International Journal of Research in Engineering and Science (IJRES)
ISSN (Online): 2320-9364, ISSN (Print): 2320-9356
www.ijres.org Volume 10 Issue 8 ||August 2022 ||PP. 483-598

Advancing Deep Reinforcement Learning Techniques: Contributions in Reward Design, Temporal Credit Assignment, State Representation Learning, and Model Learning and Planning

Ceil Hong. Zhang
Massachusetts Institute of Technology
Massachusetts, USA
zhanghongceil@pku.org.cn

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Reinforcement learning (RL) is a machine learning paradigm concerned with how an agent learns to predict and control its own experience stream so as to maximize long-term cumulative reward. In the past decade, deep reinforcement learning (DeepRL), a subfield that aims to combine the sequential decision-making techniques in RL with the powerful non-linear function approximation tools offered by deep learning, has seen great success such as defeating human champions in the ancient board game Go and achieving expert-level performance in complex strategy games like Dota 2 and Starcraft. It has also had an impact on real-world applications. Examples include robot control, stratospheric balloon navigation, and controlling nuclear fusion plasma.

This thesis aims to further advance DeepRL techniques. Concretely, this thesis makes contributions in the following four directions: 1) In reward design, we develop a novel meta-learning algorithm for learning reward functions that facilitate policy optimization. Our algorithm improves the performance of policy-gradient methods and outperforms handcrafted heuristic reward functions. In a follow-up study, we show that the learned reward functions can capture knowledge about long-term exploration and exploitation and can generalize to different RL algorithms and changes in the environment dynamics. 2) In temporal credit assignment, we explore methods based on pairwise weights that are functions of the state in which the action was taken, the state in which the reward was received, and the time elapsed in between. We develop a metagradient algorithm for adapting these weights during policy learning. Our experiments show that our method achieves better performance than competing approaches. 3) In state representation learning, we investigate using random deep action-conditional prediction tasks as auxiliary tasks to help agents learn better state representations. Our experiments show that random deep action-conditional predictions can often yield better performance than handcrafted auxiliary tasks. 4) In model learning and planning, we develop a new method for learning value-equivalent models, a class of models that demonstrates strong empirical performance lately, that generalizes existing methods. Our experiments show that our method can improve both the model prediction accuracy and the control performance of the downstream planning procedure.

CHAPTER 1

Introduction

Reinforcement learning (RL) [Sutton and Barto, 2018] is a machine learning paradigm concerned with how an agent learns to predict and control its own experience stream so as to maximize the accumulation of a scalar signal called reward. At each time step, the agent takes an action based on its history of observations and actions in the environment. Through the action, the agent impacts the environment and then receives the next observation and a reward signal. The mapping from a history to an action (or a probability distribution over the action space) is called a policy. The goal of an RL agent is to find a good policy so as to maximize the cumulative reward in the long term.

In the past decade, deep learning (DL) [LeCun et al., 2015], a field of machine learning concerned with learning hierarchical representations of data from raw sensory inputs, has seen great success and revolutionized various areas including computer vision [Krizhevsky et al., 2012, He et al., 2016], speech recognition [Hinton et al., 2012], and natural language processing [Bahdanau et al., 2015, Devlin et al., 2018]. Benefiting from recent advances in DL, RL also saw significant progress and made several groundbreaking achievements. For example, deep Q-network (DQN) [Mnih et al., 2015] is the first RL agent that learned to play Atari 2600 video games [Bellemare et al., 2013] from raw pixels. AlphaGo [Silver et al., 2016] is the first computer program that defeated a human champion in the ancient board game Go. OpenAI Five [Berner et al., 2019] and AlphaStar [Vinyals et al., 2019] demonstrate human-level performance in challenging video games like Dota 2 and StarCraft. In addition to simulated domains like games, RL also had an impact on real-world applications such as robot control [Levine et al., 2016, Andrychowicz et al., 2020], stratospheric balloon navigation [Bellemare et al., 2013], and controlling nuclear fusion plasma [Degrave et al., 2022].

At the core of these recent advances is the combination of RL techniques and DL techniques. By employing DL models (mostly deep neural networks) as powerful non-linear function approximators, RL agents can learn policies and/or value functions directly from raw sensory inputs. This enables RL agents to scale up to high-dimensional domains that were thought intractable before such as Go. Due to the success and the growing interest in combining RL and DL, the intersection of these two fields quickly evolved into a vital subfield called deep reinforcement learning (DeepRL).

This thesis aims to further advance DeepRL techniques. To this end, we explore the following four directions. The first direction is reward design. When formulating a task into an RL problem, the agent designer needs to design a reward function that defines the desired behavior for the agent. It is known that certain transformations of the reward function will not change the ordering over policies [Ng et al., 1999]. However, they can change the learning efficiency for better or for worse. Thus an important question in reward design is to find a reward function that can help the agent learn the desired behavior efficiently. Finding such a reward function often requires a lot of human effort and domain knowledge. Thus it would be appealing if the agent can *learn* an *intrinsic reward function* for training itself. Chapter 3 and Chapter 4 address how to learn intrinsic reward functions. The second direction is temporal credit assignment. How much credit/blame should an action take for a later reward? This is the fundamental temporal credit assignment problem in RL. Most existing methods assign credits based on recency. For each reward, actions closer in time receive more credits and actions farther in time receive less credit. However, this simple heuristic ignores the inherent structure of the environment and thus is ineffective at assigning credit. Chapter 5 studies how to enable an agent to discover useful structures of the environment so that it can assign credit more effectively. The third direction is state representation learning. One of the most prominent advantages of DeepRL is its ability to learn non-linear state representations from raw inputs in an end-to-end manner. Usually rewards are the only learning signals. However, rewards are often sparse and delayed and thus learning can be inefficient. One way of addressing this learning inefficiency is to design unsupervised auxiliary tasks to help the agent learn better state representations. In practice, simple auxiliary tasks often yield good state representations and strong empirical performance. To explore the extreme, Chapter 6 presents an empirical study of using random deep action-conditional predictions as auxiliary tasks for state representation learning. The fourth direction is model learning. Model-based RL methods learn a simulator of the environment and plan with the learned simulator. This simulator is called a model. Learning an accurate model of the environment is the key to the success of model-based RL methods and thus it is critical to learn a model efficiently. Chapter 7 addresses this problem and presents a new algorithm for learning a specific class of models called value-equivalent models.

Chapter 3, Chapter 4, and Chapter 5 can also be viewed as addressing the topic of *discovery* in RL. Instead of specifying "what" an agent needs to learn and "how" it learns, the discovery problem asks how to enable an agent to discover the "what" and the "how". Chapter 3 and Chapter 4 address the "what" part and focus on discovering intrinsic reward functions. Chapter 5 addresses the "how" part and focuses on discovering temporal credit assignment mechanisms.

1.1 Roadmap and Preview of Contributions

In this section, we first present a roadmap for this thesis. Then we give a preview for the contribution of each chapter. We finish by providing references to publications based on the work done in this thesis.

Chapter 2 provides some background knowledge in RL and DeepRL. Chapter 3, 4, 5, 6, and 7 are the main contributions. Chapter 8 concludes this thesis and discusses future research directions. Chapter 3 and Chapter 4 address the reward design problem. In Chapter 3, we present a meta-learning algorithm that learns an intrinsic reward function for policy-gradient-based RL agents in a data-driven way. Then in Chapter 4, we study the property of the learned intrinsic reward functions. Chapter 5 addresses temporal credit assignment. We explore a more general temporal credit assignment mechanism and propose a practical meta-learning implementation of it. Chapter 6 addresses state representation learning. We investigate how random deep action-conditional predictions can form good auxiliary tasks that provide additional signals for learning state representations. Chapter 7 addresses model learning. We focus on a specific class of models called value-equivalent models and propose a new method for updating such models which generalizes from existing methods. A more detailed preview for the contribution of each chapter is as follows.

Learning Intrinsic Rewards for Policy Gradient Methods (Chapter 3)

This chapter addresses the reward design problem. In RL, the reward function defines the task and thus is usually considered fixed and immutable. However, in practice, agent designers often find it convenient to modify the reward function in a way that facilitates learning. Thus it is useful to distinguish two different kinds of rewards: the *extrinsic rewards* which capture the preference of the agent designer over the agent's behaviors and the *intrinsic rewards* which serve as learning signals to improve the learning dynamics of the agent. This distinction is formalized in the optimal reward framework [Singh et al., 2010]. Building on the optimal reward framework, the main contribution of this chapter is a novel meta-learning algorithm for learning intrinsic rewards for policy-gradient based *learning* agents. Through the empirical study in the Atari domain and the Mujoco continuous control domain, we show that augmenting the policy learner with additive intrinsic rewards learned by the proposed method yields better performance than the base policy learner. We also show that using the learned intrinsic rewards performs better than using handcrafted heuristic intrinsic rewards.

What Can Learned Intrinsic Rewards Capture? (Chapter 4)

In the previous chapter, we developed a gradient-based meta-learning algorithm for learning an intrinsic reward function for a policy gradient learning agent. In this chapter, we focus on understanding what can be captured by the learned reward functions. To investigate this, we propose a scalable metagradient framework for learning useful intrinsic reward functions across multiple lifetimes of experience. Through several proof-of-concept experiments, we show that it is feasible to learn and capture knowledge about long-term exploration and exploitation into a reward function. Furthermore, we show that unlike policy transfer methods that capture "how" the agent should behave, the learned reward functions can generalise to other kinds of agents and to changes in the dynamics of the environment by capturing "what" the agent should strive to do.

Adaptive Pairwise Weights for Temporal Credit Assignment (Chapter 5)

This chapter addresses temporal credit assignment in RL, i.e., how much credit (or blame) an action should taken in a state get for a future reward. One of the earliest and still most widely used heuristics is to assign this credit based on a scalar coefficient λ (treated as a hyperparameter) raised to the power of the time interval between the state-action and the reward. In this chapter, we explore heuristics based on more general pairwise weightings that are functions of the state in which the action was taken, the state at the time of the reward, as well as the time interval between the two. To set the pairwise weights properly, we develop a metagradient algorithm for adapting these weight functions during the usual policy optimization process. Our empirical work shows that it is often possible to learn these pairwise weight functions during learning of the policy to achieve better performance than competing approaches.

