rewards were then concatenated and passed into a single-layer GRU with 64 hidden neurons to capture temporal dependencies. Two separate output heads processed the GRU output. The actor head was comprised of two feed-forward layers (64 neurons with tanh activation, followed by a layer matching the action space dimension) and output the distribution over the actions. The critic head was comprised of two feed-forward layers (64 neurons with tanh activation, followed by a single output neuron) and output the estimated value function.

BYOL-Explore Implementation BYOL-Explore built upon the PPO architecture and shared the same hyperparameter values for common parameters. BYOL-Explore extended PPO by calculating intrinsic rewards that were combined with the extrinsic rewards. BYOL-Explore contained a target network and an online network. The target network was a 2-layer feed-forward neural network (dense layer with ReLU activation followed by a dense output layer) that took in the current observation and output the feature representation. The online network consisted of an encoder (2-layer feed-forward neural network with ReLU activation), a closed-loop GRU cell, an open-loop GRU cell, and a predictor network (2-layer feed-forward neural network with ReLU activation). The difference between the target network and the online network served as the intrinsic reward; see equation (2.3.12). The intrinsic rewards were normalised at batch level by dividing them by the EMA estimate of their standard deviation and using reward prioritisation. The EMA parameter had a value of 0.99. For detailed hyperparameter specifications, see Appendix D.

RND Implementation RND also built upon PPO and shared common hyperparameter values. RND comprised a predictor network and a target network, both using feed-forward neural networks that took in the current observation and output feature representations. The target network was a 2-layer feed-forward neural network (dense layer with ReLU activation followed by a dense output layer). The predictor network was a 3-layer feed-forward neural network with ReLU activations between layers. The intrinsic reward was given by the difference in outputs; see equation (2.3.7). The intrinsic rewards were normalised at batch level by dividing them by the running estimate of the standard deviations of the intrinsic returns. For detailed hyperparameter specifications, see Appendix D.

Reward Combiner Implementation

The RC network extended the BYOL-Explore architecture by replacing the static λ parameter with a dynamic neural network output. The RC was selected to use BYOL-Explore over RND to provide intrinsic rewards because BYOL-Explore achieved superior performance on the training environments. The RC network employed a GRU-based architecture to handle long-term dependencies, as the reward combiner required the history of inputs over the course of training the RL agent to adjust λ appropriately. The value of λ depended on how the agent was performing during training, requiring a network capable of capturing long-term temporal dependencies. A GRU was selected over an LSTM because an LSTM had more parameters, hence it would take longer to train than a GRU.

The RC network took three inputs: normalised extrinsic rewards, normalised intrinsic rewards from BYOL-Explore, and the agent's actions. Actions were included as input because they provided task-specific context for what the agent was attempting to solve, enabling the reward combiner to adjust λ accordingly for different task types. For example, when the reward combiner observes non-zero extrinsic reward following specific actions (e.g., picking up objects vs. unlocking doors), it can adjust λ accordingly for that task type.

The discrete actions were embedded using an embedding layer (6 actions embedded to 16 dimensions), whilst rewards were embedded using a feed-forward neural network (16 neurons with ReLU activation). These embeddings were then concatenated and passed through a GRU cell. The GRU output was then passed through a feed-forward neural network (64 neurons with ReLU activation) before passing through a sigmoid activation function to ensure the output λ remained between 0 and 1. Figure 4.1 illustrates the complete RC architecture.

Both intrinsic and extrinsic rewards were normalised at the batch level by dividing them by the EMA estimate of their standard deviation, with an EMA parameter value of 0.99 for both. For intrinsic rewards, reward prioritisation was applied as in BYOL-Explore, whilst for extrinsic rewards, reward prioritisation was not applied.

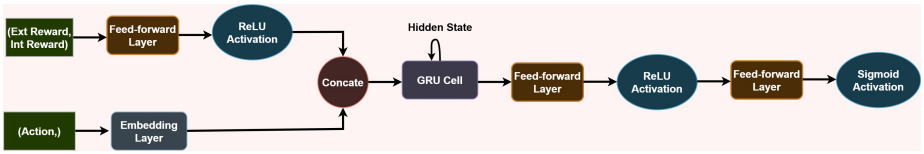


Figure 4.1: Reward Combiner (RC) architecture showing the flow from inputs (extrinsic rewards, intrinsic rewards, and actions) through embedding layers, GRU cell, and feed-forward layers to output λ. The discrete actions are embedded via an embedding layer, whilst rewards are embedded through a feed-forward layer with ReLU activation. The concatenated embeddings pass through a GRU cell, followed by a feed-forward layer and sigmoid activation to ensure $\lambda \in [0, 1]$.

Hyperparameter Search for λ

For the curiosity-driven baselines, the extrinsic and intrinsic rewards were combined using a weighted sum,

$$\hat{r} = r^e + \lambda r^i.$$

Where $\hat{r}$ was the total reward. The algorithms used a single value head for the total reward as the intrinsic reward and extrinsic reward were treated as episodic. Treating the intrinsic reward as episodic (unlike the approach described in Section 2.3.2) allowed us to use a single value head for the total reward, which simplified fine-tuning as we only needed to perform a hyperparameter sweep over the single λ parameter rather than searching over two weighting parameters (β and χ) as required when using separate advantage functions.

These λ values were selected based on ranges reported in prior curiosity-driven RL work [47] and expanded the range through empirical testing. When optimal λ values occurred at the boundaries of our initial search range, the range was extended to ensure we had not missed better values. The λ values over which we searched were

$$\lambda \in \{0.0001, 0.0003, 0.0005, 0.0008, 0.001, 0.003, 0.005, 0.01, \\ 0.02, 0.03, 0.05, 0.1, 0.2, 0.5, 0.8, 1.0\}.$$

In all experiments, 30 random seeds were used, to obtain the mean performance for each algorithm. However, running 30 random seeds for each λ value for each task was quite computationally expensive. Therefore, to perform the hyperparameter sweep, the method outlined by Patterson et al. [56] was used. For each λ value, $N = 10$ random seeds were used. The episode returns at the end of training for each agent were then obtained. Next, sampling with replacement was performed of N of those N runs and the average performance from the sampled set was computed. This procedure of sampling N of those N runs and calculating the mean was repeated $M = 1000$ times. The $M \times N$ sampled runs were used to obtain the average episode return for each λ for that specific task.

The λ value with the highest average episode return was then selected as the λ value for that specific task.

4.2 Environment Details

XLand-MiniGrid environments [8] are the JAX [57] equivalent of the MiniGrid environments [23]. These are discrete grid-worlds with different tasks on various grid sizes. In this section, the observation space and action space of the environments are covered, as well as the reward function. The discussion then moves to specific environments used for training and testing the RC.

XLand-MiniGrid Overview

XLand-MiniGrid environments consist of rectangular grid-worlds where each cell represents a discrete state. The agent can perform six actions at any time-step: `move_forward`, `turn_left`, `turn_right`, `pick_up`, `put_down`, and `toggle`.

The agent receives a partial observation of the environment through a forward-facing 7×7 view centred on its current position and orientation. Figure 4.2 illustrates the observation of the agent in the `Empty-16x16` environment, where the gray area represents the observable region within the agent's view.

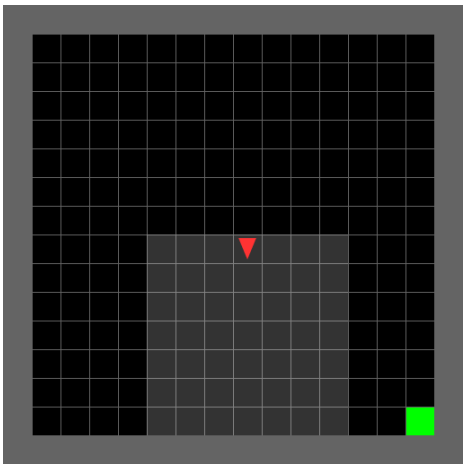


Figure 4.2: The agent's point of view in the `Empty-16x16` environment, showing the 7×7 partial observation centred on the agent's current position and orientation. The red triangle represents the agent, the green square represents the goal position, and the gray area represents the observable region within the agent's field of view, whilst the black areas are outside the agent's observation space. Taken from [8].

Each cell in the observation is encoded using two values: a tile ID and a colour

ID. The tile ID indicates the tile type (e.g., walls, doors, keys, goals), whilst the colour ID specifies its appearance. The agent cannot move through walls or closed/locked doors, and can only pass through open doors. Objects such as keys can be picked up, but the agent cannot occupy the same cell as an object unless it picks up that object first. The grid-world size includes the boundary walls.

The agent's view can be obstructed by walls and closed or locked doors. When the agent cannot see beyond these obstacles, the occluded cells are filled with empty tiles. Similarly, if the dimensions of the grid-world are smaller than 7×7, the observation is padded with empty tiles to maintain the consistent observation size. The final observation is therefore a 3D array of shape `(7, 7, 2)`, where the last dimension contains the tile ID and colour ID for each cell in the agent's view.

Within XLand-MiniGrid, the reward function remains the same across all tasks. The reward for completing a task is `1 - 0.9 * (step_count / max_steps)` for success and 0 for failure, meaning agents receive higher rewards for faster task completion. The variable `step_count` represents the number of steps the agent has taken. The variable `max_steps` represents the maximum number of steps in an episode, which depends on the grid-world size and is given by the following formula: `max_steps = 4 * (height * width)`, where `height` and `width` are the dimensions of the grid-world.

The agent only receives a reward upon completing the task. This results in sparse rewards in XLand-MiniGrid, with larger grid-worlds requiring more steps to reach the goal. Therefore, the agent goes through longer periods of not obtaining extrinsic rewards, i.e., feedback. This makes the task more difficult and the agent has to explore longer to solve it. These environments are ideal for testing the exploration capabilities of RL algorithms. Additionally, they are well-suited for meta-RL, as they are relatively computationally inexpensive to run, all tasks share the same action space and observation space, and so they provide an ideal setting for evaluating generalisation.

Experimental Environments

Next, the environments used within XLand-MiniGrid experiments are discussed. The discussion begins with the `Empty` grid-world.

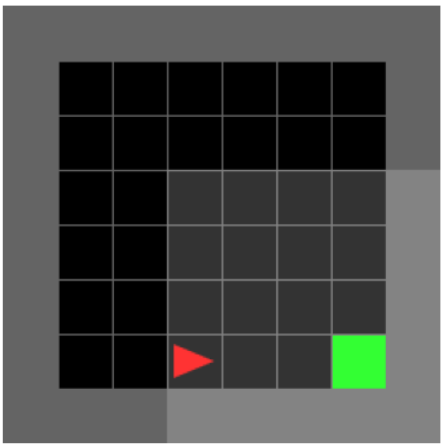


Figure 4.3: The `Empty` grid-world. The red triangle represents the agent and the green square represents the goal position. Taken from [8].

Figure 4.3 illustrates the `Empty` grid-world. The task is to reach the green cell, which represents the goal position. The RL agent's starting position varies at the beginning of each episode. This environment has 4 different sizes: `5x5`, `6x6`, `8x8`, and `16x16`.

The `Unlock` environment involves key-door mechanics. This environment only comes in one size: `6x11`.

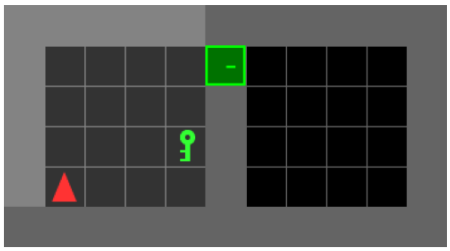


Figure 4.4: The `Unlock` grid-world. The red triangle represents the agent, the green cell represents the door, and the yellow object represents the key. Taken from [8].

Figure 4.4 depicts the `Unlock` environment. In this environment, the RL agent's goal is to obtain the green key by using the action `pick_up` and unlock the door using the action `toggle`. At the start of each episode, the position of the door, agent and key position can change, however, the agent always appears on the same side of the door as the key.

Building on the key mechanics from `Unlock`, the `UnlockPickUp` environment re-

quires that the agent unlock the door to pick up the object. It is an environment with a single size of `6x11`.

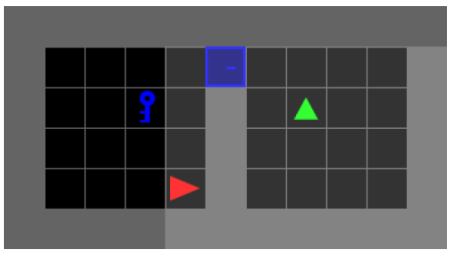


Figure 4.5: The `UnlockPickUp` grid-world. The red triangle represents the agent, the green triangle represents the target object, the blue cell represents the door, and the yellow object represents the key. Taken from [8].

Figure 4.5 illustrates the `UnlockPickUp` environment. In this environment, the task is to obtain the blue key and unlock the blue door. The agent then makes its way to the green object and picks it up. At the start of each episode, the positions of the agent, key, door, and the object to be picked up by the agent can vary. For example, the agent could be tasked with picking up a ball instead of a different object.

The `DoorKey` environments combine navigation challenges with key mechanics. This environment has the same sizes as the `Empty` environment: `5x5`, `6x6`, `8x8`, and `16x16`.

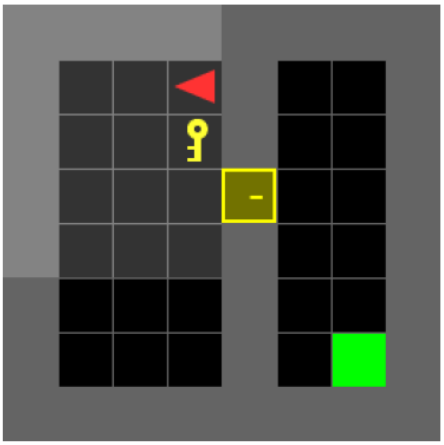


Figure 4.6: The `DoorKey` grid-world. The red triangle represents the agent, the yellow cell represents the door, the green cell represents the goal position, and the yellow object represents the key. Taken from [8].

The objective of this environment is to obtain the yellow key and unlock the yellow door. The agent then makes its way to the goal position. At the start of a new episode, the position of the agent, door and key can change. However, the goal position cannot change.