Learning State Representations from Random Deep Action-conditional Predictions (Chapter 6)

This chapter addresses state representation learning in RL. It has been demonstrated that auxiliary prediction tasks can help DeepRL agents learn better state representations. In this chapter, we investigate using random prediction tasks as auxiliary tasks for state representation learning. Our main contribution is an empirical finding that random General Value Functions (GVFs), i.e., deep action-conditional predictions—random both in what feature of observations they predict as well as in the sequence of actions the predictions are conditioned upon—form good auxiliary tasks for RL problems. In particular, we show that random deep action-conditional predictions when used as auxiliary tasks yield state representations that produce control performance competitive with state-of-the-art handcrafted auxiliary tasks in both Atari and DeepMind Lab tasks. In another set of experiments we stop the gradients from the RL part of the network to the state representation learning part of the network and show, perhaps surprisingly, that the auxiliary tasks alone are sufficient to learn state representations good enough to outperform an end-to-end trained actor-critic baseline.

Planning with Models Learned Using Generalized Value-equivalence Updates (Chapter 7)

This chapter addresses model learning. Specifically, we focus on a class of models called the value-equivalent (VE) models [Grimm et al., 2020]. Existing implementations of value-equivalence update the model by equating (a) the value of the state reached by taking a sequence of actions in the environment with (b) the value of the state reached by taking the *same* action-sequence in the model. The main contribution of this chapter is a new method called generalized value-equivalence update (G-VU) that equates (a) the value of a state reached by taking an action sequence in the environment followed by an action sequence in the model with (b) the value of a state reached by taking the *concatenation* of the two action sequences entirely in the model. Crucially, G-VU is not restricted to update only on action sequences experienced in the environment. We combine G-VU with the state-of-the-art MuZero and build a new agent called G-VUZero. In particular, G-VUZero updates the model on action sequences that are likely queried by the downstream MCTS planner. Our empirical results show that G-VUZero outperforms MuZero in two planning-focused environments, Sokoban and Minipacman.

1.2 Publications and Opensource Contributions

Some of the contributions of this thesis have been published in different venues in the past few years. We also open-sourced the source code along with some of these published articles to support reproducible research. We summarize the publications and the open-source contributions below.

Chapter 3 The research work in this chapter appears in: Zeyu Zheng, Junhyuk Oh, and Satinder Singh. On learning intrinsic rewards for policy gradient methods. In *Advances in Neural Information Processing Systems*, 2018. The source code is available online at `https://github.com/Hwhitetooth/lirpg`.

Chapter 4 The research work in this chapter appears in: 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 *International Conference on Machine Learning*, 2020.

Chapter 5 The research work in this chapter appears in: Zeyu Zheng, Risto Vuorio, Richard L Lewis, and Satinder Singh. Adaptive pairwise weights for temporal credit assignment. In *Proceedings of the Thirty-Sixth AAAI Conference on Artificial Intelligence*, 2022.

Chapter 6 The research work in this chapter appears in: Zeyu Zheng, Vivek Veeriah, Risto Vuorio, Richard L Lewis, and Satinder Singh. Learning state representations from random deep action-conditional predictions. In *Advances in Neural Information Processing Systems*, 2021. The sourc ecode is available online at `https://github.com/Hwhitetooth/random_gvfs`.

CHAPTER 2

Background

This chapter provides a brief overview of some background knowledge in RL. We start by introducing Markov decision processes (MDPs) as a theoretical framework for RL and the notion of value functions and policies. Then we introduce several fundamental RL algorithms that are used by the work in the following chapters. Finally, we briefly review some modern implementations of DeepRL agents.

2.1 Markov Decision Processes

In RL, the environment is usually modeled as a Markov decision process (MDP) [Puterman, 2014, Sutton and Barto, 2018]. An MDP is a 4-tuple $(\mathcal{S}, \mathcal{A}, P, r)$ where:

- $\mathcal{S}$ is a finite set of states;
- $\mathcal{A}$ is a finite set of actions;
- $p : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ defines the transition probability to state s' upon taking action a in state s;
- $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ defines the reward of taking action a in state s.

At every time step t, the agent observes the environment state S_t, takes an action A_t, and then receives a reward R_{t+1} and the next state S_{t+1}.

2.1.1 Episodes and Returns

Many applications come with a natural way of terminating the agent-environment interaction. We say these environments are *episodic* and call the interaction sequence between the initial state and the termination an *episode*. For an episode of length T, we define the return G_t as the cumulative rewards after time step t:

$$G_t = R_{t+1} + R_{t+2} + \cdots + R_T. \tag{2.1}$$

We often find it convenient to work with the discounted return

$$G_t = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{T-t-1} R_T, \tag{2.2}$$

where $\gamma \in [0, 1)$ is a discount factor.

2.1.2 Policies and Value Functions

The agent selects actions according to its policy π. $\pi : \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$ maps a state to a probability distribution over the action space. We often denote the probability of taking action a in state s as $\pi(a|s)$. The value function of a policy π is defined as

$$v^\pi(s) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t R_{t+1} | S_0 = s \right], \tag{2.3}$$

Here we use $\mathbb{E}_\pi[X]$ to denote the expectation of the random variable X when the agent's actions are sampled from the policy π. The constant dependency on the environment dynamics P is omitted for brevity. The value $v^\pi(s)$ of a state s denotes the expected cumulative rewards for the agent when following policy π. Similarly, the action-value function is defined as

$$q^\pi(s, a) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t R_{t+1} | S_0 = s, A_0 = a \right], \tag{2.4}$$

which denotes the expected cumulative rewards for the agent when the agent first takes action a and then follows policy π in state s. The advantage function is the difference between the action-value function and the state value function:

$$\Psi^\pi(s, a) = q^\pi(s, a) - v^\pi(s). \tag{2.5}$$

2.1.3 Optimal Policies and Optimal Value Functions

There exists at least one optimal policy whose state value is greater than or equal any other policy for all states. We use π^* to denote an optimal policy. By definition, all optimal policies share the same value function called the optimal value function v^*:

$$v^*(s) = \max_\pi v^\pi(s) \tag{2.6}$$

for all $s \in \mathcal{S}$. All optimal policies also share the same optimal action-value function q^*:

$$q^*(s,a) = \max_\pi q^\pi(s,a) \tag{2.7}$$

for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$. Given the optimal action-value function q^*, we can derive an optimal policy by taking the greedy action with respect to q^*:

$$\pi^*(s) = \arg\max_a q^*(s,a) \tag{2.8}$$

for all $s \in \mathcal{S}$. The goal of an RL agent is find an optimal policy π^* such that the values for all states are maximized.

2.2 Fundamental RL Algorithms

2.2.1 Monte-Carlo Prediction

Since the value function v^π of a policy π is usually unknown, we need to learn an approximation $\hat{v}^\pi$ instead. For brevity, we will omit the superscript unless it causes ambiguity. Learning the approximated value function $\hat{v}$ lies in the core of most RL algorithms. The most straightforward way of learning such an approximated value function is by the Monte-Carlo method. We repeatedly sample episodes by interacting with the environment following policy π. For each episode $S_0, A_0, R_1, S_1, \ldots, S_{T-1}, A_{T-1}, R_T$, we update the value function of each state in this trajectory as follows:

$$\hat{v}(S_t) \leftarrow \hat{v}(S_t) + \alpha\big[G_t - \hat{v}(S_t)\big], \tag{2.9}$$

where α is the step size or learning rate.

2.2.2 Temporal-difference Learning

Another way of learning an approximated value function is by temporal-difference (TD) learning. TD learning is one of the the most fundamental ideas in RL. In contrast to thee Monte-Carlo approach, TD does not require complete episodes to make an update. Instead, TD can update the approximated value of S_t immediately after observing S_{t+1} and R_{t+1}:

$$\hat{v}(S_t) \leftarrow \hat{v}(S_t) + \alpha\big[R_{t+1} + \gamma\hat{v}(S_{t+1}) - \hat{v}(S_t)\big]. \tag{2.10}$$

The is often called *one-step* TD. Accordingly, we call the quantity $R_{t+1} + \gamma\hat{v}(S_{t+1})$ one-step return and denote it by $G_{t:t+1}$. Moving beyond, n-step TD updates the value of S_t after n steps:

$$\hat{v}(S_t) \leftarrow \hat{v}(S_t) + \alpha\big[G_{t:t+n} - \hat{v}(S_t)\big], \tag{2.11}$$

where $G_{t:t+n} = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^n \hat{v}(S_{t+n})$ is the n-step return. We can also make a compound update by taking a convex combination of $G_{t:t+1}$, $G_{t:t+2}$, $\ldots$, $G_{t:t+T} = G_t$. A specific way of combining them with exponentially decayed weights is called λ-return:

$$G_t^{(\lambda)} = (1-\lambda)\sum_{n=1}^{T-t-1} \lambda^{n-1} G_{t:t+n} + \lambda^{T-t-1} G_t. \tag{2.12}$$

2.2.3 Policy Gradient

Policy gradient methods [Sutton et al., 2000] are a class of algorithms that directly optimize a parameterized policy π_θ. θ denotes the parameters of the policy. Recall that the objective of an RL agent is to maximize the long-term cumulative rewards:

$$J(\theta) = \mathbb{E}_\pi[\sum_{t=1}^{T} \gamma^{t-1} R_t]. \tag{2.13}$$

By the policy gradient theorem [Sutton et al., 2000], the gradient of this objective with respect to the policy parameters θ is

$$\nabla_\theta J(\theta) = \mathbb{E}_\pi\big[q(S_t, A_t)\nabla_\theta \log \pi_\theta(A_t|S_t)\big]. \tag{2.14}$$

The expectation in the above equation is hard to compute thus we often estimate it as follows. For each episode trajectory $S_0, A_0, R_1, S_1, \ldots, S_{T-1}, A_{T-1}, R_T$, the policy gradient is estimated by

$$g_\theta = \sum_{t=0}^{T-1} G_t \nabla_\theta \log \pi_\theta(A_t|S_t). \tag{2.15}$$

Then we can update the policy parameters by gradient ascent:

$$\theta \leftarrow \theta + \alpha g_\theta \tag{2.16}$$

This algorithm is often called REINFORCE [Williams, 1992].

In practice, REINFORCE often suffers high variance in the gradient estimation. A common technique for variance reduction is to subtract a baseline from the return:

$$g_\theta = \sum_{t=0}^{T-1} \big[G_t - b(S_t)\big] \nabla_\theta \log \pi_\theta(A_t|S_t). \tag{2.17}$$

The baseline function $b(S_t)$ can be any function as long as it does not depend on the action A_t or any quantity that comes after time step t. One can show that the value function v^π is the optimal baseline function for variance reduction. In that case, the policy gradient in Eq. 2.14 can be rewritten as

$$\begin{aligned} \nabla_\theta J(\theta) =& \mathbb{E}_\pi \Big[\big(q(S_t, A_t) - v(S_t)\big) \nabla_\theta \log \pi_\theta(A_t|S_t) \Big] \\ =& \mathbb{E}_\pi \Big[\Psi(S_t, A_t) \nabla_\theta \log \pi_\theta(A_t|S_t) \Big]. \end{aligned} \tag{2.18}$$

In practice, we often use the approximated value function $\hat{v}$ instead because the true value function v^π is unknown. The approximated policy gradient is computed as

$$g_\theta = \sum_{t=0}^{T-1} \big[G_t - \hat{v}(S_t)\big] \nabla_\theta \log \pi_\theta(A_t|S_t). \tag{2.19}$$

We denote

$$\hat{\Psi}_t^{\text{MC}} = G_t - \hat{v}(S_t) \tag{2.20}$$

and call $\hat{\Psi}_t^{\text{MC}}$ the Monte-Carlo estimate of the advantage $\Psi(S_t, A_t)$.

2.2.4 Actor-critic

We can combine policy gradient with TD learning by replacing the return G_t with an 1-step return $G_{t:t+1}$. The resulting algorithm is called 1-step actor-critic [Sutton et al., 2000]. 1-step actor-critic learns both an approximated value function $\hat{v}$ and a parameterized policy π_θ and updates them at every time step as follows:

$$\begin{aligned} \theta \leftarrow& \theta + \alpha_\pi \big[G_{t:t+1} - \hat{v}(S_t)\big] \nabla_\theta \log \pi_\theta(A_t|S_t) \\ \hat{v}(S_t) \leftarrow& \hat{v}(S_t) + \alpha_v \big[G_{t:t+1} - \hat{v}(S_t)\big]. \end{aligned} \tag{2.21}$$

α_π and α_v denote the learning rates for the policy and the approximated value function respectively. We can easily generalize to n-step actor-critic by replacing the 1-step return $G_{t:t+1}$ in Eq. 2.21 with the n-step return $G_{t:t+n}$. We can of course use the λ-return $G_t^{(\lambda)}$ as well. In that case, we denote

$$\hat{\Psi}_t^{(\lambda)} = G_t^{(\lambda)} - \hat{v}(S_t) \tag{2.22}$$

and call $\hat{\Psi}_t^{(\lambda)}$ the λ-estimate of $\Psi(S_t, A_t)$.

2.3 Modern DeepRL Implementations of RL Algorithms

Thanks to the advances in modern hardware design and the development of complex benchmark environments [Bellemare et al., 2013, Duan et al., 2016a, Beattie et al., 2016] in the past decade, we are able to apply RL to problems with high-dimensional state spaces. In such cases, it is infeasible to store or update the value and policy for each individual state. We must embrace function approximation. In modern DeepRL implementations, we often parameterize the approximated value function $\hat{v}$ and the policy π with deep neural networks. For example, in environments where the observations are images, we often use convolutional neural networks (CNNs) [LeCun et al., 1998] to parameterize $\hat{v}$ and π.

Modern deep neural networks are extremely computation demanding thus their training often requires specialized hardware such graphic processing units (GPUs) and corresponding auto-differentiation software tools [Abadi et al., 2016, Paszke et al., 2019, Bradbury et al., 2018]. As part of the process, the DeepRL community adapted to some of the DL terminologies so that we can benefit from the DL infrastructures more easily. One prominent instance is that instead of defining updates, we often define loss functions and the actual updates are computed by stochastic gradient descent (SGD). For example, we often define the loss function for policy gradient as

$$J(\theta) = \mathbb{E}_\pi \left[G_t \log \pi_\theta(A_t|S_t) \right]. \tag{2.23}$$

It is worth highlighting that the TD update (Eq. 2.10 and Eq. 2.11) is not the gradient of any function thus cannot be expressed by a loss function. To overcome this inconvenience, we often define a mean-squared loss but apply a stop-gradient operation on the returns to recover the correct TD update:

$$\mathcal{L}^{\text{TD}} = \left[\hat{v}(S_t) - \text{SG}(G_{t:t+n})\right]^2, \tag{2.24}$$

where $\text{SG}(\cdot)$ denotes the stop-gradient operation.

Modern deep neural networks are also data demanding. In supervised learning, people usually need to collect a large scale dataset before they can train the neural networks. Due to the online nature of RL, we must generate trajectory data on-the-fly, which makes the agent-environment interaction process the bottleneck of the training pipeline. One solution to this problem is to instantiate multiple parallel interaction processes to increase the throughput of data generation. Each interaction process is often called an *actor*.

2.3.1 Parallel Advantage Actor-critic

A popular DeepRL agent called parallel advantage actor-critic (A2C) [Mnih et al., 2016] is used for the work in Chapter 3, 4, 5, and 6. Here we provide an introduction of A2C. The work in Chapter 7 uses a different agent called MuZero [Schrittwieser et al., 2020]. We will delay the introduction of MuZero to Chapter 7 as it is not related other chapters.

Algorithm 1 provides an overview of the algorithm. We learn a policy π_θ and a value function $\hat{v}_\phi$ and both of them are parameterized by deep neural networks. The weights of the networks are denoted by θ and ϕ respectively. We use B parallel actors to generate the training trajectories. At each iteration, we first run each actor for n steps in parallel and collect B n-step trajectories. Then we compute the policy gradient loss for the policy on each trajectory and take the summation of them:

$$J(\theta) = \sum_{b=1}^{B} \sum_{i=t}^{t+n-1} \left[G_{i:t+n}^{(b)} - \hat{v}(s_i^{(b)})\right] \log \pi_\theta(a_i^{(b)}|s_i^{(b)}). \tag{2.25}$$

In addition, we also compute an entropy regularization for the policy:

$$\mathcal{L}^{\text{ent}}(\theta) = \sum_{b=1}^{B} \sum_{i=t}^{t+n-1} \mathcal{H}\left[\pi_\theta(\cdot|s_i^{(b)})\right], \tag{2.26}$$

where

$$\mathcal{H}[\pi_\theta(\cdot|s)] = -\sum_{a\in\mathcal{A}} \pi_\theta(a|s) \log \pi_\theta(a|s) \tag{2.27}$$

denotes the entropy of the probability distribution over the action space in state s. The purpose of this regularization is to encourage the policy to stay stochastic and avoid collapsing to a (nearly) deterministic policy. So the overall loss function for the policy is

$$\mathcal{L}^{\pi}(\theta) = -J(\theta) - \beta\mathcal{L}^{\text{ent}}(\theta) \tag{2.28}$$

where β is a scalar coefficient that controls the strength of the regularization. After that, we also compute the TD loss for the value function:

$$\mathcal{L}^{v}(\phi) = \sum_{b=1}^{B} \sum_{i=t}^{t+n-1} \left[\hat{v}_\phi(s_t^{(b)}) - \text{SG}(G_{i:t+n}^{(b)})\right]^2 \tag{2.29}$$

Lastly, we update the weights of the neural networks by stochastic gradient descent to minimize

Algorithm 1 Parallel Advantage Actor-critic

```
Input TD steps $n$, number of parallel actors $B$, entropy regularization $\beta$
Randomly initialize the policy network with parameters $\theta$
Randomly initialize the value network with parameters $\phi$
Initialize $B$ parallel actors
repeat
    Collect one $n$-step trajectory from each actor: $\{s_t^{(i)}, a_t^{(i)}, r_{t+1}^{(i)}, s_{t+1}^{(i)}, \dots, s_{t+n}^{(i)}\}_{i=1}^{B}$
    Compute the policy gradient loss $J(\theta) = \sum_{b=1}^{B} \sum_{i=t}^{t+n-1} \left[G_{i:t+n}^{(b)} - \hat{v}(s_i^{(b)})\right] \log \pi_\theta(a_i^{(b)} | s_i^{(b)})$
    Compute the entropy regularization $\mathcal{L}^{\text{ent}}(\theta) = \sum_{b=1}^{B} \sum_{i=t}^{t+n-1} \mathcal{H}[\pi_\theta(\cdot | s_i^{(b)})]$
    Compute the policy loss $\mathcal{L}^{\pi}(\theta) = -J(\theta) - \beta \mathcal{L}^{\text{ent}}(\theta)$
    Compute the value loss $\mathcal{L}^{v}(\phi) = \sum_{b=1}^{B} \sum_{i=t}^{t+n-1} [\hat{v}_\phi(s_t^{(b)}) - \text{SG}(G_{i:t+n}^{(b)})]^2$
    Update $\theta$ and $\phi$ by stochastic gradient descent to minimize $\mathcal{L}^{\pi}$ and $\mathcal{L}^{v}$ respectively
until done
```

the policy loss and the value loss respectively. This loop repeats until training terminates.

CHAPTER 3

On Learning Intrinsic Rewards for Policy Gradient Methods

One of the challenges facing an agent-designer in formulating a sequential decision making task as a reinforcement learning (RL) problem is that of defining a reward function. In some cases a choice of reward function is clear from the designer's understanding of the task. For example, in board games such as Chess or Go the notion of win/loss/draw comes with the game definition, and in Atari games there is a game score that is part of the game. In other cases there may not be any clear choice of reward function. For example, in domains in which the agent is interacting with humans in the environment and the objective is to maximize human-satisfaction it can be hard to define a reward function. Similarly, when the task objective contains multiple criteria such as minimizing energy consumption and maximizing throughput and minimizing latency, it is not clear how to combine these into a single scalar-valued reward function.

Even when a reward function can be defined, it is not unique in the sense that certain transformations of the reward function, e.g., adding a potential-based reward [Ng et al., 1999], will not change the resulting ordering over agent behaviors. While the choice of potential-based or other (policy) order-preserving reward function used to transform the original reward function does not change what the optimal policy is, it can change for better or for worse the sample (and computational) complexity of the RL agent learning from experience in its environment using the transformed reward function.