In the experimental setup, `Empty` and `Unlock` served as training environments for the reward combiner, whilst `UnlockPickUp` and `DoorKey` were used as test environments to evaluate generalisation capabilities. With the task environments defined, how the reward combiner was trained using multi-task ES is next described.

4.3 Training of the reward combiner

The RC's task was to output λ values at each time-step to determine the appropriate intrinsic reward weighting. In order to do this, the RC was designed to consider the lifetime of the RL agent. This was similar to Zheng et al. [49], where the authors also made use of the lifetime return in training their model. The research question falls under many-shot meta-RL. As discussed in section 2.4.1, backpropagation cannot be used to obtain gradients in this setting due to computational constraints. Therefore, ES was used to train the RC.

Each individual in the population trained a single RL agent. Therefore, the objective of the RC during training was to maximise,

$$J_{\text{RC}} = \sum_{t=0}^{T} r_t^e. \tag{4.3.1}$$

Here, T is the total number of training steps to train the RL agent. In other words, the RC was attempting to maximise the sum of extrinsic rewards that the RL agent received during training. This encouraged the RC to output λ values that helped the RL agent explore effectively across multiple episodes during its lifetime, rather than optimising for individual episode performance. As discussed in Section 3.1, considering the agent's entire lifetime enables exploration strategies that carry information across episodes [49].

To achieve generalisation across different tasks, the RC was trained on multiple XLand-MiniGrid environments. Algorithm 5 describes how one can train a neural network using ES. However, this assumes that every individual in the population is being trained on the same task. However, extending this to multi-task training presents challenges. Training every individual in a generation on every task in the task distribution is computationally expensive. Simply alternating between tasks and training the population on a single task per generation is not ideal, as this introduces bias by focusing each gradient update entirely on a single task [9].

Therefore, to train the RC, the multi-task ES approach introduced by Jackson

et al. [9] was used. In multi-task ES, N perturbations are first sampled and their negations (antithetic pairs), where N is the number of individuals in the population. Each antithetic pair is then tested on a randomly sampled task from the task distribution. Then, for the individual in the pair that performs the best, i.e., the one with the higher J_{RC}, a fitness of 1 is assigned and the other individual receives a fitness of 0. In other words, a rank transform, F, is applied to the fitness of each individual in the antithetic pair. This rank transformation normalises the fitness across tasks and as pointed out by Jackson et al. [9] it leads to more stable training. F is defined as:

$$F(x, y) = \begin{cases} 1 & \text{if } x > y \\ 0 & \text{otherwise.} \end{cases} \tag{4.3.2}$$

This rank transformation is applied to all N antithetic pairs.

Therefore, the gradient update becomes [9],

$$\frac{1}{2\sigma}\mathbb{E}_{\epsilon\sim\mathcal{N}(0,I)}\left[\mathbb{E}_{m\sim p(\mathcal{M})}\left[\epsilon \cdot (F(G_m(\theta+\sigma\epsilon), G_m(\theta-\sigma\epsilon)) - F(G_m(\theta-\sigma\epsilon), G_m(\theta+\sigma\epsilon)))\right]\right], \tag{4.3.3}$$

where $p(\mathcal{M})$ is the task distribution, m is the sampled task, $G_m(\theta)$ is the cumulative extrinsic reward obtained by training an RL agent with RC parameters θ on task m, and θ is the current set of parameters. Once the gradient estimate is obtained using equation (4.3.3), the Adam optimiser [58] is used to update the parameters of the RC. Algorithm 6 details how the RC is trained. Figure 4.7 provides a comprehensive overview of the complete multi-task ES training process across a single generation.

Algorithm 6 Multi-task ES for Reward Combiner Training

Require: Task distribution $p(\mathcal{M})$, noise standard deviation $\sigma > 0$, population size N, initial RC parameters θ, training steps T, number of generations M
Ensure: Optimised RC parameters θ^*

Initialise RC parameters θ_0 randomly
Initialise Adam optimiser with learning rate α
for generation $= 1$ to M **do**
    Sample perturbations $\epsilon_1, \ldots, \epsilon_N \sim \mathcal{N}(0, I)$
    Generate antithetic perturbations: $-\epsilon_1, \ldots, -\epsilon_N$
    for $i = 1$ to N **do**
        Sample task $m_i \sim p(\mathcal{M})$
        Randomly initialise BYOL-Explore agent parameters ϕ_i
        Train BYOL-Explore agent from ϕ_i with RC($\theta + \sigma\epsilon_i$) on task m_i and obtain cumulative extrinsic reward $G^+_{m_i}$
        Train BYOL-Explore agent from ϕ_i with RC($\theta - \sigma\epsilon_i$) on task m_i and obtain cumulative extrinsic reward $G^-_{m_i}$
        Apply rank transformation:
            $F^+_i \leftarrow F(G^+_{m_i}, G^-_{m_i})$
            $F^-_i \leftarrow F(G^-_{m_i}, G^+_{m_i})$
    end for
    Compute gradient estimate: $\hat{g} = \frac{1}{2N\sigma} \sum_{i=1}^{N} \epsilon_i (F^+_i - F^-_i)$
    Update parameters: $\theta \leftarrow \text{Adam}(\theta, \hat{g}, \alpha)$
end for
return θ

With the training methodology for the reward combiner established, the discussion now turns to the experimental design for evaluating its performance against baseline algorithms.

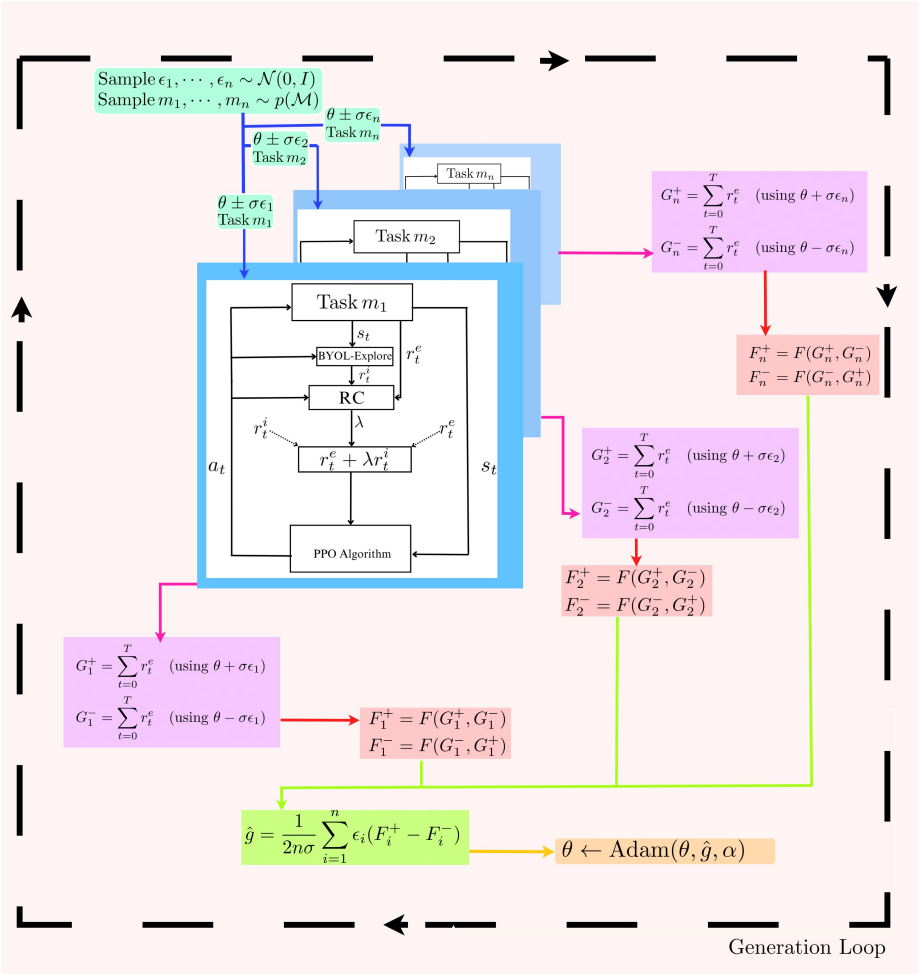


Figure 4.7: Multi-task ES [9] training process for the reward combiner (RC) across a single generation. The process begins by sampling N perturbations $\epsilon_1, \ldots, \epsilon_n \sim \mathcal{N}(0, I)$ where σ is the fixed standard deviation, and tasks $m_1, \ldots, m_n$ from the MDP distribution $p(\mathcal{M})$ (training environments). For each antithetic pair, two separate RL agents are trained: one using RC parameters $\theta + \sigma\epsilon_i$ and another using parameters $\theta - \sigma\epsilon_i$, where θ represents the current RC parameters. Both agents train on the same task m_i using the detailed process shown for Task m_1. The inner loop demonstrates how the RC combines extrinsic rewards r_t^e and intrinsic rewards r_t^i from BYOL-Explore to produce λ-weighted rewards $(r_t^e + \lambda r_t^i)$ for the PPO algorithm. After T training steps, each agent produces a return $G = \sum_{t=0}^{T} r_t^e$ (sum of extrinsic rewards). These returns undergo rank transformation F (Equation 4.3.2) to compute fitness values, which are used to estimate the gradient $\hat{g}$. The RC parameters θ are then updated using the Adam optimiser with learning rate α. This process repeats for a user-specified number of generations.

4.4 Experiments

This section details the experiments conducted. Each experiment varied in the environment used. For each experiment, to compare the different algorithms, their average episode return at the end of training was used.

In the experiments, the following labelling convention was used:

- **BYOL-Explore** and **RND**: When mentioned in the context of a specific environment, these refer to the algorithms using the optimal λ value found through hyperparameter search within that same environment.
- **BYOL-Explore-[Environment]**: This refers to the BYOL-Explore algorithm using the optimal λ value found in [Environment] but applied to a different target environment. For example, if experiments are conducted in `Unlock`, then BYOL-Explore refers to using the optimal λ value discovered in `Unlock`, whilst BYOL-Explore-`Empty-16x16` refers to using the optimal λ value found in `Empty-16x16` but applied to the `Unlock` environment.
- **RC-NoAction**: This refers to the RC that does not take actions as input but just the normalised rewards. RC-NoAction was trained on `Unlock` and `Empty-16x16`.

Experiment 1: Fixed λ Generalisation The first experiment was conducted to examine how one λ found on one task generalised to another task and to demonstrate the limitations of fixed λ values across tasks. In this experiment, the focus was on BYOL-Explore since this was the algorithm used to provide intrinsic rewards to the RC. In particular, how the λ optimised for the `Empty-16x16` grid-world performed on the `Unlock` grid-world was examined. The performance of BYOL-Explore was compared to BYOL-Explore-`Empty-16x16` and to the base performance of PPO. A similar experiment was conducted in the reverse direction, comparing BYOL-Explore-`Unlock` to BYOL-Explore and standard PPO in `Empty-16x16`. The purpose of this experiment was to examine how well different λ values optimised for specific tasks generalised to other tasks. The reason for conducting the reverse direction was to determine if the environment chosen for the hyperparameter sweep was important.

Experiment 2: Cross-size Generalisation The second set of experiments aimed to determine whether the reward combiner could generalise across different grid-sizes, i.e., whether the RC could adapt to different levels of reward sparsity. This experiment addressed Sub-question 1. The focus was on the `DoorKey` grid-world environment. The RC was tested on all four sizes of the `DoorKey` environment against the curiosity-driven baselines, BYOL-Explore and RND, as well as standard PPO.

Experiment 3: Cross-task Generalisation In the next set of experiments, Sub-question 2 was addressed. The RC was tested on `UnlockPickUp`. This experiment tested whether the RC could generalise to tasks where the reward-associated action differed from those seen during training. Unlike the training environments where navigation-based actions led to rewards, `UnlockPickUp` required the `pick_up` action to achieve the extrinsic reward.

Experiment 4: Ablation Studies The final experiment was conducted to test different input variants into the reward combiner on all the `DoorKey` grid sizes and the `UnlockPickUp` environment. From the above experiments, it was observed that to test generalisation across different tasks, the action was inputted into the reward combiner as well, along with the normalised rewards. To determine if including action inputs was necessary, the performance of RC and RC-NoAction in the test environments was compared.

Statistical Analysis

The statistical analysis procedures used to determine statistically significant differences when comparing algorithms are now described. The average episode return is the metric used to compare the algorithms. 30 random seeds were used for each algorithm to obtain the average episode return at the end of training. To capture the certainty of the mean episode return, 95% confidence intervals were calculated. The confidence intervals were calculated using the percentile bootstrap method with 10,000 resamples.

Agarwal et al. [59] and Patterson et al. [56] recommend against using p-values when comparing algorithms because they are often misinterpreted. As highlighted by Patterson et al. [56], it is common to view the p-value as the probability that the null hypothesis is true (there is no difference between algorithm A and algorithm B) when it actually represents the probability of observing the data given that the null hypothesis is true, and it does not indicate whether the null hypothesis is true.

Patterson et al. [56] suggest using confidence intervals on the difference. This approach was used when comparing two algorithms. For example, when two algorithms A and B each with 30 random seeds are compared, each algorithm uses the same initialisation key to create the same set of 30 random seeds. Element-wise differences are calculated between the algorithm results ($A_i - B_i$ for each random seed i) to obtain 30 difference values. The confidence interval of the difference is then calculated using the same percentile bootstrap method with an error rate of ϵ between A and B to obtain a range of their performance difference and the uncertainty of that range. If this confidence interval excludes zero, a statistically significant difference between the algorithms is concluded. The significance of the difference is then reported along with the certainty of the difference between the algorithms.

In the experiments, multiple comparisons were performed. In experiment 1, all

pairwise comparisons were performed between algorithms, whereas in Experiments 2 and 3, the RC was compared to each baseline individually. Experiment 4 involved a single comparison between RC and RC-NoAction. Each comparison was treated independently, but this means the error rate across all comparisons, κ, grows. Therefore, the Bonferroni correction was performed [60]. κ was targeted to remain at 0.05 across all comparisons within each experiment. To ensure that it remained at this level when performing multiple comparisons, it was ensured that the confidence interval of a single comparison had an error rate of $\frac{\kappa}{n}$ where n is the number of comparisons performed.

In each experiment, κ was targeted to remain at 0.05. Experiments 1 to 3 each contained 3 comparisons. Therefore, the error rate of a single comparison was $\frac{\kappa}{3}$. Therefore, the confidence intervals of a single comparison in experiments 1 to 3 were 98.33%. Experiment 4 only contained a single comparison, so no correction was needed.

With the experimental design and implementation details now established, the next Chapter presents the experimental results.

Chapter 5

Results and Discussion

This Chapter presents the empirical findings from our experiments evaluating how an agent that learns to dynamically output λ performs compared to manually fine-tuned curiosity-driven algorithms and standard PPO across grid-world tasks involving navigation and/or door-key interaction mechanics. We conducted the experiments in XLand-MiniGrid environments and address our two sub-questions about generalisation across different grid sizes and different task objectives. We also present plots of the RC's λ output in the `DoorKey-5x5` environment.

We also examine the implications of our experimental findings and place them within the broader context of the literature. We do this by analysing why fixed λ values fail to generalise across environments, and how the empirical findings validate the necessity of environment-specific λ tuning observed in prior work. Additionally, the minimal improvements observed from curiosity-driven baselines over PPO on test environments are discussed. To understand how the RC was able to generalise, its behaviour observed in the `DoorKey-5x5` environment is examined. The discussion also covers how ES enabled optimisation over the full agent lifetime. Finally, the limitations of this approach are presented.

5.1 Experiment Results

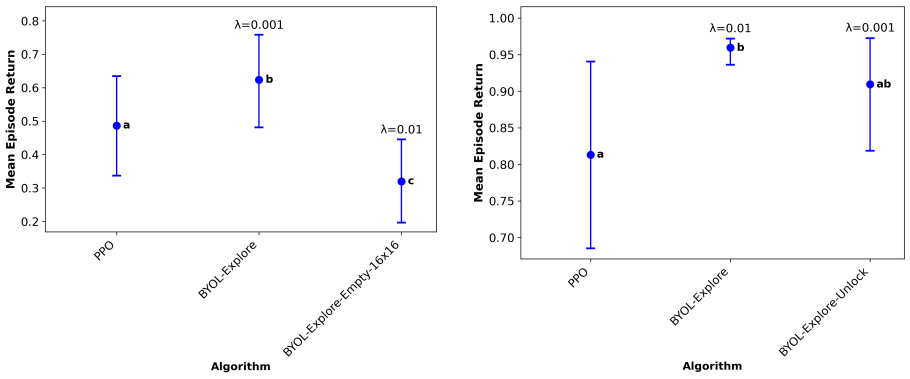
Experiment 1: Fixed λ Generalisation

Before evaluating the RC's performance, we first demonstrate the limitations of using fixed λ values across different environments. In Experiment 1, we performed pairwise comparisons between all algorithms using letters to indicate whether the difference in performance is significant. In Figure 5.1 algorithms sharing the same letter mean that the difference in performance is not significant, whilst algorithms with different letters mean that the difference in performance is significant. Recall from Section 4.4 that we determine whether the difference is

significant by checking whether the 98.33% confidence intervals of the differences in performance contain zero or not.

Figure 5.1a shows the performance of PPO, BYOL-Explore, and BYOL-Explore-`Empty-16x16` in the `Unlock` environment. Whereas, Figure 5.1b shows the performance of PPO, BYOL-Explore, and BYOL-Explore-`Unlock` in the `Empty-16x16` environment. Notably, the optimal λ values differ substantially between environments, with $\lambda = 0.001$ for `Unlock` versus $\lambda = 0.01$ for `Empty-16x16`. This demonstrates that these environments require very different intrinsic reward weightings for optimal performance.

In the `Unlock` environment, both BYOL-Explore and PPO significantly outperform BYOL-Explore-`Empty-16x16`, showing that fixed λ values do not always generalise well across environments.



(a) `Unlock` environment (b) `Empty-16x16` environment

Figure 5.1: Average final episode return at the end of training across 30 agents with 95% confidence intervals. (a) `Unlock` environment. (b) `Empty-16x16` environment. BYOL-Explore uses the optimal λ value found in the respective environment, whilst BYOL-Explore-`Empty-16x16` uses the optimal λ value found in `Empty-16x16` but applied to `Unlock`, and BYOL-Explore-`Unlock` uses the optimal λ value found in `Unlock` but applied to `Empty-16x16`. Optimal λ values shown above algorithm names. Letters indicate statistical significance groups: algorithms sharing the same letter do not have a statistically significant difference in performance, whilst algorithms with different letters have a statistically significant difference.

However, in the `Empty-16x16` environment, we observe better transfer from BYOL-Explore-`Unlock`. Although BYOL-Explore outperforms BYOL-Explore-`Unlock`, the difference in their performance is not significant. BYOL-Explore-`Unlock` achieves higher mean performance than PPO, though this difference is not significant. These results highlighted the difficulty in finding optimal λ values that work well across different environments within XLand-MiniGrid.

Experiment 1 demonstrated why per-environment λ optimisation is necessary in practice. The results from this experiment indicated that the λ value found in one environment does not always transfer well to another. The `Empty-16x16` results suggest that one might find an environment to identify a λ value that works reasonably across environments, though not optimally. However, how one identifies such an environment remains unclear.

Transferring λ values between environments can result in significant performance degradation. Experiment 1 provides empirical support for the approach used by Zheng et al. [47]. As mentioned in Section 3.1, Zheng et al. [47] optimised λ separately for each training environment. As the number of environments increases, this per-environment tuning becomes computationally expensive. This motivates the need for a dynamic approach that can adapt λ automatically across different tasks.

Experiment 2: Achieving Generalisation Across Different Grid Sizes

Experiment 2 was designed to test if the RC could generalise across different sizes of the `DoorKey` environment, which addresses sub-question 1. In this experiment we compared the RC to each of the baselines: PPO, RND and BYOL-Explore. Figure 5.2 shows the macro-performance across all `DoorKey` grid-world sizes, calculated by averaging episode returns across all 120 agents (30 agents $\times$ 4 grid-sizes). The λ values for BYOL-Explore and RND were optimised for each grid-world size. The RC outperformed all baselines not only in macro-performance but also on each individual grid-world size, as shown in Figure 5.2 and the individual plots in Appendix E. The difference between the RC and each of the baselines is significant, confirming our hypothesis.

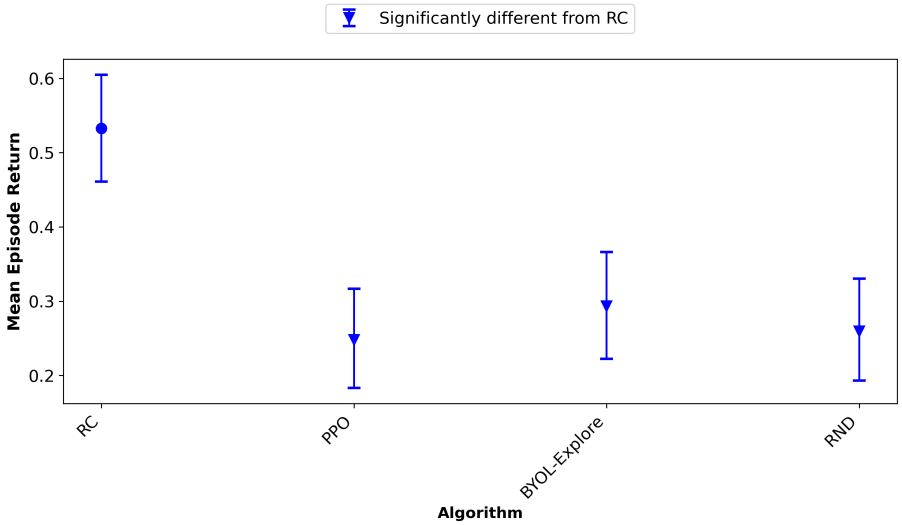


Figure 5.2: Macro-performance comparison across all `DoorKey` grid-worlds, showing the average final episode return across 30 agents with 95% confidence intervals for different algorithms.

These results support Hypothesis 1. The RC not only meets the expectation of performing as well as the baselines when generalising across different grid-world sizes but actually outperforms both manually fine-tuned curiosity-driven algorithms and standard PPO.

Experiment 3: Achieving Generalisation Across Different Tasks

This experiment addresses sub-question 2 by comparing the RC's performance against baselines when generalising to an unseen task with a different objective from training. Figure 5.3 shows the average episode returns of the baselines and the RC in the `UnlockPickUp` environment. In the `UnlockPickUp` environment the agent is tasked with picking up an object to obtain the extrinsic reward. The RC did not see the `pick_up` action associated with extrinsic rewards during training. During training, the RC observed extrinsic rewards associated with navigation actions (`move_forward`) in `Empty-16x16` and unlocking doors (`toggle`) in `Unlock`. However, in `UnlockPickUp`, the extrinsic reward is obtained when the agent unlocks the door and picks up the object.

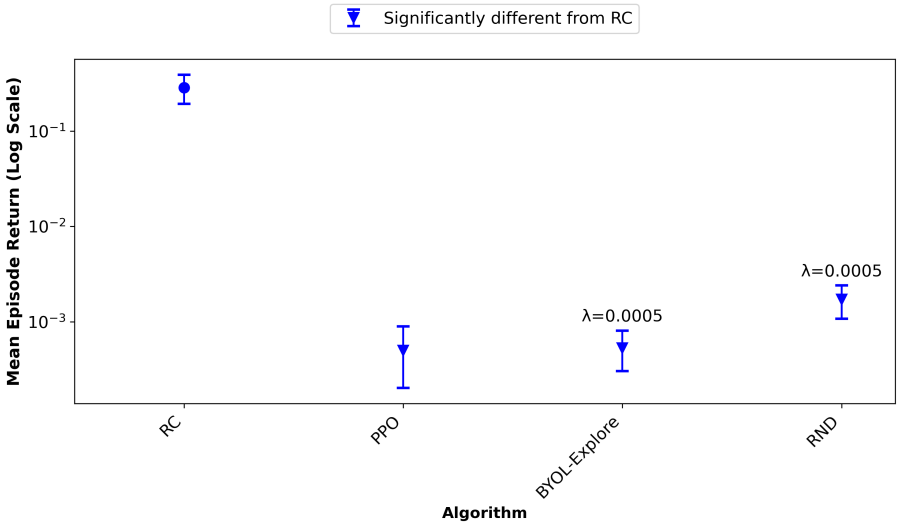


Figure 5.3: Average final episode return at the end of training in the `UnlockPickUp` grid-world, across 30 agents with 95% confidence intervals. Results are presented using a logarithmic scale. Optimal λ values shown above BYOL-Explore and RND.

Figure 5.3 highlights that the RC again achieves a higher average episode return, and that the difference is significant. The performance difference is substantial, with the RC achieving returns approximately two orders of magnitude higher than the baselines. These results support Hypothesis 2. The RC not only meets the expectation of performing as well as the baselines when generalising to an unseen task where the objective differs from training but actually outperforms the baselines.

The results from Experiments 2 and 3 showed that the RC can dynamically output appropriate λ values across unseen environments with navigation and door-key mechanics. This successfully addresses our main research question. The RC outperformed manually fine-tuned baselines across both different grid-world sizes and different task objectives.

These results are particularly noteworthy as neither `DoorKey` nor `UnlockPickUp` were included in the RC's training environments. The RC's ability to generalise across both different grid sizes and different task objectives within environments sharing navigation and door-key mechanics supports both our hypotheses and highlights the potential for eliminating environment-specific λ searches when deploying curiosity-driven RL across multiple environments.

Minimal Improvements from Curiosity-Driven Baselines

The results from Experiments 2 and 3 showed that RND and BYOL-Explore did not demonstrate statistically significant improvements over PPO. This result is surprising, as one would expect curiosity-driven baselines to significantly improve performance in sparse reward environments.

The `UnlockPickUp` and `DoorKey` environments are more challenging than the `Unlock` and `Empty` training environments, as in our initial experiments we saw that PPO did not perform as well on the test environments compared to the training environments. Despite this, we used the same training duration of 5 million time-steps for both the training and test environments. We hypothesise that with more time-steps, we could potentially observe the curiosity-driven baselines provide statistically significant improvements over PPO. Interestingly, the RC achieved statistically significant improvements over PPO and the curiosity-driven baselines, despite also training for only 5 million time-steps.

Experiment 4: Ablation Studies

This experiment was designed to see how important action inputs were to the reward combiner generalising to unseen tasks. In this experiment we compared the RC to RC-NoAction on the testing environments. RC-NoAction simply obtained the normalised intrinsic and extrinsic rewards as input.

Table 5.1: Ablation study results in `UnlockPickUp` environment: Average final episode return across 30 agents with 95% confidence intervals (CIs).

Algorithm	Mean Episode Return (95% CI)
RC	0.284 (0.193, 0.389)
RC-NoAction	0.000* (0.000, 0.000)

* Statistically significantly different from the RC.

Table 5.1 and Table 5.2 highlight that the RC achieved a higher average episode return and that the difference was significant. RC-NoAction completely fails to generalise. This highlights that using rewards alone is insufficient for generalising to unseen tasks. The action inputs are crucial and allows the RC to adjust λ accordingly.

Table 5.2: Ablation study results across all `DoorKey` grid-world sizes: Average final episode return across 30 agents with 95% confidence intervals (CIs).

Algorithm	Mean Episode Return (95% CI)
RC	0.533 (0.460, 0.605)
RC-NoAction	0.012* (0.010, 0.014)

* Statistically significantly different from the RC.

The ablation study highlighted that actions are crucial to the RC performing

well on the test environments. We believe that actions provide task context to guide the RC in determining λ, i.e., the actions indicate what the agent is trying to achieve and what the task is. RC-NoAction lacks this context and cannot distinguish between different types of environments and adjust λ accordingly.

Having demonstrated the RC's superior performance and the importance of action inputs, we now examine the RC's output i.e., the λ values. The focus is on the λ values output in the `DoorKey-5x5` environment, where we observed the most interpretable plots.