Yet another aspect to the challenge of reward-design stems from the observation that in many complex real-world tasks an RL agent is simply not going to learn an optimal policy because of various bounds (or limitations) on the agent-environment interaction (e.g., inadequate memory, representational capacity, computation, training data, etc.). Thus, in addressing the reward-design problem one may want to consider transformations of the task-specifying reward function that change the optimal policy. This is because it could result in the bounded-agent achieving a more desirable (to the agent designer) policy than otherwise. This is often done in the form of shaping reward functions that are less sparse than an original reward function and so lead to faster learning of a good policy even if it in principle changes what the theoretically optimal policy might be [Rajeswaran et al., 2017]. Other examples of transforming the reward function to aid learning in RL agents is the use of exploration bonuses, e.g., count-based reward bonuses for agents that encourage experiencing infrequently visited states [Bellemare et al., 2016, Ostrovski et al., 2017, Tang et al., 2017].

The above challenges make reward-design difficult, error-prone, and typically an iterative process. Reward functions that *seem* to capture the designer's objective can sometimes lead to unexpected and undesired behaviors. Phenomena such as reward-hacking [Amodei et al., 2016] illustrate this vividly. There are many formulations and resulting approaches to the problem of reward-design including preference elicitation, inverse RL, intrinsically motivated RL, optimal rewards, potential-based shaping rewards, more general reward shaping, and mechanism design; often the details of the formulation depends on the class of RL domains being addressed. In this chapter we build on the optimal rewards problem formulation of Singh et al. [2010]. We discuss the optimal rewards framework as well as some other approaches for learning intrinsic rewards in Section 3.1.

Our main contribution in this chapter is the derivation of a new stochastic-gradient-based method for learning parametric *intrinsic* rewards that when added to the task-specifying (hereafter *extrinsic*) rewards can improve the performance of policy-gradient based learning methods for solving RL problems. The policy-gradient updates the policy parameters to optimize the sum of the extrinsic and intrinsic rewards, while simultaneously our method updates the intrinsic reward parameters to optimize the extrinsic rewards achieved by the policy. We evaluate our method on several Atari games with a state of the art A2C (Advantage Actor-Critic) [Mnih et al., 2016] agent as well as on a few Mujoco domains with a similarly state of the art PPO agent and show that learning intrinsic rewards can outperform using just extrinsic reward as well as using a combination of extrinsic reward and a constant "live bonus" [Duan et al., 2016a]. On Atari games, we also compared our method with a count-based methods, i.e., pixel-SimHash [Tang et al., 2017]. Our method showed better performance.

3.1 Related Work

Optimal rewards and reward design. Our work builds on the Optimal Reward Framework [Singh et al., 2010]. Formally, the optimal intrinsic reward for a specific combination of RL agent and environment is defined as the reward that when used by the agent for its learning in its environment maximizes the extrinsic reward. The main intuition is that in practice all RL agents are bounded (computationally, representationally, in terms of data availability, etc.) and the optimal intrinsic reward can help mitigate these bounds. Computing the optimal reward remains a big challenge, of course. The paper introducing the framework used exhaustive search over a space of intrinsic reward functions and thus does not scale. Sorg et al. [2010] introduced PGRD (Policy Gradient for Reward Design), a scalable algorithm that only works with lookahead-search (e.g., UCT) based planning agents (and hence the agent itself is not a learning-based agent; only the reward to use with the fixed planner is learned). Its insight was that the intrinsic reward can be treated as a parameter that influences the outcome of the planning process and thus can be trained via gradient ascent as long as the planning process is differentiable (which UCT and related algorithms are). Guo et al. [2016] extended the scalability of PGRD to high-dimensional image inputs in Atari 2600 games and used the intrinsic reward as a reward bonus to improve the performance of the Monte Carlo Tree Search algorithm using the Atari emulator as a model for the planning. A big open challenge is deriving a sound algorithm for learning intrinsic rewards for learning-based RL agents and showing that it can learn intrinsic rewards fast enough to beneficially influence the online performance of the learning based RL agent. Our main contribution in this chapter is to answer this challenge.

Reward shaping and Auxiliary rewards. Reward shaping [Ng et al., 1999] provides a general answer to what space of reward function modifications do not change the optimal policy, specifically potential-based rewards. Other attempts have been made to design auxiliary rewards with desired properties. For example, the UNREAL agent [Jaderberg et al., 2017] used pseudo-reward computed from unsupervised auxiliary tasks to refine its internal representations. In Bellemare et al. [2016], Ostrovski et al. [2017], and Tang et al. [2017], a pseudo-count based reward bonus was given to the agent to encourage exploration. Pathak et al. [2017] used self-supervised prediction errors as intrinsic rewards to help the agent explore. In these and other similar examples [Schmidhuber, 2010, Stadie et al., 2015, Oudeyer and Kaplan, 2009], the agent's learning performance improves through the reward transformations, but the reward transformations are expert-designed and not learned. The main departure point in this chapter is that we learn the parameters of an intrinsic reward function that maps high-dimensional observations and actions to rewards.

Hierarchical RL. Another approach to a form of intrinsic reward is in the work on hierarchical RL. For example, FeUdal Networks (FuNs) [Vezhnevets et al., 2017] is a hierarchical architecture with a Manager and a Worker learning at different time scales. The Manager conveys abstract goals to the Worker and the Worker optimizes its policy to maximize the extrinsic reward and the cosine distance to the goal. The Manager optimizes its proposed goals to guide the Worker to learn a better policy in terms of the cumulative extrinsic reward. A large body of work on hierarchical RL also generally involves a higher level module choosing goals for lower level modules. All of this work can be viewed as a special case of creating intrinsic rewards within a multi-module agent architecture. One special aspect of hierarchical-RL work is that these intrinsic rewards are usually associated with goals of achievement, i.e., achieving a specific goal state while in our setting the intrinsic reward functions are general mappings from observation-action pairs to rewards. Another special aspect is that most evaluations of hierarchical RL show a benefit in the transfer setting with typically worse performance on early tasks while the manager is learning and better performance on later tasks once the manager has learned. In our setting we take on the challenge of showing that learning and using intrinsic rewards can help the RL agent perform better while it is learning on a single task. Finally, another difference is that hierarchical RL typically treats the lower-level learner as a black box while we train the intrinsic reward using gradients through the policy module in our architecture.

Meta RL. Our work in this chapter can be viewed as an instance of meta learning [Andrychowicz et al., 2016, Santoro et al., 2016, Nichol and Schulman, 2018] in the sense that the intrinsic reward function module acts as a meta-learner that learns to improve the agent's objective (i.e., mixture of extrinsic and intrinsic reward) by taking into account how each gradient step of the agent affects the true objective (i.e., extrinsic reward) through the meta-gradient. However, a key distinction from the prior work on meta learning for RL [Finn et al., 2017a, Duan et al., 2017, Wang et al., 2016a, Duan et al., 2016b] is that our method aims to meta-learn intrinsic rewards within a single task, whereas much of the prior work is designed to quickly adapt to new tasks in a few-shot learning scenario. Xu et al. [2018b] concurrently proposed a similar idea that learns to find meta-parameters (e.g., discount factor) such that the agent can learn more efficiently within a single task. In contrast to state-independent meta-parameters in [Xu et al., 2018b], we propose a richer form of state-dependent meta-learner (i.e., intrinsic rewards) that directly changes the reward function of the agent, which can be potentially extended to hierarchical RL.

3.2 Gradient-Based Learning of Intrinsic Rewards

As noted earlier, the most practical previous work in learning intrinsic rewards using the Optimal Rewards framework was limited to settings where the underlying RL agent was a planning (i.e., needs a model of the environment) agent that use lookahead search in some form (e.g, UCT). In these settings the only quantity being learned was the intrinsic reward function. By contrast, in this section we derive our algorithm for learning intrinsic rewards for the setting where the underlying RL agent is itself a learning agent, specifically a policy gradient based learning agent.

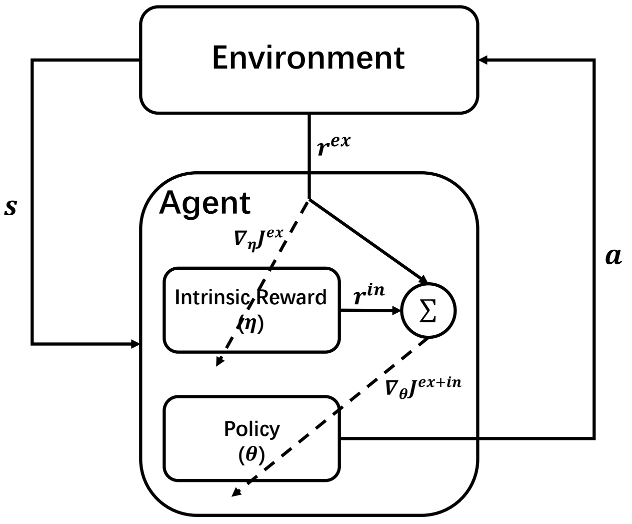


Figure 3.1: The LIRPG agent architecture. Inside the agent are two modules, a policy function parameterized by θ and an intrinsic reward function parameterized by η. In our experiments the policy function (A2C / PPO) has an associated value function as does the intrinsic reward function . As shown by the dashed lines, the policy module is trained to optimize the weighted sum of intrinsic and extrinsic rewards while the intrinsic reward module is trained to optimize just the extrinsic rewards.

3.2.1 LIRPG: Learning Intrinsic Rewards for Policy Gradient

Notation. We use the following notation throughout.

- θ: policy parameters
- η: intrinsic reward parameters
- r^{ex}: extrinsic reward from the environment
- $r^{in}_\eta = r^{in}_\eta(s, a)$: intrinsic reward estimated by η
- $G^{ex}(s_t, a_t) = \sum_{i=t}^{\infty} \gamma^{i-t} r^{ex}_i$
- $G^{in}(s_t, a_t) = \sum_{i=t}^{\infty} \gamma^{t-i} r^{in}_\eta(s_i, a_i)$
- $G^{ex+in}(s_t, a_t) = \sum_{i=t}^{\infty} \gamma^{i-t}(r^{ex}_i + \lambda r^{in}_\eta(s_i, a_i))$
- $J^{ex} = \mathbb{E}_\theta[\sum_{t=0}^{\infty} \gamma^t r^{ex}_t]$
- $J^{in} = \mathbb{E}_\theta[\sum_{t=0}^{\infty} \gamma^t r^{in}_\eta(s_t, a_t)]$
- $J^{ex+in} = \mathbb{E}_\theta[\sum_{t=0}^{\infty} \gamma^t (r^{ex}_t + \lambda r^{in}_\eta(s_t, a_t)]$

- λ: relative weight of intrinsic reward.