5.2 The RC's Output

The RC's Episode-level Output

In analysing the RC's output in certain episodes, we noticed that it output high λ values at the door position in the `DoorKey-5x5` environment. This trend was not as noticeable in the larger grid-world sizes of `DoorKey` and `UnlockPickUp`, hence the focus is on the `DoorKey-5x5` environment.

Figure 5.4 shows a split heatmap from a particular episode where the RC output high λ values at the door position. The heatmap shows average metric values in each grid-world cell that the agent visited during this episode. The heatmap displays the grid-world layout with walls shown in grey and floor tiles in lighter colours. Each grid cell splits into two halves: the bottom half shows the normalised intrinsic rewards, whilst the top half shows the λ values. Emojis mark the key and door positions for clarity, and a checkered flag indicates the goal position with the extrinsic reward value (Ext: 0.96) obtained when the task was solved. The goal position appears white because the environment automatically resets upon reaching the terminal state, meaning no λ or reward statistics are recorded at this position.

In this particular episode, the RC outputs a λ value of 0.98 at the door position, which represents the highest value observed in this episode. There were instances where λ values remained near zero throughout the entire grid-world. This occasional but pronounced trend at door positions led us to investigate how λ and normalised intrinsic rewards evolve at specific positions over training.

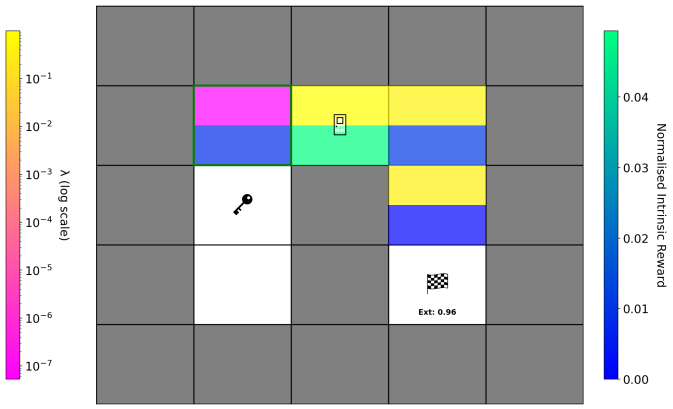


Figure 5.4: Split heatmap showing average normalised intrinsic rewards (bottom half of each cell) and average λ values (top half of each cell) from a single episode in `DoorKey-5x5` for a single RL agent. Values represent averages across all visits to each cell during the episode. Grey areas represent walls, white areas represent cells not visited by the agent, emojis indicate key and door positions, and the checkered flag marks the goal position. The extrinsic reward value (Ext: 0.96) shows the extrinsic reward obtained when the task was completed. The goal position appears white because the environment automatically resets upon task completion, preventing recording of statistics at this terminal position.

The λ Values and Intrinsic Reward at Different Grid Positions

We investigated the average λ value at the door position and compared that to the average λ value at an ordinary floor tile position. The Figures in this section use data from a single RL agent, where each update step represents the average across all parallel environments in PPO's vectorised architecture. We applied a Simple Moving Average (SMA) with a window size of 30 to smooth the data.

Figure 5.5 shows the average λ value and the normalised intrinsic reward at the door position over the course of training. The figure highlights that the λ values output by the RC appear to increase as the normalised intrinsic rewards decrease. Despite some initial fluctuations, the general trend shows λ increasing from approximately 0 to 0.08 over the training period, whilst the normalised intrinsic rewards decrease, plateau, and then decrease to zero.

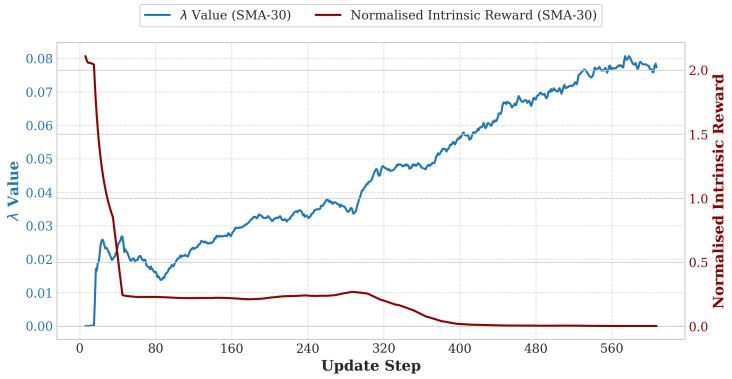


Figure 5.5: Simple Moving Average (window size 30) of the average λ and the average normalised intrinsic reward at the door position from one RL agent in `DoorKey-5x5`.

Figure 5.6 shows the weighted normalised intrinsic reward and episode return at the door position over the course of training. We observe that the weighted normalised intrinsic reward initially increases, then dips, increases again, but as the episode return plateaus near optimal performance, the weighted normalised intrinsic reward begins to decrease.

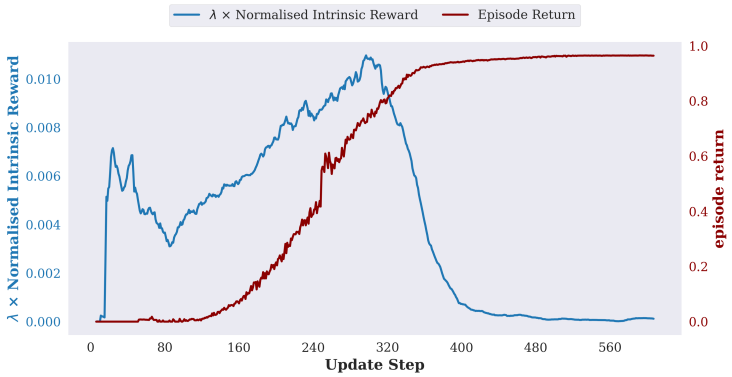


Figure 5.6: The weighted normalised intrinsic reward ($\lambda\times$ normalised intrinsic reward) and episode return at the door position from one RL agent in `DoorKey-5x5`.

We also plotted the average λ value and the normalised intrinsic reward at ordinary floor positions over the course of training in Figure 5.7. In contrast to the door position, the λ values at floor positions show a different trend: an initial increase followed by a decrease that eventually plateaus at a lower level, ranging from 0 to approximately 0.03. The normalised intrinsic rewards at floor positions decay faster compared to the door position.

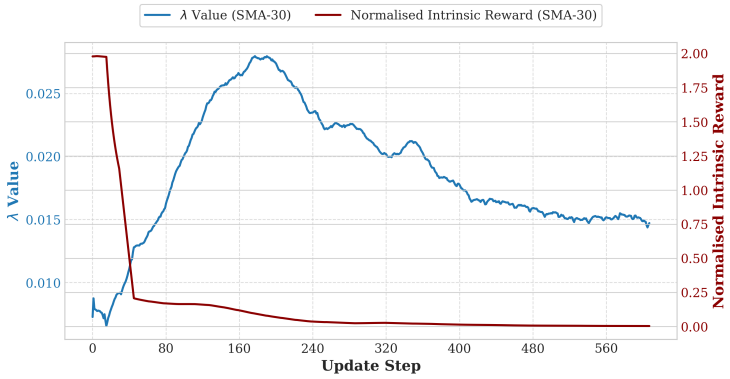


Figure 5.7: Simple Moving Average (window size 30) of the average λ and the average normalised intrinsic reward at floor positions from one RL agent in `DoorKey-5x5`.

Figure 5.5 and Figure 5.7 also show that the normalised intrinsic reward decays more slowly compared to an ordinary floor position. We suspect this occurs because when the agent reaches the door position, it can now see the room that the door was hiding, causing the tile IDs to change from being classified as empty to being classified as a floor or the goal position. This sudden change in the observable environment makes it harder to predict what happens at the door position, which we suspect explains why the intrinsic rewards (based on prediction error) decay more slowly there.

Figures 5.5 and 5.7 raise an intriguing question: how does the RC know it is at the door position if it doesn't receive state information as input? We hypothesise that the RC learns to recognise specific action sequences, particularly the sequence of `toggle` (to unlock the door) followed by `move_forward`, as indicators of contextually important moments. Further research would be needed to validate this hypothesis.

Figure 5.8 shows the weighted normalised intrinsic reward at ordinary floor positions, which exhibits a simpler plot compared to Figure 5.6. The weighted normalised intrinsic reward plateaus briefly before gradually decreasing to zero over time.

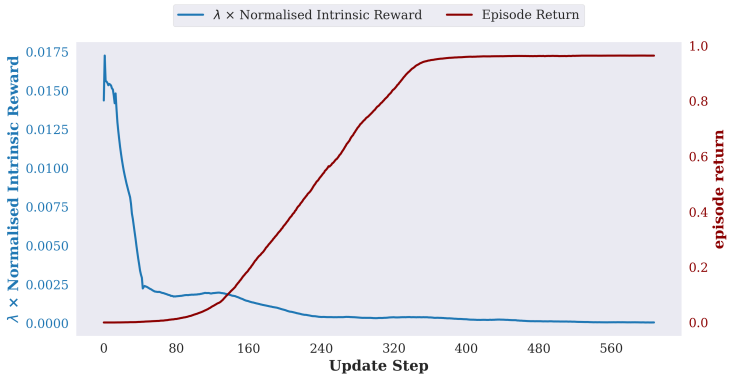


Figure 5.8: The weighted normalised intrinsic reward ($\lambda\times$ normalised intrinsic reward) and episode return at floor positions from one RL agent in `DoorKey-5x5`.

Comparing Figure 5.5 and Figure 5.7 reveals distinct temporal trends at different position types. The door position shows a sustained increase in λ values reaching higher magnitudes (up to 0.08), whilst floor positions exhibit more modest λ values (up to 0.03) with an eventual plateau. We observed this specific trend primarily in the `DoorKey-5x5` environment and not consistently across other test environments.

The trend of the weighted normalised intrinsic reward at the door position in Figure 5.6 is noteworthy as it aligns with what we might expect from intrinsic motivational systems. As the agent is learning, the intrinsic rewards increase. However, when performance nears optimal levels, the intrinsic rewards diminish. This behaviour avoids the agent becoming fixated on intrinsic rewards, which is what occurs in the Noisy TV Problem [41].

These behavioural patterns provide important insights into our main research question. The observation that the RC exhibits different λ patterns at different positions (higher values at doors vs. lower values at floor tiles) shows that it dynamically outputs appropriate intrinsic reward weightings rather than learning a static strategy. However, it is important to note that these trends were only clearly observable in the `DoorKey-5x5` environment and were not consistently observed across other test environments. Therefore, we cannot establish generalised behavioural trends for the RC based on these observations alone.

Optimising the Lifetime Return with ES

Our results from `UnlockPickUp` and `DoorKey` environments demonstrate that ES is a viable approach for optimising lifetime return objectives in many-shot meta-RL. Unlike prior work on lifetime intrinsic rewards [49] which required bootstrap estimates due to backpropagation constraints, our ES approach enabled direct optimisation of complete agent lifetimes, demonstrating that full lifetime returns

can be effectively optimised in practice.

In preliminary experiments, we found that applying the rank transformation from Jackson et al. [9] provided stable training, consistent with their findings in multi-task ES settings.

Limitations

While our approach shows promise for reducing computational expense in hyperparameter tuning, it has several limitations that need addressing.

First, we focused on XLand-MiniGrid environments with navigation and/or door-key mechanics. It remains unclear how well the RC generalises to other mechanics within XLand-MiniGrid, such as `Memory` environments where agents must remember initial objects, or more complex door-key scenarios like `BlockedUnlockPickUp` where objects block doors.

Second, the RC requires environments to share the same action space. This constraint becomes apparent when considering domains such as Brax [54], where robotic agents have different numbers of limbs and joints, resulting in varied action spaces. This limitation aligns with observations by Alet et al. [50] and Hospedales et al. [43] that meta-learning neural network weights approaches often face challenges when generalising across environments in different domains.

Third, our training environment selection relied on heuristic assessment rather than principled guidelines. We selected `Unlock` and `Empty-16x16` based on observed navigation and door-key mechanics that appeared relevant to our target test environments (`DoorKey` and `UnlockPickUp`). However, we lack systematic methods for determining which training environments enable optimal generalisation to new tasks.

Finally, our implementation was not end-to-end in JAX [57]. While individual RL agent training with the RC was implemented in JAX, we could not implement the task distribution sampling component in JAX, which likely reduced computational efficiency during the RC training.

These results demonstrate the RC's performance across all test environments and provide insight into its λ output in the `DoorKey-5x5` environment. The findings answer both research sub-questions and support our hypotheses, showing that the RC can generalise across different grid sizes and task objectives while consistently outperforming manually fine-tuned baselines. In the next Chapter we conclude our work.

Chapter 6

Conclusions

This journey began with highlighting that in RL, it is not always possible to design environments with dense reward functions. There are environments with sparse reward functions where RL agents often struggle to learn effectively. In such sparse reward environments, agents need to explore in order to solve the task, but they struggle to explore efficiently in these environments. Curiosity-driven exploration algorithms are designed to help agents explore efficiently by providing an additional reward known as the intrinsic reward to the RL agent. The intrinsic reward and extrinsic reward are typically combined using a weighted sum given as:

$$\hat{r} = r_e + \lambda r_i$$

where λ is the intrinsic reward weighting.

However, the λ value is a hyperparameter that often requires a hyperparameter sweep to find the optimal value. When changing tasks, this process must typically be repeated for each new environment. This remains true even when tasks are within the same domain, such as grid-world tasks. We decided to focus on XLand-MiniGrid, which is a collection of discrete grid-world environments that are sparse rewarded and implemented in JAX.

We first demonstrated this problem empirically, showing that λ values optimised for one environment (λ=0.01 for `Empty-16x16`) performed significantly worse when applied to different environments (`Unlock`), and that the difference in performance was significant. While the reverse direction (`Unlock` to `Empty-16x16`) also showed performance degradation, the difference was not significant, suggesting some environments may enable better transfer than others.

This motivated our main research question:

> **Research Question:** How does the performance of an agent that learns to dynamically output intrinsic reward weightings (λ) compare to manually fine-tuned curiosity-driven algorithms and standard PPO across grid-world tasks involving navigation and/or door-key interaction mechanics?