The departure point of our approach to reward optimization for policy gradient is to distinguish between the extrinsic reward, r^{ex}, that defines the task, and a separate intrinsic reward r^{in} that additively transforms the extrinsic reward and influences learning via policy gradients. It is crucial to note that the ultimate measure of performance we care about improving is the value of the extrinsic rewards achieved by the agent; the intrinsic rewards serve only to influence the change in policy parameters. Figure 3.1 shows an abstract representation of our intrinsic reward augmented policy gradient based learning agent.

Algorithm Overview. An overview of our algorithm, LIRPG, is presented in Algorithm 2. At each iteration of LIRPG, we simultaneously update the policy parameters θ and the intrinsic reward parameters η. More specifically, we first update θ in the direction of the gradient of J^{ex+in} which is the weighted sum of intrinsic and extrinsic rewards. After updating policy parameters, we update η in the direction of the gradient of J^{ex} which is just the extrinsic rewards. Intuitively, the policy is updated to maximize the sum of extrinsic and intrinsic rewards, while the intrinsic reward function is updated to maximize only the extrinsic reward. We describe more details of each step below.

Updating Policy Parameters (θ). Given an episode where the behavior is generated according to policy π_θ, we update the policy parameters using regular policy gradient using the sum of intrinsic and extrinsic rewards as the reward:

$$\begin{aligned} \theta' &= \theta + \alpha \nabla_\theta J^{ex+in}(\theta) && (3.1) \\ &\approx \theta + \alpha G^{ex+in}(s_t, a_t) \nabla_\theta \log \pi_\theta(a_t|s_t), && (3.2) \end{aligned}$$

where Equation 3.2 is a stochastic gradient update.

Updating Intrinsic Reward Parameters (η). Given an episode and the updated policy parameters θ', we update intrinsic reward parameters. Intuitively, updating η requires estimating the effect such a change would have on the extrinsic value through the change in the policy parameters. Our key idea is to use the chain rule to compute the gradient as follows:

$$\nabla_\eta J^{ex} = \nabla_{\theta'} J^{ex} \nabla_\eta \theta', \tag{3.3}$$

where the first term ($\nabla_{\theta'} J^{ex}$) sampled as

$$\nabla_{\theta'} J^{ex} \approx G^{ex}(s_t, a_t) \nabla_{\theta'} \log \pi_{\theta'}(a_t|s_t) \tag{3.4}$$

Algorithm 2 LIRPG: Learning Intrinsic Reward for Policy Gradient

```
Input: step-size parameters $\alpha$ and $\beta$
Init: initialize $\theta$ and $\eta$ with random values
repeat
    Sample a trajectory $\mathcal{D} = \{s_0, a_0, s_1, a_1, \cdots\}$ by interacting with the environment using $\pi_\theta$
    Approximate $\nabla_\theta J^{ex+in}(\theta; \mathcal{D})$ by Equation 3.2
    Update $\theta' \leftarrow \theta + \alpha \nabla_\theta J^{ex+in}(\theta; \mathcal{D})$
    Approximate $\nabla_{\theta'} J^{ex}(\theta'; \mathcal{D})$ on $\mathcal{D}$ by Equation 3.9
    Approximate $\nabla_\eta \theta'$ by Equation 3.8
    Compute $\nabla_\eta J^{ex} = \nabla_{\theta'} J^{ex}(\theta'; \mathcal{D}) \nabla_\eta \theta'$
    Update $\eta' \leftarrow \eta + \beta \nabla_\eta J^{ex}$
until done
```

is an approximate stochastic gradient of the extrinsic value with respect to the updated policy parameters θ' when the behavior is generated by $\pi_{\theta'}$, and the second term can be computed as follows:

$$\begin{aligned}
\nabla_\eta \theta' &= \nabla_\eta \left(\theta + \alpha G^{ex+in}(s_t, a_t) \nabla_\theta \log \pi_\theta(a_t|s_t)\right) && (3.5)\\
&= \nabla_\eta \left(\alpha G^{ex+in}(s_t, a_t) \nabla_\theta \log \pi_\theta(a_t|s_t)\right) && (3.6)\\
&= \nabla_\eta \left(\alpha \lambda G^{in}(s_t, a_t) \nabla_\theta \log \pi_\theta(a_t|s_t)\right) && (3.7)\\
&= \alpha \lambda \sum_{i=t}^{\infty} \gamma^{i-t} \nabla_\eta r_\eta^{in}(s_i, a_i) \nabla_\theta \log \pi_\theta(a_t|s_t). && (3.8)
\end{aligned}$$

Note that to compute the gradient of the extrinsic value J^{ex} with respect to the intrinsic reward parameters η, we needed a new episode with the updated policy parameters θ' (cf. Equation 3.4), thus requiring two episodes per iteration. To improve data efficiency we instead *reuse* the episode generated by the policy parameters θ at the start of the iteration and correct for the resulting mismatch by replacing the on-policy update in Equation 3.4 with the following off-policy update using importance sampling:

$$\nabla_{\theta'} J^{ex} = G^{ex}(s_t, a_t) \frac{\nabla_{\theta'} \pi_{\theta'}(a_t|s_t)}{\pi_\theta(a_t|s_t)}. \tag{3.9}$$

The parameters η are updated using the product of Equations 3.8 and 3.9 with a step-size parameter β; this approximates a stochastic gradient update (cf. Equation 3.3).

Implementation on A2C and PPO. We described LIRPG using the most basic policy gradient formulation for simplicity. There have been many advances in policy gradient methods that reduce the variance of the gradient and improve the data-efficiency. Our LIRPG algorithm is also compatible with such actor-critic architectures. Specifically, for our experiments on Atari games we used a reasonably state of the art advantage actor-critic (A2C) architecture, and for our experiments on Mujoco domains we used a similarly reasonably state of the art proximal policy optimization (PPO) architecture. [1]

3.3 Experiments on Atari Games

Our overall objective in the following first set of experiments is to evaluate whether augmenting a policy gradient based RL agent with intrinsic rewards learned using our LIRPG algorithm (henceforth, augmented agent in short) improves performance relative to the baseline policy gradient based RL agent that uses just the extrinsic reward (henceforth, A2C baseline agent in short). To this end, we first perform this evaluation on multiple Atari games from the Arcade Learning Environment (ALE) platform [Bellemare et al., 2013] using the same open-source implementation with exactly the same hyper-parameters of the A2C algorithm [Mnih et al., 2016] from OpenAI [Dhariwal et al., 2017] for both our augmented agent as well as the baseline agent. The extrinsic reward used is the game score change as is standard for the work on Atari games. The LIRPG algorithm has two additional parameters relative to the baseline algorithm, the parameter λ that controls how the intrinsic reward is scaled before adding it to the extrinsic reward and the step-size β; we describe how we choose these parameters below in our results.

We also conducted experiments against two other baselines. The first baseline simply added a constant positive value as a live bonus to the agent's reward at each time step (henceforth, A2C-live-bonus baseline agent in short). The live bonus heuristic encourages the agent to live longer so that it will potentially have a better chance of getting extrinsic rewards. The second baseline augmented the agent with a count-based bonus generated by the pixel-SimHash algorithm [Tang et al., 2017] (henceforth, A2C-pixel-SimHash baseline agent in short.)

Note that the policy module inside the agent is really two networks, a policy network and a value function network (that helps estimate G^{ex+in} as required in Equation 3.2). Similarly the intrinsic reward module in the agent is also two networks, a reward function network and a value function network (that helps estimate G^{ex} as required in Equation 3.4).

3.3.1 Implementation Details

The intrinsic reward module has two very similar neural network architectures as the policy module described above. It has a "policy" network that instead of a softmax over actions produces a scalar reward for every action through a tanh nonlinearity to keep the scalar output in $[-1, 1]$; we will refer to it as the intrinsic reward network. It also has a value network that estimates G^{ex}; this has

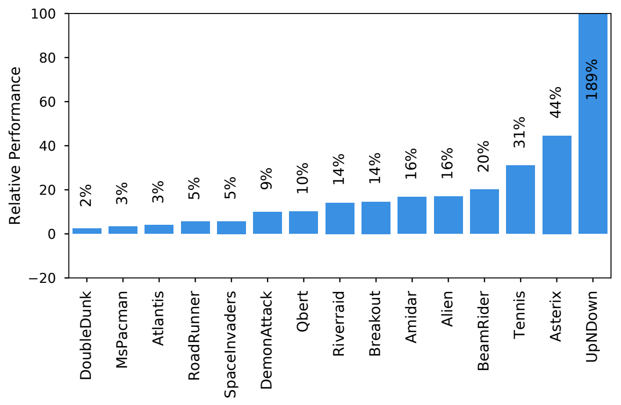
[1]Our implementation is available at: https://github.com/Hwhitetooth/lirpg

the same architecture as the intrinsic reward network except for the output layer that has a single scalar output without a non-linear activation. These two networks share the parameters of the first four layers with each other. We keep the default values of all hyper-parameters in the original OpenAI implementation of the A2C-based policy module unchanged for both the augmented and baseline agents. We use RMSProp to optimize the two networks of the intrinsic reward module. Recall that there are two parameters special to LIRPG. Of these, the step size β was initialized to 0.0007 and annealed linearly to zero over 50 million time steps for all the experiments reported below. We did a small hyper-parameter search for λ for each game (described below).

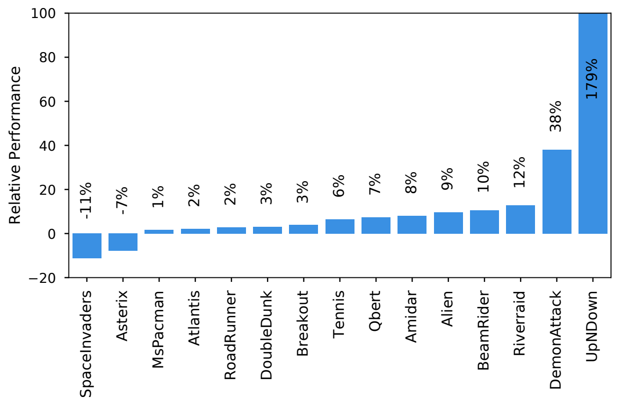
3.3.2 Overall Performance

Figure 3.2 shows the improvements of the augmented agents over baseline agents on 15 Atari games: Alien, Amidar, Asterix, Atlantis, BeamRider, Breakout, DemonAttack, DoubleDunk, MsPacman, Qbert, Riverraid, RoadRunner, SpaceInvaders, Tennis, and UpNDown. We picked as many games as our computational resources allowed in which the published performance of the underlying A2C baseline agents was good but where the learning was not so fast in terms of sample complexity so as to leave little room for improvement. We ran each agent for 5 separate runs each for 50 million time steps on each game for both the baseline agents and augmented agents. For the augmented agents, we explored the following values for the intrinsic reward weighting coefficient λ, $\{0.003, 0.005, 0.01, 0.02, 0.03, 0.05\}$ and the following values for the term ξ, $\{0.001, 0.01, 0.1, 1\}$, that weights the loss from the value function estimates with the loss from the intrinsic reward function (the policy component of the intrinsic reward module). We plotted the best results from the hyper-parameter search in Figure 3.2. For the A2C-live-bonus baseline agents, we explored the value of live bonus over the set $\{0.001, 0.01, 0.1, 1\}$ on two games, Amidar and MsPacman, and chose the best performing value of 0.01 for all 15 games. For the A2C-pixel-SimHash baseline agents, we adopted all hyper-parameters from [Tang et al., 2017].