To address this question, we developed a reward combiner (RC), a recurrent neural network, that dynamically outputs λ values by observing normalised intrinsic rewards, extrinsic rewards, and agent actions. We trained the RC using multi-task ES on `Unlock` and `Empty-16x16` environments, with the fitness function being the sum of extrinsic rewards during the agent's training lifetime.

We divided this main research question into two sub-questions to test generalisation across different aspects: adapting to different grid sizes of an unseen task, and adapting to a task with a different objective from those encountered during training. The first sub-question and hypothesis are given below.

> **Sub-question 1:** How does the meta-learning agent's performance compare with fine-tuned curiosity-driven algorithms and standard PPO when evaluating across different grid-world sizes on an unseen task?

> **Hypothesis 1:** The meta-learning agent will perform as well as the baselines when evaluating across different grid-world sizes on an unseen task with navigation and/or door-key interaction mechanics.

Our experiments on the `DoorKey` environments across all grid sizes `(5x5, 6x6, 8x8, and 16x16)` showed that the RC outperformed the manually fine-tuned BYOL-Explore, RND, and standard PPO, and the difference in performance was significant. The RC achieved superior performance not only in macro-performance across all grid sizes but also on each individual grid size. This exceeded our hypothesis because the RC not only performed as well as the baselines but outperformed them, with the difference in performance being significant across all comparisons.

We then moved to the second sub-question and hypothesis.

> **Sub-question 2:** How does the meta-learning agent's performance compare with fine-tuned curiosity-driven algorithms and standard PPO when evaluating on an unseen task where the objective differs from those seen during training?

Hypothesis 2: The meta-learning agent will perform as well as the baselines when evaluating on an unseen task with navigation and/or door-key interaction mechanics where the objective differs from those seen during training.

Our experiments on the `UnlockPickUp` environment demonstrated that the RC successfully generalised to a task with a different objective (pick-up the object) from those observed during training. The RC outperformed all the baselines, and the difference in performance was significant. This exceeded our hypothesis because the RC substantially outperformed the curiosity-driven baselines and standard PPO.

Additional findings from our ablation studies revealed that action inputs are crucial for generalisation. We developed RC-NoAction, which is the RC trained without action inputs. RC-NoAction completely failed to generalise to unseen tasks, highlighting that the RC relies on action sequences to identify task-specific contexts and adjust λ accordingly. Our behavioural analysis revealed interesting trends in the `DoorKey-5x5` environment, where the RC output higher λ values at door positions compared to ordinary floor positions. When examining the weighted intrinsic reward at door positions, we observed trends that align with expected intrinsically motivated system behaviour, as the weighted intrinsic reward began decreasing when performance plateaued near optimal levels. However, these interpretable trends were not consistently observable across other test environments, and so we could not establish generalised behavioural trends.

Despite these promising results, our approach has several limitations. We focused on XLand-MiniGrid environments with navigation and/or door-key mechanics, and it remains unclear how well the RC generalises to other mechanics within XLand-MiniGrid or to entirely different domains. Our approach requires environments to share the same action space, limiting applicability to domains like Brax where different environments have varying action spaces. Additionally, our training environment selection relied on heuristic assessment rather than principled guidelines, and our implementation was not end-to-end in JAX, particularly the task distribution sampling component, which limited computational efficiency.

Future Work

Several future research directions emerge from this work's limitations. First, implementing task distribution sampling in JAX would enable fully end-to-end optimisation, addressing the computational efficiency challenges we encountered and accelerating the meta-learning process significantly.

Second, developing a more systematic approach to selecting training environments. This is particularly important when attempting to generalise across a large range of environments, and represents a crucial methodological improve-

ment for determining which training environments enable optimal generalisation to new tasks.

Third, expanding the approach to cover broader task types within XLand-MiniGrid would test whether reward combination can generalise across the full domain rather than just navigation and door-key mechanics. This could include memory-based tasks and other mechanics within the environment suite. Additionally, evaluating the RC with other curiosity algorithms such as RND would determine whether our approach can work with any curiosity-driven exploration algorithm.

Fourth, conducting detailed computational cost analysis comparing our approach against traditional hyperparameter sweeping would quantify the practical efficiency gains our method enables.

Finally, investigating how to handle environments with different action spaces, such as in Brax, represents the next challenge for expanding beyond grid-world domains. This would require addressing the fundamental constraint that our approach currently requires environments to share the same action space.

All in all, the limitations of our approach highlight that there is still work to be done before we can eliminate the need to perform hyperparameter searches across different environments. However, our results provide promising evidence that there is potential in our methodology for addressing this computational challenge.

Bibliography

[1] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym, June 2016. URL https://arxiv.org/abs/1606.01540.

[2] Satinder Singh, Richard L. Lewis, Andrew G. Barto, and Jonathan Sorg. Intrinsically Motivated Reinforcement Learning: An Evolutionary Perspective. *IEEE Transactions on Autonomous Mental Development*, 2(2):70–82, June 2010. ISSN 1943-0612. doi: 10.1109/TAMD.2010.2051031. URL https://ieeexplore.ieee.org/document/5471106.

[3] Satinder Singh, Andrew G. Barto, and Nuttapong Chentanez. Intrinsically motivated reinforcement learning. In *Proceedings of the 17th International Conference on Neural Information Processing Systems*, NIPS'04, pages 1281–1288, Cambridge, MA, USA, December 2004. MIT Press. URL https://proceedings.neurips.cc/paper/2004/hash/4be5a36cbaca8ab9d2066debfe4e65c1-Abstract.html.

[4] Deepak Pathak, Pulkit Agrawal, Alexei A. Efros, and Trevor Darrell. Curiosity-driven Exploration by Self-supervised Prediction. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 2778–2787. PMLR, 06–11 Aug 2017. URL https://proceedings.mlr.press/v70/pathak17a.html.

[5] Thomas Simonini. Curiosity Driven Learning through Random Network Distillation, 2019. URL https://medium.com/data-from-the-trenches/curiosity-driven-learning-through-random-network-distillation-488ffd8e5938.

[6] Zhaohan Guo, Shantanu Thakoor, Miruna Pislar, Bernardo Avila Pires, Florent Altché, Corentin Tallec, Alaa Saade, Daniele Calandriello, Jean-Bastien Grill, Yunhao Tang, Michal Valko, Remi Munos, Mohammad Gheshlaghi Azar, and Bilal Piot. BYOL-Explore: Exploration by Bootstrapped Prediction. In *Advances in Neural Information Processing Systems*, volume 35, pages 31855–31870. Curran Associates, Inc., 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/ced0d3b92bb83b15c43ee32c7f57d867-Paper-Conference.pdf.

[7] Matthew Botvinick, Sam Ritter, Jane X. Wang, Zeb Kurth-Nelson, Charles Blundell, and Demis Hassabis. Reinforcement Learning, Fast and Slow. *Trends in Cognitive Sciences*, 23(5):408–422, May 2019. ISSN 13646613. doi: 10.1016/j.tics.2019.02.006. URL https://www.sciencedirect.com/science/article/pii/S1364661319300610.

[8] Alexander Nikulin, Vladislav Kurenkov, Ilya Zisman, Viacheslav Sinii, Artem Agarkov, and Sergey Kolesnikov. XLand-MiniGrid: Scalable Meta-Reinforcement Learning Environments in JAX. In *Intrinsically-Motivated and Open-Ended Learning Workshop, NeurIPS2023*, 2023. URL https://openreview.net/forum?id=xALDC4aHGz.

[9] Matthew Thomas Jackson, Chris Lu, Louis Kirsch, Robert Tjarko Lange, Shimon Whiteson, and Jakob Nicolaus Foerster. Discovering temporally-aware reinforcement learning algorithms. In *International Conference on Learning Representations*, volume 12, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/file/3908cadfcc99db12001eafb1207353e9-Paper-Conference.pdf.

[10] David Silver. Lectures on Reinforcement Learning, 2015. URL https://www.davidsilver.uk/teaching/.

[11] Stuart Russell. If We Succeed. *Daedalus*, 151(2):43–57, 05 2022. ISSN 0011-5266. doi: 10.1162/daed_a_01899. URL https://doi.org/10.1162/daed_a_01899.

[12] Shahab Shamshirband, Mahdis Fathi, Abdollah Dehzangi, Anthony Theodore Chronopoulos, and Hamid Alinejad-Rokny. A review on deep learning approaches in healthcare systems: Taxonomies, challenges, and open issues. *Journal of Biomedical Informatics*, 113:103627, 2021. ISSN 1532-0464. doi: https://doi.org/10.1016/j.jbi.2020.103627. URL https://www.sciencedirect.com/science/article/pii/S1532046420302550.

[13] Ilan Price, Alvaro Sanchez-Gonzalez, Ferran Alet, Tom R. Andersson, Andrew El-Kadi, Dominic Masters, Timo Ewalds, Jacklynn Stott, Shakir Mohamed, Peter Battaglia, Remi Lam, and Matthew Willson. Probabilistic weather forecasting with machine learning. *Nature*, 637(8044):84–90, January 2025. ISSN 1476-4687. doi: 10.1038/s41586-024-08252-9. URL https://www.nature.com/articles/s41586-024-08252-9.

[14] Boming Xia, Xiaozhen Ye, and Adnan O.M Abuassba. Recent Research on AI in Games. In *2020 International Wireless Communications and Mobile Computing (IWCMC)*, pages 505–510, June 2020. doi: 10.1109/IWCMC48107.2020.9148327. URL https://ieeexplore.ieee.org/document/9148327.

[15] Markus Dablander. Future Research Avenues for Artificial Intelligence in Digital Gaming: An Exploratory Report, December 2024. URL https://arxiv.org/abs/2412.14085.

[16] Shane Legg and Marcus Hutter. Universal Intelligence: A Definition of Machine Intelligence. *Minds and Machines*, 17(4):391–444, December 2007. ISSN 1572-8641. doi: 10.1007/s11023-007-9079-x. URL https://link.springer.com/article/10.1007/s11023-007-9079-x.

[17] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA, October 2018. ISBN 978-0-262-03924-6.

[18] Minqi Jiang, Tim Rocktäschel, and Edward Grefenstette. General Intelligence Requires Rethinking Exploration. *Royal Society Open Science*, 10(6):230539, 2023. doi: 10.1098/rsos.230539. URL https://royalsocietypublishing.org/doi/10.1098/rsos.230539.

[19] Eduardo Pignatelli, Johan Ferret, Matthieu Geist, Thomas Mesnard, Hado van Hasselt, and Laura Toni. A survey of temporal credit assignment in deep reinforcement learning. *Transactions on Machine Learning Research*, 2024. ISSN 2835-8856. URL https://openreview.net/forum?id=bNtr6SLgZf. Survey Certification.

[20] Yuri Burda, Harrison Edwards, Amos Storkey, and Oleg Klimov. Exploration by Random Network Distillation. In *7th International Conference on Learning Representations (ICLR 2019)*, 2019. URL https://arxiv.org/abs/1810.12894.

[21] Joaquin Vanschoren. *Meta-Learning*, pages 35–61. Springer International Publishing, Cham, 2019. ISBN 978-3-030-05318-5. doi: 10.1007/978-3-030-05318-5_2. URL https://doi.org/10.1007/978-3-030-05318-5_2.

[22] Jacob Beck, Risto Vuorio, Evan Zheran Liu, Zheng Xiong, Luisa Zintgraf, Chelsea Finn, and Shimon Whiteson. A Tutorial on Meta-Reinforcement Learning. *Foundations and Trends in Machine Learning*, 18(2-3):224–384, 2025. doi: 10.1561/2200000080. URL https://arxiv.org/abs/2301.08028.

[23] Maxime Chevalier-Boisvert, Bolun Dai, Mark Towers, Rodrigo Perez-Vicente, Lucas Willems, Salem Lahlou, Suman Pal, Pablo Samuel Castro, and J Terry. Minigrid & Miniworld: Modular & Customizable Reinforcement Learning Environments for Goal-Oriented Tasks. In *Advances in Neural Information Processing Systems*, volume 36, pages 73383–73394. Curran Associates, Inc., 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/e8916198466e8ef218a2185a491b49fa-Paper-Datasets_and_Benchmarks.pdf.

[24] Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning Phrase Representations using RNN Encoder-Decoder for Statistical

Machine Translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, Doha, Qatar, October 2014. Association for Computational Linguistics. doi: 10.3115/v1/D14-1179. URL https://aclanthology.org/D14-1179/.

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

[26] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with Deep Reinforcement Learning. In *Proceedings of the NIPS 2013 Deep Learning Workshop*, 2013. URL https://arxiv.org/abs/1312.5602.

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

[28] Christopher J. C. H. Watkins and Peter Dayan. Technical note: Q-learning. *Machine Learning*, 8(3):279–292, May 1992. ISSN 1573-0565. doi: 10.1007/BF00992698. URL https://doi.org/10.1023/A:1022676722315.

[29] Andrew G. Barto, Richard S. Sutton, and Charles W. Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13 (5):834–846, September 1983. doi: 10.1109/TSMC.1983.6313077. URL https://ieeexplore.ieee.org/abstract/document/6313077.

[30] Hado van Hasselt, Yotam Doron, Florian Strub, Matteo Hessel, Nicolas Sonnerat, and Joseph Modayil. Deep Reinforcement Learning and the Deadly Triad, December 2018. URL https://arxiv.org/abs/1812.02648.

[31] J.N. Tsitsiklis and B. Van Roy. An analysis of temporal-difference learning with function approximation. *IEEE Transactions on Automatic Control*, 42(5):674–690, May 1997. ISSN 1558-2523. doi: 10.1109/9.580874. URL https://ieeexplore.ieee.org/document/580874.

[32] Laura Graesser and Wah Loon Keng. *Foundations of Deep Reinforcement Learning: Theory and Practice in Python*. Addison-Wesley Professional, Boston, 1st edition edition, December 2019. ISBN 978-0-13-517238-4.

[33] Aurelien Geron. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media, Inc., 2nd edition, September 2019. ISBN 978-1-4920-3264-9.