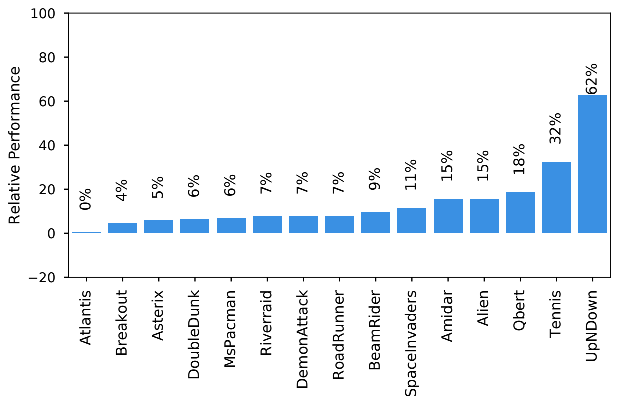
The blue bars in Figure 3.2 show the human score normalized improvements of the augmented agents over the A2C baseline agents, the A2C-live-bonus baseline agents, and the A2C-pixel-SimHash baseline agents. We see that the augmented agent outperforms the A2C baseline agent on all 15 games and has an improvement of more than ten percent on 9 out of 15 games. As for the comparison to the A2C-live-bonus baseline agent, the augmented agent still performed better on all games except for SpaceInvaders and Asterix. Note that most Atari games are shooting games so the A2C-live-bonus baseline agent is expected to be a stronger baseline. The augmented agent outperformed or was comparable to the A2C-pixel-SimHash baseline agent on all 15 games.



(a)



(b)



(c)

Figure 3.2: (a) Improvements of LIRPG augmented agents over A2C baseline agents. (b) Improvements of LIRPG augmented agents over live-bonus augmented A2C baseline agents. (c) Improvements of LIRPG augmented agents over pixel-SimHash augmented A2C baseline agents. In all figures, the columns correspond to different games labeled on the x-axes and the y-axes show human score normalized improvements.

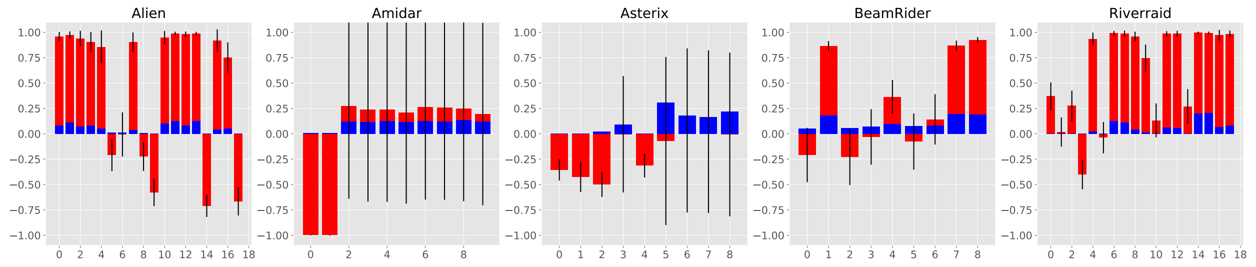


Figure 3.3: Intrinsic reward variation and frequency of action selection. For each game/plot the x-axis shows the index of the actions that are available in that game. The red bars show the means and standard deviations of the intrinsic rewards associated with each action. The blue bars show the frequency of each action being selected.

3.3.3 Analysis of the Learned Intrinsic Reward

An interesting question is whether the learned intrinsic reward function learns a general state-independent bias over actions or whether it is an interesting function of state. To explore this question we used the learned intrinsic reward module and the policy module from the end of a good run (cf. Figure 3.2) for each game with no further learning to collect new data for each game. Figure 3.3 shows the variation in intrinsic rewards obtained and the actions selected by the agent over 100 thousand steps, i.e., 400 thousand frames, on 5 games. The red bars show the average intrinsic reward per-step for each action. The black segments show the standard deviation of the intrinsic rewards. The blue bars show the frequency of each action being selected. Figure 3.3 shows that the intrinsic rewards for most actions vary through the episode as shown by large black segments, indirectly confirming that the intrinsic reward module learns more than a state-independent constant bias over actions. By comparing the red bars and the blue bars, we see the expected correlation between aggregate intrinsic reward over actions and their selection (through the policy module that trains on the weighted sum of extrinsic and intrinsic rewards).

3.4 Mujoco Experiments

Our main objective in the following experiments is to demonstrate that our LIRPG-based algorithm can extend to a different class of domains and a different choice of baseline actor-critic architecture (namely, PPO instead of A2C). Specifically, we explore domains from the Mujoco continuous control benchmark [Duan et al., 2016a], and used the open-source implementation of the PPO [Schulman et al., 2017] algorithm from OpenAI [Dhariwal et al., 2017] as our baseline agent. We also compared LIRPG to the simple heuristic of giving a live bonus as intrinsic reward

(PPO-live-bonus baseline agents for short). As for the Atari game results above, we kept all hyper-parameters unchanged to default values for the policy module of both baseline and augmented agents. Finally, we also conduct a preliminary exploration into the question of how robust the learning of intrinsic rewards is to the sparsity of extrinsic rewards. Specifically, we used the *delayed* versions of the Mujoco domains, where the extrinsic reward is made sparse by accumulating the reward for $N = 10, 20, 40$ time steps before providing it to the agent. Note that the live bonus is not delayed when we delay the extrinsic reward for the PPO-live-bonus baseline agent. We expect that the problem becomes more challenging with increasing N but expect that the learning of intrinsic rewards (that are available at every time step) can help mitigate some of that increasing hardness.

Delayed Mujoco benchmark. We evaluated 5 environments from the Mujoco benchmark, i.e., Hopper, HalfCheetah, Walker2d, Ant, and Humanoid. As noted above, to create a more-challenging sparse-reward setting we accumulated rewards for 10, 20 and 40 steps (or until the end of the episode, whichever comes earlier) before giving it to the agent. We trained the baseline and augmented agents for 1 million steps on each environment.

3.4.1 Implementation Details

The intrinsic reward function networks are quite similar to the two networks in the policy module. Each network is a multi-layer perceptron (MLP) with 2 hidden layers. We concatenated the observation vector and the action vector as the input to the intrinsic reward network. The first two layers are fully connected layers with 64 hidden units. Each hidden layer is followed by a tanh non-linearity. The output layer has one scalar output. We apply tanh on the output to bound the intrinsic reward to $[-1, 1]$. The value network to estimate G^{ex} has the same architecture as the intrinsic reward network except for the output layer that has a single scalar output without a non-linear activation. These two networks do not share any parameters. We keep the default values of all hyper-parameters in the original OpenAI implementation of PPO unchanged for both the augmented and baseline agents. We use Adam [Kingma and Ba, 2014] to optimize the two networks of the intrinsic reward module. The step size β was initialized to 0.0001 and was fixed over 1 million time steps for all the experiments reported below. The mixing coefficient λ was fixed to 1.0 and instead we multiplied the extrinsic reward by 0.01 cross all 5 environments.

3.4.2 Overall Performance

Our results comparing the use of learning intrinsic reward with using just extrinsic reward on top of a PPO architecture are shown in Figure 3.4. We only show the results of a delay of 20 here

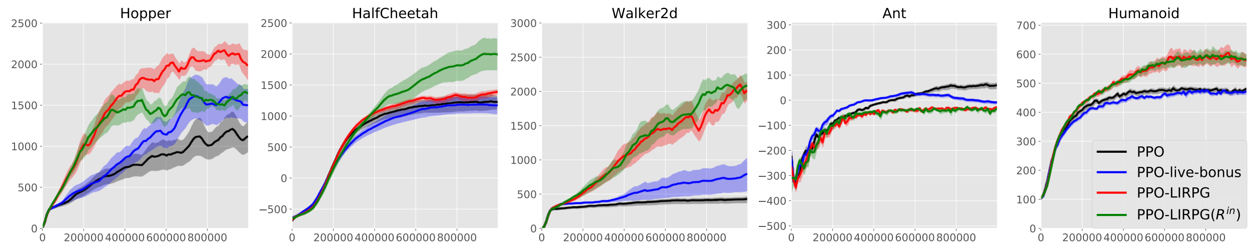


Figure 3.4: The x-axis is time steps during learning. The y-axis is the average reward over the last 100 training episodes. The black curves are for the baseline PPO architecture. The blue curves are for the PPO-live-bonus baseline. The red curves are for our LIRPG based augmented architecture. The green curves are for our LIRPG architecture in which the policy module was trained with only intrinsic rewards. The dark curves are the average of 10 runs with different random seeds. The shaded area shows the standard errors of 10 runs.

. The black curves are for PPO baseline agents. The blue curves are PPO-live-bonus baseline agents, where we explored the value of live bonus over the set $\{0.001, 0.01, 0.1, 1\}$ and plotted the curves for the domain-specific best performing choice. The red curves are for the augmented LIRPG agents.

We see that in 4 out of 5 domains learning intrinsic rewards significantly improves the performance of PPO, while in one game (Ant) we got a degradation of performance. Although a live bonus did help on 2 domains, i.e., Hopper and Walker2d, LIRPG still outperformed it on 4 out of 5 domains except for HalfCheetah on which LIRPG got comparable performance. We note that there was no domain-specific hyper-parameter optimization for the results in this figure; with such optimization there might be an opportunity to get improved performance for our method in all the domains.