[34] Aske Plaat. *Deep Reinforcement Learning, a Textbook*. Springer Nature Singapore, 2022. doi: 10.1007/978-981-19-0638-1. URL https://link.springer.com/book/10.1007/978-981-19-0638-1.

[35] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3):229–256, May 1992. ISSN 1573-0565. doi: 10.1007/BF00992696. URL https://link.springer.com/article/10.1007/BF00992696.

[36] Matthias Lehmann. The Definitive Guide to Policy Gradients in Deep Reinforcement Learning: Theory, Algorithms and Implementations, March 2024. URL https://arxiv.org/abs/2401.13662.

[37] John Schulman, Philipp Moritz, Sergey Levine, Michael I. Jordan, and Pieter Abbeel. High-Dimensional Continuous Control Using Generalized Advantage Estimation. In *International Conference on Learning Representations (ICLR 2016)*, 2016. URL https://arxiv.org/abs/1506.02438.

[38] Pawel Ladosz, Lilian Weng, Minwoo Kim, and Hyondong Oh. Exploration in deep reinforcement learning: A survey. *Information Fusion*, 85: 1–22, 2022. ISSN 1566-2535. doi: https://doi.org/10.1016/j.inffus.2022.03.003. URL https://www.sciencedirect.com/science/article/pii/S1566253522000288.

[39] Richard M. Ryan and Edward L. Deci. Intrinsic and Extrinsic Motivations: Classic Definitions and New Directions. *Contemporary Educational Psychology*, 25(1):54–67, January 2000. ISSN 0361-476X. doi: 10.1006/ceps.1999.1020. URL https://www.sciencedirect.com/science/article/pii/S0361476X99910202.

[40] Jürgen Schmidhuber. A possibility for implementing curiosity and boredom in model-building neural controllers. In *Proceedings of the First International Conference on Simulation of Adaptive Behavior on From Animals to Animats*, pages 222–227. MIT Press, 1990. doi: 10.5555/116517.116542. URL https://dl.acm.org/doi/10.5555/116517.116542.

[41] Yuri Burda, Harri Edwards, Deepak Pathak, Amos Storkey, Trevor Darrell, and Alexei A. Efros. Large-Scale Study of Curiosity-Driven Learning. In *7th International Conference on Learning Representations (ICLR 2019)*, 2019. URL https://arxiv.org/abs/1808.04355.

[42] Jean-Bastien Grill, Florian Strub, Florent Altché, Corentin Tallec, Pierre Richemond, Elena Buchatskaya, Carl Doersch, Bernardo Avila Pires, Zhaohan Guo, Mohammad Gheshlaghi Azar, Bilal Piot, koray kavukcuoglu, Remi Munos, and Michal Valko. Bootstrap your own latent: A new approach to self-supervised Learning. In *Advances in Neural Information Processing Systems*, volume 33, pages 21271–21284. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper_files/paper/2020/file/f3ada80d5c4ee70142b17b8192b2958e-Paper.pdf.

[43] Timothy Hospedales, Antreas Antoniou, Paul Micaelli, and Amos Storkey. Meta-Learning in Neural Networks: A Survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(9):5149–5169, September 2022. ISSN 1939-3539. doi: 10.1109/TPAMI.2021.3079209. URL https://ieeexplore.ieee.org/document/9428530.

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

[45] Yuhuai Wu, Mengye Ren, Renjie Liao, and Roger Grosse. Understanding Short-Horizon Bias in Stochastic Meta-Optimization. In *International Conference on Learning Representations (ICLR 2018)*, 2018. URL https://openreview.net/forum?id=H1MczcgR-.

[46] Lilian Weng. Evolution Strategies, September 2019. URL https://lilianweng.github.io/posts/2019-09-05-evolution-strategies/.

[47] Zeyu Zheng, Junhyuk Oh, and Satinder Singh. On Learning Intrinsic Rewards for Policy Gradient Methods. In *Advances in Neural Information Processing Systems*, volume 31, 2018. URL https://proceedings.neurips.cc/paper_files/paper/2018/file/51de85ddd068f0bc787691d356176df9-Paper.pdf.

[48] Janarthanan Rajendran, Richard Lewis, Vivek Veeriah, Honglak Lee, and Satinder Singh. How Should an Agent Practice? *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(04):5454–5461, April 2020. doi: 10.1609/aaai.v34i04.5995. URL https://ojs.aaai.org/index.php/AAAI/article/view/5995.

[49] Zeyu Zheng, Junhyuk Oh, Matteo Hessel, Zhongwen Xu, Manuel Kroiss, Hado Van Hasselt, David Silver, and Satinder Singh. What Can Learned Intrinsic Rewards Capture? In *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 11436–11446. PMLR, 13–18 Jul 2020. URL https://proceedings.mlr.press/v119/zheng20b.html.

[50] Ferran Alet, Martin F. Schneider, Tomas Lozano-Perez, and Leslie Pack Kaelbling. Meta-learning curiosity algorithms. In *8th International Conference on Learning Representations (ICLR 2020)*, 2020. URL https://arxiv.org/abs/2003.05325.

[51] Batsirayi Mupamhi Ziki. Exploring Meta-learned Curiosity Algorithms. In *ICLR Blogposts 2024*, 2024. URL https://iclr-blogposts.github.io/2024/blog/exploring-meta-learned-curiosity-algorithms/.

[52] Jakub Grudzien Kuba, Christian Schroeder de Witt, and Jakob Foerster. Mirror Learning: A Unifying Framework of Policy Optimisation, November 2024.

[53] Chris Lu, Jakub Kuba, Alistair Letcher, Luke Metz, Christian Schroeder de Witt, and Jakob Foerster. Discovered Policy Optimisation. In *Advances in Neural Information Processing Systems*, volume 35, pages 16455–16468. Curran Associates, Inc., 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/688c7a82e31653e7c256c6c29fd3b438-Paper-Conference.pdf.

[54] 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, June 2021. URL http://github.com/google/brax.

[55] Kenny Young and Tian Tian. MinAtar: An Atari-Inspired Testbed for Thorough and Reproducible Reinforcement Learning Experiments, June 2019. URL https://arxiv.org/abs/1903.03176.

[56] Andrew Patterson, Samuel Neumann, Martha White, and Adam White. Empirical Design in Reinforcement Learning. *Journal of Machine Learning Research*, 25(318):1–63, 2024. URL http://jmlr.org/papers/v25/23-0183.html.

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

[58] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. In *3rd International Conference on Learning Representations (ICLR 2015)*, 2015. URL http://arxiv.org/abs/1412.6980.

[59] Rishabh Agarwal, Max Schwarzer, Pablo Samuel Castro, Aaron Courville, and Marc G. Bellemare. Deep Reinforcement Learning at the Edge of the Statistical Precipice. In *Advances in Neural Information Processing Systems 34 (NeurIPS 2021)*, 2021. URL https://arxiv.org/abs/2108.13264.

[60] Olive Jean Dunn. Multiple Comparisons Among Means. *Journal of the American Statistical Association*, 56(293):52–64, 1961. ISSN 0162-1459. doi: 10.2307/2282330. URL https://www.jstor.org/stable/2282330.

[61] Hado van Hasselt, Diana Borsa, and Matteo Hessel. DeepMind x UCL | Deep Learning Lecture Series 2021. URL https://www.youtube.com/playlist?list=PLqYmG7hTraZDVH599EItlEWsUOsJbAodm.

[62] Richard Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, 1957. ISBN 9780691146683.

[63] 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://direct.mit.edu/neco/article-abstract/9/8/1735.

[64] Christopher M. Bishop and Hugh Bishop. *Deep Learning: Foundations and Concepts*. Springer International Publishing, Cham, 2024. ISBN 978-3-031-45468-4. doi: 10.1007/978-3-031-45468-4_6. URL https://link.springer.com/book/10.1007/978-3-031-45468-4.

[65] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling, December 2014. URL https://arxiv.org/abs/1412.3555.

Appendix A

Proofs and Derivations

A.1 Proofs of Theorems

Thereom 3

Proof of Theorem 3.

$$
\begin{aligned}
V_\pi(s) &\leq Q_\pi(s, \pi'(s)) \\
&= \mathbb{E}\left[r(S_t, A_t) + \gamma V_\pi(S_{t+1}) \mid S_t = s, A_t = \pi'(s)\right] && (*) \\
&= \mathbb{E}_{\pi'}\left[r(S_t, \pi'(S_t)) + \gamma V_\pi(S_{t+1}) \mid S_t = s\right] \\
&\leq \mathbb{E}_{\pi'}\left[r(S_t, \pi'(S_t)) + \gamma Q_\pi(r(S_{t+1}, \pi'(S_{t+1}))) \mid S_t = s\right] && (**) \\
&= \mathbb{E}_{\pi'}\left[r(S_t, \pi'(S_t)) + \gamma r(S_{t+1}, \pi'(S_{t+1})) + \gamma^2 V_\pi(S_{t+2}) \mid S_t = s\right] \\
&\;\;\vdots \\
&= V_{\pi'}(s).
\end{aligned}
$$

□

Note that (*) from the above proof is from equation (2.1.6) and (**) is from equation (B.1.2). For this proof we repeatedly made use of (B.1.2) until we arrived at $V_{\pi'}(s)$.

The Policy Gradient Theorem

Recall that for policy-based methods our objective is to change the policy parameter in such a way that ensures the performance of the agent improves. However, the performance of the agent just doesn't depend on the actions selected but the state distribution. So the policy parameters affect both of these things. The actions of the agent directly affect rewards hence the performance. However it is not clear how the policy's parameters affect the state distribution [17]. The change in the state distribution is often unknown and complex. Therefore in order to calculate the gradient of the objective when the gradient depends on

the unknown effects of the state distribution we make use of the Policy Gradient Theorem. The Policy Gradient Theorem allows to find an expression of the gradient of objective with respect to the policy's parameters and this expression does not involve the derivative of the state distribution. We consider the Policy Gradient Theorem in the episodic case, i.e, the agent-environment sequences are divided in to episodes with a clear end and start. In this case we define the objective as,

$$J(\theta) := \mathbb{E}_\theta \left[G_0 | s_0 \right]. \tag{A.1.1}$$

In the above equation s_0 is the state that every episodes starts in. With all this we can now state the Policy Gradient Theorem taken from [61, Lecture 9].

Theorem 1 (The Policy Gradient Theorem (episodic))**.** *For any differentiable policy $\pi_\theta(s, a)$, let s_0 be the non-random starting state for each episode. Then the policy gradient of $J(\theta)$ is*

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta} \left[\sum_{t=0}^{T} \gamma^t Q_\pi(S_t, A_t) \nabla_\theta \log \pi_\theta(A_t | S_t) | s_0. \right] \tag{A.1.2}$$

where $Q_\pi(s, a) := \mathbb{E}_\pi \left[G_t | S_t = s, A_t = a \right]$ (see equation (2.1.4)) and T is the final time-step in the episode.

Before we go into the proof, let use first define some terms. The expectation $\mathbb{E}_{\pi_\theta}$ represents an expectation over all possible trajectories τ_t sampled from the policy π_θ. The probability of a trajectory $\tau_T = (S_0, A_0, S_1, A_1, \ldots, S_T, A_T)$ under the policy π_θ is defined as,

$$\Pr(\tau | \theta) = \prod_{t=0}^{T} \pi_\theta(A_t | S_t) P(S_{t+1} | S_t, A_t), \tag{A.1.3}$$

where $\pi_\theta(A_t | S_t)$ is the policy, and $P(S_{t+1} | S_t, A_t)$ is the transition probability function.

Proof. Recall our definition of the return from equation (2.1.1),

$$G_t := \sum_{k=t}^{T-1} \gamma^{k-t} r(S_k, A_k).$$

Notice that the return actually depends on the trajectory τ_T,

$$\tau_T = S_0, A_0, r(S_0, A_0), \cdots, S_T, A_T, r(S_T, A_T).$$

Therefore,

$$
\begin{aligned}
\nabla_\theta J(\theta) &= \nabla_\theta \mathbb{E}_{\pi_\theta}\left[G_0|s_0\right] \\
&= \nabla_\theta \int_{\tau_t} \Pr(\tau_t|\theta)\cdot G_0\, d\tau_t \quad \text{(using the definition of the expectation)} \\
&= \int_{\tau_t} \nabla_\theta \Pr(\tau_t|\theta)\cdot G_0\, d\tau_t \quad \text{(using the product rule and the fact that } G_0 \text{ does not depend on } \theta) \\
&= \int_{\tau_t} \nabla_\theta \Pr(\tau_t|\theta)\cdot G_0 \cdot \frac{\Pr(\tau_t|\theta)}{\Pr(\tau_t|\theta)}\, d\tau_t \\
&= \int_{\tau_t} \frac{\nabla_\theta \Pr(\tau_t|\theta)}{\Pr(\tau_t|\theta)}\cdot G_0 \cdot \Pr(\tau_t|\theta)\, d\tau_t \\
&= \int_{\tau_t} \nabla_\theta \log \Pr(\tau_t|\theta)\cdot G_0 \cdot \Pr(\tau_t|\theta)\, d\tau_t \quad \text{(using the definition of the logarithmic derivative)} \\
&= \mathbb{E}_{\pi_\theta}\left[G_0 \cdot \nabla_\theta \log \Pr(\tau_t|\theta)|s_0\right] \\
&= \mathbb{E}_{\pi_\theta}\left[G_0 \cdot \nabla_\theta \log \left(\prod_{t=0}^{T-1} \pi_\theta(A_t|S_t) P(S_{t+1}|S_t,A_t)\right)|s_0\right] \quad \text{(using equation (A.1.3))} \\
&= \mathbb{E}_{\pi_\theta}\left[G_0 \nabla_\theta \sum_{t=0}^{T-1} \left(\log \pi_\theta(A_t|S_t) + \log P(S_{t+1}|S_t,A_t)\right)|s_0\right] \quad \text{(using the product rule of logarithms)}
\end{aligned}
\tag{A.1.4}
$$

Now note that the environment dynamics $P(S_{t+1}|S_t, A_t)$ do not depend on θ as we are already conditioning on action A_t. We can then see that the explicit dynamics of the MDP disappear! Furthermore, we assume that the state distribution under the policy, $d_\pi(s)$, is stationary. This assumption allows us to treat the state distribution as fixed for a given policy. Now let us continue,

$$
\begin{aligned}
\nabla_\theta J(\theta) &= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \sum_{t=0}^{T-1} G_0 \left(\log \pi_\theta(A_t|S_t)\right)|s_0\right] \\
&= \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T-1} \gamma^t G_t \nabla_\theta \left(\log \pi_\theta(A_t|S_t)\right)|s_0\right].
\end{aligned}
$$

Now G_t is a random variable and Q_π represents its expected value under policy θ_π. The dynamics of the environment are captured implicitly in the $Q^\pi(s,a)$ function [61, Lecture 9]. We therefore can replace G_t with Q_π to end with,

$$
\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T-1} \gamma^t Q_\pi(S_t,A_t) \nabla_\theta \left(\log \pi_\theta(A_t|S_t)\right)|s_0\right].
$$

□

Appendix B

Algorithms

B.1 Dynamic Programming

Dynamic Programming (DP) allows us to find optimal policies for a given MDP. DP enables us to solve problems that have a temporal or sequential component by breaking them down into smaller subproblems and solving these subproblems individually. The optimal solutions from these subproblems are then combined to solve the overall problem.