Training with Only Intrinsic Rewards. We also conducted a more challenging experiment on Mujoco domains in which we used only intrinsic rewards to train the policy module. Recall that the intrinsic reward module is trained to optimize the extrinsic reward. In 3 out of 5 domains, as shown by the green curves denoted by ‘PPO-LIRPG(R^{in})’ in Figure 3.4, using only intrinsic rewards achieved similar performance to the red curves where we used a mixture of extrinsic rewards and intrinsic rewards. Using only intrinsic rewards to train the policy performed worse than using the mixture on Hopper but performed even better on HalfCheetah. It is important to note that training the policy using only live-bonus reward without the extrinsic reward would completely fail, because there would be no learning signal that encourages the agent to move forward. In contrast, our result shows that the agent can learn complex behaviors solely from the learned intrinsic reward on MuJoCo environment, and thus the intrinsic reward captures far more than a live bonus does; this is because the intrinsic reward module takes into account the extrinsic reward structure through its training.

3.5 Conclusion

Our experiments on using LIRPG with A2C on multiple Atari games showed that it helped improve learning performance in all of the 15 games we tried. Similarly using LIRPG with PPO on multiple Mujoco domains showed that it helped improve learning performance in 4 out 5 domains (for the version with a delay of 20). Note that we used the same A2C / PPO architecture and hyper-parameters in both our augmented and baseline agents.

In this chapter, we derived a novel practical algorithm, LIRPG, for learning intrinsic reward functions in problems with high-dimensional observations for use with policy gradient based RL agents. This is the first such algorithm to the best of our knowledge. Our empirical results show promise in using intrinsic reward function learning as a kind of meta-learning to improve the performance of modern policy gradient architectures like A2C and PPO.

CHAPTER 4

What Can Learned Intrinsic Rewards Capture?

Reinforcement learning (RL) agents can store knowledge in their policies, value functions, state representations, and models of the environment dynamics. These components can be the *loci* of knowledge in the sense that they are structures in which knowledge, either learned from experience by the agent's algorithm or given by the agent-designer, can be deposited and reused. The objective of the agent is defined by a reward function, and the goal is to learn to act so as to maximise cumulative rewards. In this chapter we consider the proposition that the reward function itself is a good locus of knowledge. This is unusual (but not novel) in that most prior work treats the reward as given and immutable, at least as far as the learning algorithm is concerned. In fact, agent designers often do find it convenient to modify the reward function given to the agent to facilitate learning. It is therefore useful to distinguish between two kinds of reward functions [Singh et al., 2010]: *extrinsic* rewards define the task and capture the designer's preferences over agent behaviour, whereas *intrinsic* rewards serve as helpful signals to improve the learning dynamics of the agent.

Most existing work on intrinsic rewards falls into two categories: task-dependent and task-independent. Both are typically designed by hand. Hand-designing *task-dependent* rewards can be fraught with difficulty as even minor misalignment between the actual reward and the intended bias/goals can lead to unintended and sometimes catastrophic consequences [Clark and Amodei, 2016]. *Task-independent* intrinsic rewards are also typically hand-designed, often based on an intuitive understanding of animal/human behaviour or on heuristics on desired exploratory behaviour. It can, however, be hard to match such task-independent intrinsic rewards to the specific learning dynamics induced by the interaction between agent and environment. In this chapter, we are interested in the comparatively under-explored possibility of *learned* (not hand-designed) task-dependent intrinsic rewards. Although there have been a few attempts to learn useful intrinsic rewards from experience [Singh et al., 2009, Zheng et al., 2018], how to capture complex knowledge such as exploration across episodes into a reward function remains an open question.

We emphasise that it is *not* our objective to show that rewards are a *better* locus of learned knowledge than others; the best locus likely depends on the kind of knowledge that is most useful in a given task. In particular, knowledge captured in rewards provides guidance on “what” the agent should strive to do while knowledge captured in policies provides guidance on “how” an agent should behave. Knowledge about “what” captured in rewards is indirect and thus slower to make an impact on behaviour because it takes effect through learning, while knowledge about “how” can directly have an immediate impact on behaviour. At the same time, because of its indirectness the former can generalise better to changes in dynamics and different learning agents, as we empirically show in this chapter.

How should we measure the usefulness of a learned reward function? Ideally, we would like to measure the effect the learned reward function has on the learning dynamics. Of course, learning happens over multiple episodes, indeed it happens over an entire lifetime. Therefore, we choose *lifetime return*, the cumulative extrinsic reward obtained by the agent over its entire lifetime, as the main objective. To this end, we adopt the multi-lifetime setting of the Optimal Rewards Framework [Singh et al., 2009] in which an agent is initialised randomly at the start of each lifetime and then faces a stationary or non-stationary task drawn from some distribution. In this setting, the only knowledge that is transferred across lifetimes is the reward instead of the policy. Specifically, the goal is to learn a single intrinsic reward function that, when used to adapt the agent’s policy using a standard episodic RL algorithm, ends up optimising the cumulative extrinsic reward over its lifetime.

In previous work, good reward functions were found via exhaustive search, limiting the range of applicability. We develop a more scalable gradient-based method for learning intrinsic rewards by exploiting the fact that the interaction between the policy update and the reward function is differentiable(see Chapter 3; also [Zheng et al., 2018]). Moreover, unlike the prior work, we parameterize the reward function by a recurrent neural network unrolled over the entire lifetime and train it to maximise lifetime return, which is crucial for the reward function to capture long-term temporal dependencies (e.g., novelty of states across episodes). To handle long-term credit assignment that spans the lifetime, we use a lifetime value function that estimates the remaining lifetime return.

Our main contributions and findings are as follows: (1) Through carefully designed environments, we show that learned intrinsic reward functions can capture a rich form of knowledge such as long-term exploration (e.g., exploring uncertain states) and exploitation (e.g., anticipating environment changes) across multiple episodes. To our knowledge, this is the first work that shows the feasibility of learning such complex knowledge into reward functions. (2) We show that “what to do” knowledge captured by the reward functions can generalise to changing dynamics of the environment and new learning agents, whereas policy transfer methods do not generalise well, which provides insights into the usefulness of rewards as a locus of knowledge.

4.1 Related Work

Hand-designed Rewards There is a long history of work on designing rewards to accelerate learning in reinforcement learning. Reward shaping aims to design task-specific rewards towards known optimal behaviours, typically requiring domain knowledge. Both the benefits [Randlöv and Alström, 1998, Ng et al., 1999, Harutyunyan et al., 2015] and the difficulty [Clark and Amodei, 2016] of task-specific reward shaping have been studied. On the other hand, many intrinsic rewards have been proposed to encourage exploration, inspired by animal behaviours. Examples include prediction error [Schmidhuber, 1991a,b, Oudeyer et al., 2007, Gordon and Ahissar, 2011, Mirolli and Baldassarre, 2013, Pathak et al., 2017], surprise [Itti and Baldi, 2006], deviation from a default policy [Goyal et al., 2018], weight change [Linke et al., 2019], and state-visitation counts [Sutton, 1990, Poupart et al., 2006, Strehl and Littman, 2008, Bellemare et al., 2016, Ostrovski et al., 2017]. Although these kinds of intrinsic rewards are not domain-specific, they are often not well-aligned with the task that the agent is solving, and ignore the effect on the agent's learning dynamics. In contrast, our work aims to learn intrinsic rewards from data that take into account the agent's learning dynamics without requiring prior knowledge from a human.

Rewards Learned from Experience There have been a few attempts to learn useful intrinsic rewards from data. Singh et al. [2009] introduced the Optimal Reward Framework which aims to find a good reward function that allows agents to solve a distribution of tasks using exhaustive search. The empirical study only showed simple intrinsic reward functions such as preference over certain objects due to the inefficient exhaustive search method employed. Although there have been follow-up works [Sorg et al., 2010, Guo et al., 2016] that use a gradient-based method, they consider a non-parameteric policy using Monte-Carlo Tree Search. Our work is closely related to the LIRPG algorithm in Chapter 3 which proposed a meta-gradient method to learn intrinsic rewards. However, LIRPG considers a single task in a single lifetime with a myopic episode return objective, which is limited in that it does not allow exploration across episodes or generalisation to different agents. In contrast, our approach takes into account both the long-term effect of intrinsic rewards on the learning dynamics and the lifetime history of the agent. We show this is crucial for capturing long-term knowledge, such as seeking for novel states across episodes, which is not achieved in previous work. Finally, unlike AGILE [Bahdanau et al., 2019] which showed that a learned reward function can generalise to unseen instructions in instruction-following RL problems, our work shows new and interesting kind of generalisation: to new agent-environment interfaces and algorithms.

Meta-learning for Exploration and Task Adaptation Meta-learning [Schmidhuber et al., 1996, Thrun and Pratt, 1998] has recently received considerable attention in RL. Recent advances include few-shot adaptation [Finn et al., 2017a], few-shot imitation [Finn et al., 2017b, Duan et al., 2017], model adaptation [Clavera et al., 2019], and inverse RL [Xu et al., 2019]. In particular, our work is related to the prior work on meta-learning good exploration strategies [Wang et al., 2016b, Duan et al., 2016b, Stadie et al., 2018, Xu et al., 2018a] in that both perform temporal credit assignment across episode boundaries by maximising rewards accumulated beyond an episode. Unlike the prior work that aims to directly transfer an exploratory policy, our framework indirectly drives exploration via a reward function which can be reused by different learning agents.

Meta-learning Update Rules There have been a few studies that directly meta-learn how to update the agent's parameters via meta-parameters including discount factor and returns [Xu et al., 2018b], auxiliary tasks [Schlegel et al., 2018, Veeriah et al., 2019], unsupervised learning update rules [Metz et al., 2019], and RL objectives [Bechtle et al., 2019, Kirsch et al., 2019]. Our work also belongs to this category in that our meta-parameters are the reward function used in the agent's update. In particular, our multi-lifetime formulation is similar to ML^3 [Bechtle et al., 2019] and MetaGenRL [Kirsch et al., 2019]. However, ML^3 cannot generalise to different agent-environment interfaces, whereas intrinsic rewards can as shown in Section 4.5. In addition, we propose to use the lifetime return as opposed to the myopic episodic objective of ML^3 and MetaGenRL, which is crucial for cross-episode exploration.