More formally, for DP to be applicable, our problem needs to have optimal substructure. By optimal substructure, we mean that the Principle of Optimality applies to our problem. Bellman [62, p. 83] define the Principle of Optimality as:

Theorem 2 (Principle of Optimality)**.** *An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.*

In other words, the Principle of Optimality states that a policy π has the optimal value function from state s if and only if, for any state s' reachable from s, the policy π also achieves the optimal value from state s' [10].

In practice, it is beneficial if our problem also has overlapping subproblems. When we say a problem has overlapping subproblems, we mean that the problem's subproblems appear multiple times during the solution process. This characteristic allows us to avoid redundant calculations by caching the solutions to the subproblems and reusing these solutions whenever the same subproblem arises again. For example, consider the shortest path problem, adapted from Silver [10, Lecture 3]. If we want to find the shortest path from point A to point C, we might first compute the shortest path from A to an intermediate point B, and then from B to C. The overall shortest path from A to C would be the sum of the shortest paths from A to B and B to C. If we later need to find the

shortest path from C back to A, we can reuse the calculations for the path from B to A and B to C, assuming the paths are reversible. This reuse of calculations illustrates the concept of overlapping subproblems in Dynamic Programming.

Markov Decision Processes (MDPs) satisfy the optimal substructure required for applying Dynamic Programming (DP) techniques. Recall the recursive definitions of the value functions from Equations (2.1.7) and (2.1.8). These equations demonstrate how we can decompose the value function into recursively defined components: first, by performing the action that maximises the expected return at step t in state S_t, and then by continuing with optimal actions in subsequent steps.

Value functions can also serve as a cache, storing the values for all possible states in our state space. Using DP, we can evaluate policies (planning) by calculating the value function for a given policy, determining how good that policy is. Furthermore, policy iteration (control) enables us to find both the optimal policy and the optimal value function for a specific MDP. This approach requires full knowledge of the MDP, including the transition dynamics and the reward function.

Next, we will take a look at DP algorithms: Policy Evaluation and Policy Improvement, and end with Generalised Policy Iteration (GPI).

Policy Evaluation

Policy Evaluation is a prediction problem. We aim to predict how good a policy is by calculating its value functions. We randomly initialise a value function and apply iterative updates until we reach V_π, the true value function of policy π. We use synchronous updates, i.e., we update the value function for all states in $\mathcal{S}$. It is important to note that we assume that the dynamics of the environments are known , e.g. we know the transition probability function of the MDP.

When the MDP dynamics are known, equation (2.1.5) can be interpreted as a linear system comprising $|\mathcal{S}|$ equations with an equal number of unknowns, where each unknown represents $V_\pi(s)$ for a state s in the state space. Here $|\mathcal{S}|$ is the number of elements in the state space. We then update our approximations of the value function using equation (2.1.5) as our update rule. Equation (B.1.1) illustrates this update rule,

$$V_k := \sum_{a \in \mathcal{A}} \pi(a|s) \sum_{s' \in \mathcal{S}} P(s'|s,a) \left[r(s,a) + \gamma V_{k-1}(s'). \right] \tag{B.1.1}$$

Now if we let k denote the iteration number then V_k does converge to V_π as $k \to \infty$. The existence and uniqueness of V_π is guaranteed for as long as $\gamma < 1$ or there's eventual termination of the MDP [17, p. 74]. In practice we have a limited number of iterations to work with. Once the difference between V_k and V_{k+1} is less than some Θ value where $0 < \Theta \ll 1$, we stop the Policy Evaluation algorithm. In other words the algorithm stops when the difference between our

old estimate and the new estimate of the value function is sufficiently small. Algorithm 7 is the Policy Evaluation algorithm taken from [17, p. 75].

Algorithm 7 Policy Evaluation from [17, p. 75]

Require: Policy π, threshold $\Theta > 0$, and a MDP
Ensure: State-value function V for policy π and $V(s) = 0$ for terminal states.

```
Initialise V(s) = 0 for all states s
repeat
    Δ ← 0
    for each state s do
        v ← V(s)
        V(s) ← Σ_{a∈A} π(a|s) Σ_{s'} p(s'|s,a)[r + γV(s')]
        Δ ← max(Δ, |v − V(s)|)
    end for
until Δ < Θ
return V
```

Policy Improvement

Policy Improvement is concerned with improving deterministic policies. Using Policy Evaluation we can determine the expected return of policy π. To do this we start with equation (2.1.6),

$$Q_\pi(s,a) = \sum_{s' \in \mathcal{S}} P(s'|s,a)\left[r(s,a) + \gamma Q_\pi(s',a')\right].$$

Recall that the above equation describes the value of selecting action a in state s and thereafter following policy π. If we have $Q_\pi(s,a) > V_\pi(s)$, this implies that: 1) It is better to pick action a every time we encounter state s. 2) This new approach of choosing a in state s results in a better policy than the original policy π. With this idea we can now introduce the Policy Improvement Theorem [17, 10].

Theorem 3 (Policy Improvement Theorem)**.** *Let π and π' be two deterministic policies such that for all $s \in \mathcal{S}$,*

$$Q_\pi(s, \pi'(s)) \geq V_\pi(s). \tag{B.1.2}$$

Then policy π' must be as good or better than policy π. The expected return of policy π' from all states should therefore be equal or greater than the expected return of policy π. In other words,

$$V_{\pi'}(s) \geq V_\pi(s).$$

The proof is provided in Appendix A.1.

The Policy Improvement Theorem shows how we can improve the policy at a particular state by selecting a specific action. This concept can be extended to

all states by considering all possible actions. We achieve this by selecting the action with the highest value with respect to Q_π in every state. This behaviour of selecting the action with the highest value is known as acting greedily. In other words, we obtain a better policy π' given by

$$\pi'(s) = \underset{a}{\operatorname{argmax}}\, Q_\pi(s, a). \tag{B.1.3}$$

This new policy π' selects the action that maximises the action-value function of the original policy π. This process of obtaining a better policy by acting greedily with respect to the action-value function of the original policy is known as policy improvement [17]. Policy improvement stops when

$$Q_\pi(s, \pi'(s)) = \max_a Q_\pi(s, a) = Q_\pi(s, \pi(s)) = V_\pi(s).$$

Algorithm 8 is the Policy Improvement algorithm taken from [17],

Algorithm 8 Policy Improvement, taken from [17]

Require: Policy π and a MDP
Ensure: A new policy that is at least as good as the old policy π.

```
1: policy-stable ← True
2: Initialise policy π for all states.
3: for each state s do
4:     old-action ← π(s)
5:     π(s) ← argmax_a Q_π(s, a)
6:     if old-action ≠ π(s) then
7:         policy-stable ← False
8:     end if
9: end for
10: if policy-stable then
11:     return π
12: else
13:     go to to line 3
14: end if
```

Generalised Policy Iteration

Generalised Policy Iteration (GPI) refers to the interaction between the processes of Policy Evaluation and Policy Improvement. It is a framework for policy optimisation, and RL algorithms can be described by this framework. Policy iteration is an algorithm that fits within this framework. Policy Iteration works by first performing Policy Evaluation and then Policy Improvement. We go around these two processes iteratively until we reach the optimal policy. Figure B.1, taken from [10, Lecture 3], illustrates Policy Iteration.

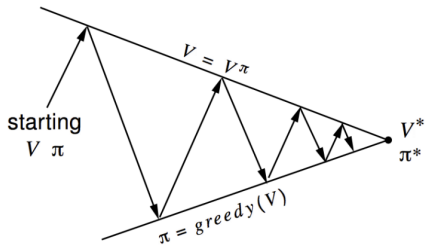


Figure B.1: An illustration of how Policy Iteration works. Taken from [10, Lecture 3].

From Figure B.1, we can observe that the upward facing arrows represent policy evaluation, while the downward facing arrows represent policy improvement. Each new policy obtained is a strict improvement over the previous policy [17], unless the old policy is already optimal. Once these two processes no longer produce any changes, we say that the algorithm has stabilised. This indicates that we have indeed obtained the optimal policy and the optimal value function. Recall that we are focusing on finite MDPs, and therefore, Policy Iteration converges to the optimal policy within a finite number of iterations [17]. Algorithm 9 presents the algorithm for Policy Iteration, as taken from [17, p.83].

Algorithm 9 Policy Iteration [17, p.83]

Require: Policy π, threshold $\Theta > 0$, and a Markov Decision Process (MDP)
Ensure: The optimal state-value function V^* for the optimal policy π^*

1: **Initialise:** $V(s)$ and $\pi(s)$ for all states s, ensuring $V(s) = 0$ for terminal states
2: **repeat**
3: Perform Policy Evaluation (Algorithm 7)
4: Perform Policy Improvement (Algorithm 8)
5: **until** the policy is stable
6: **return** π and V

B.2 Advantage Actor Critic

The following algorithm presents the implementation details of the Advantage Actor-Critic (A2C) method.

Algorithm 10 Advantage Actor-Critic (A2C) Algorithm taken from [32]

Require: Entropy regularisation weight $\beta \geq 0$, actor learning rate $\alpha \geq 0$, critic learning rate $\alpha_C \geq 0$, maximum number of episodes $MAX\_EPISODE$, discount factor $\gamma \in (0, 1]$, and policy and value networks with parameters θ (actor) and ϕ (critic) respectively.

Ensure: Optimised policy π^* and value function V^*.

1: Randomly initialise actor and critic parameters θ and ϕ.
2: **for** episode = 0 to $MAX\_EPISODE$ **do**
3: Collect a trajectory $\tau = \{s_t, a_t, r_t, s_{t+1}\}_{t=0}^{T}$ by interacting with the environment using policy π_θ.
4: **for** $t = 0$ to T **do**
5: Compute the predicted value $\hat{V}(s_t)$ using the critic network:

$$\hat{V}(s_t; \phi)$$

6: Compute the GAE.
7: Compute policy entropy H_t:

$$H_t := -\sum_a \pi_\theta(a|s_t) \log \pi_\theta(a|s_t)$$

8: **end for**
9: Compute value loss using mean squared error (MSE):

$$\mathcal{L}_{\text{value}}(\phi) = \frac{1}{T} \sum_{t=0}^{T} \left(r_t + \gamma \hat{V}(s_{t+1}; \phi) - \hat{V}(s_t; \phi) \right)^2 \tag{B.2.1}$$

10: Compute policy loss:

$$\mathcal{L}_{\text{policy}}(\theta) = -\frac{1}{T} \sum_{t=0}^{T} \left(\hat{A}(s_t, a_t) \log \pi_\theta(a_t|s_t) - \beta H_t \right) \tag{B.2.2}$$

11: Update critic parameters using gradient descent:

$$\phi \leftarrow \phi - \alpha_C \nabla_\phi \mathcal{L}_{\text{value}}(\phi)$$

12: Update actor parameters using gradient descent:

$$\theta \leftarrow \theta - \alpha \nabla_\theta \mathcal{L}_{\text{policy}}(\theta)$$

13: **end for**
14: **return** θ, ϕ

Appendix C

Deep Learning: LSTM and GRU Architectures

In this section, we assume that the reader is familiar with deep learning (DL) and, as such, we will not cover it in great detail. The main focus of this section is recurrent neural network (RNN) architectures. Specifically, we examine the long short-term memory (LSTM) architecture [63] and the gated recurrent unit (GRU) architecture [24]. In particular, we first examine the update mechanisms for LSTM's cell and hidden states, before exploring the simplified update process in GRU architecture. This is because GRUs were designed as a simplification of LSTMs; therefore, understanding LSTMs first provides the necessary background to understand the GRU's design choices. GRUs are emphasized because all the RL algorithms implemented in this project make use of the GRU architecture.

RNNs are mainly used to process sequential data. However, classic RNNs are known to have exploding and vanishing gradient issues. As a result, the classic RNN struggles to handle data with long-term dependencies due to vanishing gradients [64].

The LSTM architecture [63] was designed to address the vanishing gradient problem that the classic RNN encounters. Its architecture has better gradient flow properties due to its gating mechanism. An LSTM has three gates: a forget gate, an input gate, and an output gate.

The classic RNN's module is quite simple compared to an LSTM, as it involves multiplying the hidden state and current input by weight matrices, summing them, and passing the result through a tanh activation function to output the next hidden state. In LSTMs, we do not only have a hidden state but also a cell state. The cell state serves as the long-term memory of the network, whereas the hidden state acts as the short-term memory and is used for output generation at each time-step. The gates of the LSTM allow it to remove and add information

to the cell state. The gates are single-layer neural networks with an activation function and a pointwise multiplication operation.

We start with the forget gate, which is a neural network layer with a sigmoid activation function. The sigmoid function, σ, outputs real numbers between 0 and 1. The forget gate takes in the current input, x_t, and the previous hidden state, h_{t-1}, and outputs:

$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] + b_f\right). \tag{C.0.1}$$

In the above equation, $[\cdot, \cdot]$ denotes concatenation of vectors, W_f and b_f are the weights and bias of the forget gate, respectively. The forget gate determines how much of the previous cell state, C_{t-1}, should be retained. If f_t has values close to one, this indicates that the gate considers the current information very important and retains most of it. Likewise, if f_t outputs values close to zero, it indicates that the current values are not important and should be discarded.

Next, we determine what information will be stored in the cell state. To achieve this, we first use the input gate, another sigmoid-activated neural network layer, which decides which values should be updated. Secondly, we use a tanh-activated neural network layer called the candidate layer. The candidate gate normalises the previous hidden state and the current input between -1 and 1 and is responsible for generating new information that can be passed into the updated cell state. The output of the input gate is similar to that of the forget gate:

$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] + b_i\right). \tag{C.0.2}$$

The candidate layer's output is:

$$\tilde{C}_t = \tanh\left(W_c \cdot [h_{t-1}, x_t] + b_c\right). \tag{C.0.3}$$

With this, we can finally update the cell state:

$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t. \tag{C.0.4}$$

The operation $\odot$ represents element-wise multiplication. With the cell state updated, the next step is to determine what the LSTM will output. The output gate decides how much of the current cell state should be used to calculate the new hidden state. The output gate's output is:

$$o_t = \sigma\left(W_o \cdot [h_{t-1}, x_t] + b_o\right). \tag{C.0.5}$$

If o_t is close to 1, more information is used; if o_t is close to 0, less information is used. We can then finally calculate the new hidden state:

$$h_t = o_t \odot \tanh\left(C_t\right). \tag{C.0.6}$$

While LSTMs effectively handle long-term dependencies, their architecture's complexity leads to longer training times compared to classic RNNs [64]. The

Gated Recurrent Unit (GRU) [24] was introduced as an alternative that maintains the benefits of gating mechanisms while simplifying the architecture. Empirical studies have shown that GRUs can achieve comparable performance to LSTMs despite their simpler structure [65].

In a GRU cell, the following simplifications are made: (1) the cell state is removed, and the hidden and cell states are merged into a single state still known as the hidden state; (2) the forget and input gates are combined into a single gate: the update gate. The update gate determines how much the hidden state updates its content, i.e., how much past information needs to be passed along to the future. The output from the update gate is given by:

$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t] + b_z). \tag{C.0.7}$$

The reset gate determines how much of the past hidden state should be forgotten before calculating the new candidate values, $\tilde{h}_t$. Its output is:

$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t] + b_r). \tag{C.0.8}$$

We then use r_t to obtain $\tilde{h}_t$ using the following formula:

$$\tilde{h}_t = \tanh(W_{\tilde{h}} \cdot [r_t \odot h_{t-1}, x_t]). \tag{C.0.9}$$

Finally, we compute the new hidden state:

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t. \tag{C.0.10}$$

If $z_t \approx 1$, most of the past information from the hidden state is ignored. Conversely, if $z_t \approx 0$, most of the new information is ignored when calculating the new hidden state.

Appendix D

Implementation and Hyperparameter Details

Table D.1: PPO Agent Architecture Details

Component	Specification
Input Processing	
Observation Encoder	Feed-forward(256) + ReLU
Action Embedding	6 actions, 16 dimensions
Core Architecture	
GRU	64 hidden units, 1 layer
Output Heads	
Actor Head	Feed-forward(64) + Tanh → Feed-forward(6)
Critic Head	Feed-forward(64) + Tanh → Feed-forward(1)

Table D.2: BYOL-Explore Architecture Details

Component	Specification
Online Network	
Encoder	Feed-forward(256) + ReLU → Feed-forward(256)
Action Embedding	6 actions, 16 dimensions
Closed-loop GRU	256 hidden units, 1 layer
Open-loop GRU	256 hidden units, 1 layer
Predictor	Feed-forward(256) + ReLU → Feed-forward(256)
Target Network	
Target Encoder	Feed-forward(256) + ReLU → Feed-forward(256)

Table D.3: Random Network Distillation Architecture Details

Component	**Specification**
Target Network	
Architecture	Feed-forward(256) + ReLU → Feed-forward(256)
Predictor Network	
Architecture	Feed-forward(256) + ReLU → Feed-forward(256) + ReLU → Feed-forward(256)

Table D.4: Reward Combiner Architecture Details

Component	**Specification**
Input Processing	
Action Embedding	6 actions, 16 dimensions
Reward Embedding	Feed-forward(16) + ReLU
Core Architecture	
GRU	128 hidden units
Output Processing	
Feed-forward	Feed-forward(128) + ReLU → Feed-forward(1)
Output Activation	Sigmoid (ensures $\lambda \in [0, 1]$)

Table D.5: PPO Agent with GRU Architecture Hyperparameters

Hyperparameter	**Value**
Agent Architecture	
Action Embedding Dimension	16
RNN Hidden Dimension	1024
RNN Number of Layers	1
Head Hidden Dimension	256
Training Parameters	
Number of Environments	8192
Number of Steps	16
Update Epochs	1
Number of Minibatches	16
Total Timesteps	5,000,000
Learning Rate	0.001
Clipping Epsilon	0.2
Discount Factor (γ)	0.99
GAE Lambda (λ)	0.95
Entropy Coefficient	0.01
Value Function Coefficient	0.5
Maximum Gradient Norm	0.5
Seed	42

Table D.6: BYOL-Explore Agent Hyperparameters

Hyperparameter	**Value**
Agent Architecture	
Action Embedding Dimension	16
RNN Hidden Dimension	1024
RNN Number of Layers	1
Head Hidden Dimension	256
Training Parameters	
Number of Environments	8192
Number of Steps	16
Update Epochs	1
Number of Minibatches	16
Total Timesteps	5,000,000
Learning Rate	0.001
Clipping Epsilon	0.2
Discount Factor (γ)	0.99
GAE Lambda (λ)	0.95
Entropy Coefficient	0.01
Value Function Coefficient	0.5
Maximum Gradient Norm	0.5
Seed	42
Intrinsic Reward Parameters	
Predictor Learning Rate	0.001
Reward Normalization Parameter	0.99
EMA Parameter	0.99
Anneal Predictor Learning Rate	False

Table D.7: Random Network Distillation (RND) Agent Hyperparameters

Hyperparameter	**Value**
Agent Architecture	
Action Embedding Dimension	16
RNN Hidden Dimension	1024
RNN Number of Layers	1
Head Hidden Dimension	256
Training Parameters	
Number of Environments	8192
Number of Steps	16
Update Epochs	1
Number of Minibatches	16
Total Timesteps	5,000,000
Learning Rate	0.001
Clipping Epsilon	0.2
Discount Factor (γ)	0.99
GAE Lambda (λ)	0.95
Entropy Coefficient	0.01
Value Function Coefficient	0.5
Maximum Gradient Norm	0.5
Seed	42
Intrinsic Reward Parameters	
Predictor Learning Rate	0.001
Intrinsic Reward Discount Factor (γ_{int})	0.99

Table D.8: Evolution Strategy (ES) Hyperparameters for the RC-Action

Hyperparameter	**Value**
Training Parameters	
Population Size	128
Number of Generations	80
Evolution Strategy Seed	69696969
OpenES Strategy Parameters	
Optimizer	Adam
Learning Rate Decay	1.0
Sigma Decay	0.999
Initial Sigma	0.04
Objective	Maximize

Appendix E

Additional Plots

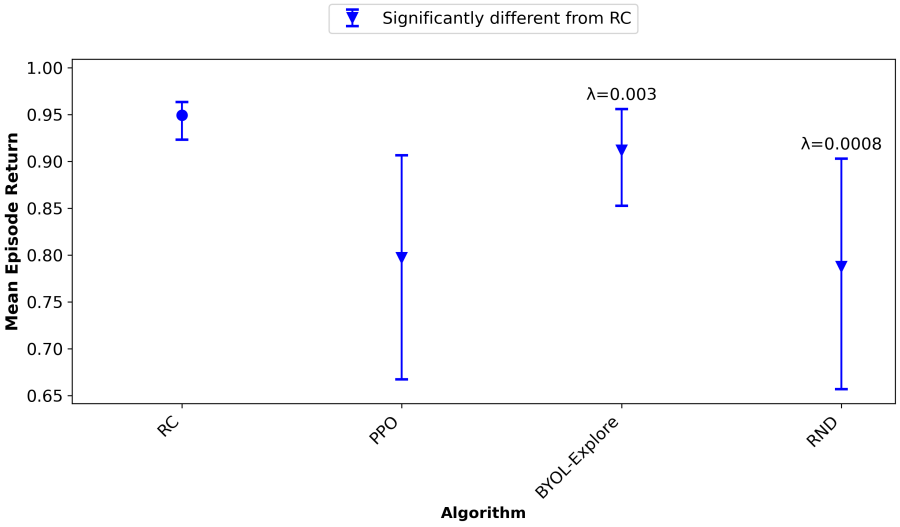


Figure E.1: Average final episode return at the end of training in the `DoorKey-5x5` grid-world, across 30 agents with 95% confidence intervals. Optimal λ values shown above BYOL-Explore and RND.

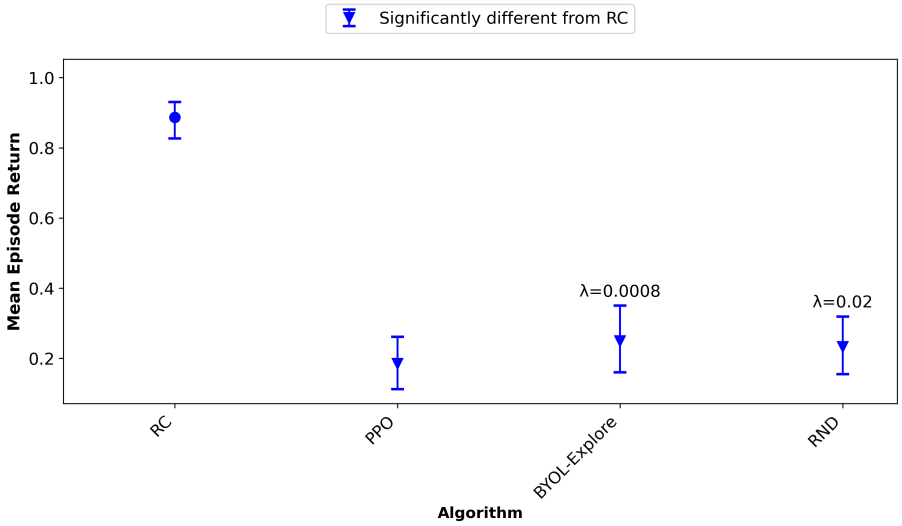


Figure E.2: Average final episode return at the end of training in the `DoorKey-6x6` grid-world, across 30 agents with 95% confidence intervals. Optimal λ values shown above BYOL-Explore and RND.

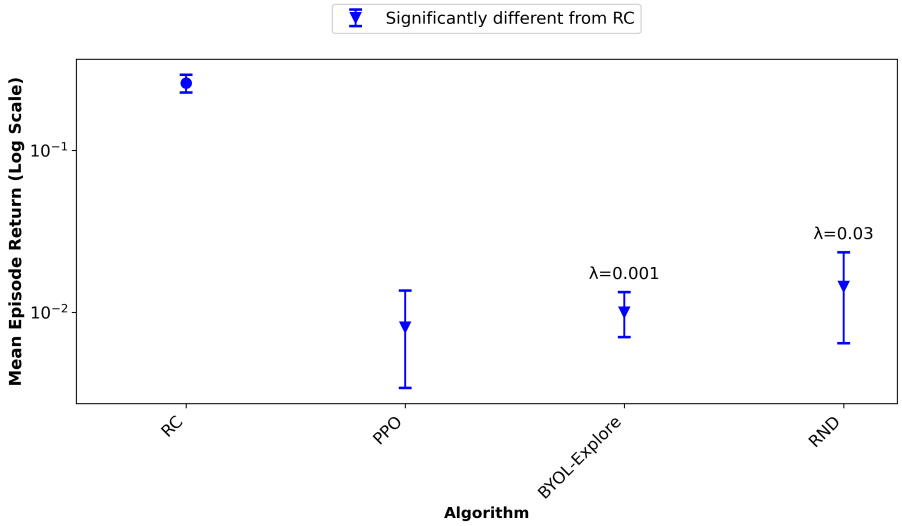


Figure E.3: Average final episode return at the end of training in the `DoorKey-8x8` grid-world, across 30 agents with 95% confidence intervals. Optimal λ values shown above BYOL-Explore and RND.

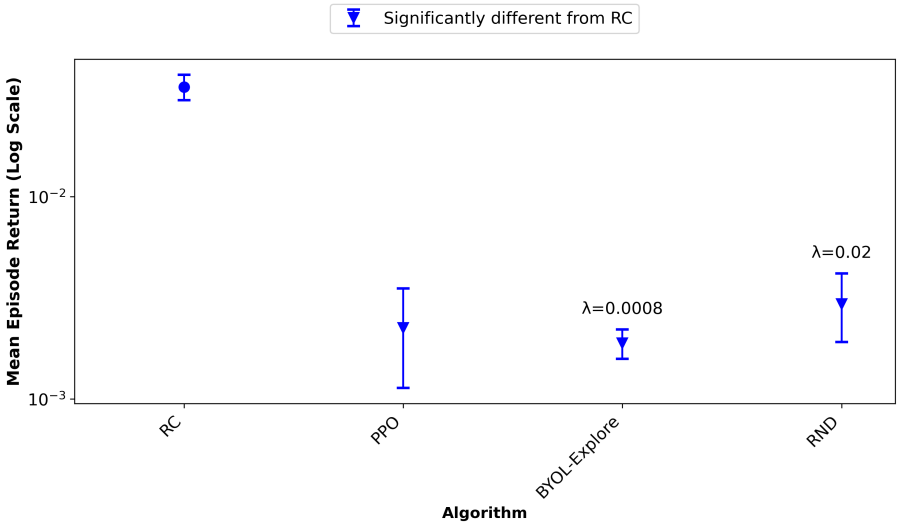


Figure E.4: Average final episode return at the end of training in the `DoorKey-16x16` grid-world, across 30 agents with 95% confidence intervals. Optimal λ values shown above BYOL-Explore and RND.