Cognitive Study. Several cognitive science studies on the exploration-exploitation dilemma [Cohen et al., 2007, Wilson et al., 2014] have shown that humans use both a random exploration strategy [Thompson, 1933, Watkins, 1989] and an information-seeking strategy [Gittins, 1974, 1979] when facing uncertainty. Computationally, the former can be easily implemented, whereas the latter usually requires carefully handcrafted methods to guide the agent's behaviour. In this work, we hypothesize and empirically verify that an information-seeking intrinsic reward function can naturally emerge if it is *useful* for solving the tasks. The condition of being useful resembles a recent study [Dubey and Griffiths, 2019] which posited that a rational agent should explore in a way such that the usefulness of its knowledge is maximised.

4.2 The Optimal Reward Problem

We first introduce some terminology.

- **Agent**: A learning system interacting with an environment. On each step t the agent selects an action a_t and receives from the environment an observation s_{t+1} and an *extrinsic* reward

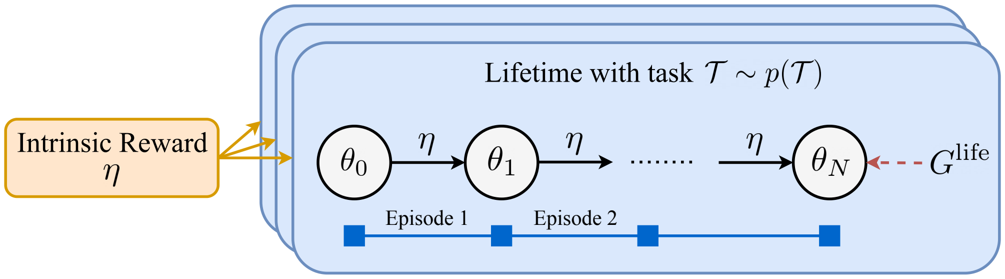


Figure 4.1: Illustration of the proposed intrinsic reward learning framework. The intrinsic reward r_η is used to update the agent's parameter θ_i throughout its lifetime which consists of many episodes. The goal is to find the optimal intrinsic reward parameters η^* across many lifetimes that maximises the lifetime return (G^{life}) given any randomly initialised agents and possibly non-stationary tasks drawn from some distribution $p(\mathcal{T})$.

r_{t+1} defined by a task $\mathcal{T}$. The agent chooses actions based on a policy $\pi_\theta(a_t|s_t)$ parameterized by θ.

- **Episode**: A finite sequence of agent-environment interactions until the end of the episode defined by the task. An episode return is defined as: $G^{\text{ep}} = \sum_{t=0}^{T_{\text{ep}}-1} \gamma^t r_{t+1}$, where γ is a discount factor, and the random variable T_{ep} gives the number of steps until the end of the episode.
- **Lifetime**: A finite sequence of agent-environment interactions until the end of training defined by an agent-designer, which can consist of multiple episodes. The *lifetime return* is $G^{\text{life}} = \sum_{t=0}^{T-1} \gamma^t r_{t+1}$, where γ is a discount factor, and T is the number of steps in the lifetime.
- **Intrinsic reward**: A reward function $r_\eta(\tau_{t+1})$ parameterized by η, where $\tau_t = (s_0, a_0, r_1, d_1, s_1, \ldots, r_t, d_t, s_t)$ is a lifetime history with (binary) episode terminations d_i.

The Optimal Reward Problem [Singh et al., 2010], illustrated in Figure 4.1, aims to learn the parameters of the intrinsic reward such that the resulting rewards achieve a learning dynamic for an RL agent that maximises the lifetime (extrinsic) return on tasks drawn from some distribution. Formally, the objective function is defined as:

$$J(\eta) = \mathbb{E}_{\theta_0 \sim \Theta, \mathcal{T} \sim p(\mathcal{T})} \left[\mathbb{E}_{\tau \sim p_\eta(\tau|\theta_0)} \left[G^{\text{life}} \right] \right], \tag{4.1}$$

where Θ and $p(\mathcal{T})$ are an initial policy distribution and a distribution over possibly non-stationary tasks respectively. The likelihood of a lifetime history τ is

$$p_\eta(\tau|\theta_0) = p(s_0) \prod_{t=0}^{T-1} \pi_{\theta_t}(a_t|s_t) p(d_{t+1}, r_{t+1}, s_{t+1}|s_t, a_t),$$

where $\theta_t = f(\theta_{t-1}, \eta)$ is a policy parameter as updated with update function f, which is policy gradient in this chapter.[1] Note that the optimisation of η spans multiple lifetimes, each of which can span multiple episodes.

Using the lifetime return G^{life} as the objective instead of the conventional episodic return G^{ep} allows exploration across multiple episodes as long as the lifetime return is maximised in the long run. In particular, when the lifetime is defined as a fixed number of episodes, we find that the lifetime return objective is sometimes more beneficial than the episodic return objective, even for the episodic return performance measure. However, different objectives (e.g., final episode return) can be considered depending on the definition of what a good reward function is.

4.3 Meta-Learning Intrinsic Reward

We propose a meta-gradient approach [Xu et al., 2018b, Zheng et al., 2018] to solve the optimal reward problem. At a high-level, we sample a new task $\mathcal{T}$ and a new random policy parameter θ at each lifetime iteration. We then simulate an agent's lifetime by updating the parameter θ using an intrinsic reward function r_η (Section 4.3.1) with policy gradient (Section 4.3.2). Concurrently, we compute the meta-gradient by taking into account the effect of the intrinsic rewards on the policy parameters to update the intrinsic reward function with a lifetime value function (Section 4.3.3). Algorithm 3 gives an overview of our algorithm. The following sections describe the details.

4.3.1 Architectures

The intrinsic reward function is a recurrent neural network (RNN) parameterized by η, which produces a scalar reward on arriving in state s_t by taking into account the history of an agent's lifetime $\tau_t = (s_0, a_0, r_1, d_1, s_1, ..., r_t, d_t, s_t)$. We claim that giving the lifetime history across episodes as input is crucial for balancing exploration and exploitation, for instance by capturing how frequently a certain state is visited to determine an exploration bonus reward. The lifetime value function is a separate recurrent neural network parameterized by ϕ, which takes the same inputs as the intrinsic reward function and produces a scalar value estimation of the expected future return within the lifetime.

4.3.2 Policy Update

Each agent interacts with an environment and a task sampled from a distribution $\mathcal{T} \sim p(\mathcal{T})$. However, instead of directly maximising the extrinsic rewards defined by the task, the agent maximises

[1] We assume that the policy parameter is updated after each time-step throughout the chapter for brevity. However, the parameter can be updated less frequently in practice.

Algorithm 3 Learning intrinsic rewards

```
Input: $p(\mathcal{T})$: Task distribution
Input: $\Theta$: Randomly-initialised policy distribution
Initialise intrinsic reward $\eta$ and lifetime value $\phi$
repeat
  Initialise task $\mathcal{T} \sim p(\mathcal{T})$ and policy $\theta \sim \Theta$
  while lifetime not ended do
    $\theta_0 \leftarrow \theta$
    for $k = 1, 2, \dots, N$ do
      Generate a trajectory using $\pi_{\theta_{k-1}}$
      Update policy $\theta_k \leftarrow \theta_{k-1} + \alpha \nabla_{\theta_{k-1}} J_\eta(\theta_{k-1})$ using intrinsic rewards $r_\eta$ (Eq. 4.3)
    end for
    Update intrinsic reward function $\eta$ using Eq. 4.4
    Update lifetime value function $\phi$ using Eq. 4.6
    $\theta \leftarrow \theta_N$
  end while
until $\eta$ converges
```

the intrinsic rewards (r_η) by using policy gradient [Williams, 1992, Sutton et al., 2000]:

$$J_\eta(\theta) = \mathbb{E}_\theta \left[\sum_{t=0}^{T_{\text{ep}}-1} \bar{\gamma}^t r_\eta(\tau_{t+1}) \right] \tag{4.2}$$

$$\nabla_\theta J_\eta(\theta) = \mathbb{E}_\theta \left[G^{\text{ep}}_{\eta,t} \nabla_\theta \log \pi_\theta(a|s) \right], \tag{4.3}$$

where $r_\eta(\tau_{t+1})$ is the intrinsic reward at time t, and $G^{\text{ep}}_{\eta,t} = \sum_{k=t}^{T_{\text{ep}}-1} \bar{\gamma}^{k-t} r_\eta(\tau_{k+1})$ is the return of the intrinsic rewards accumulated over an episode with discount factor $\bar{\gamma}$.

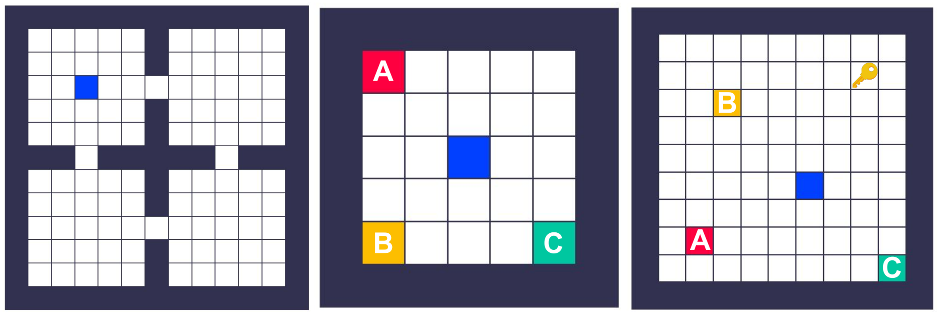
4.3.3 Intrinsic Reward and Lifetime Value Update

To update the intrinsic reward parameters η, we directly take a meta-gradient ascent step using the overall objective (Equation 4.1). Specifically, the gradient is

$$\nabla_\eta J(\eta) = \mathbb{E}_{\theta_t, \mathcal{T}} \left[G^{\text{life}}_t \nabla_{\theta_t} \log \pi_{\theta_t}(a_t|s_t) \nabla_\eta \theta_t \right], \tag{4.4}$$

The chain rule is used to get the meta-gradient ($\nabla_\eta \theta_t$) as in Chapter 3. The computation graph of this procedure is illustrated in Figure 4.1.

Computing the true meta-gradient in Equation 4.4 requires backpropagation through the entire lifetime, which is infeasible as each lifetime can involve thousands of policy updates. To partially address this issue, we truncate the meta-gradient after N policy updates but approximate the



(a) Empty Rooms (b) ABC (c) Key-Box

Figure 4.2: Illustration of domains. (a) The agent needs to find the goal location which gives a positive reward, but the goal is not visible to the agent. (b) Each object (A, B, and C) gives rewards. (c) The agent is required to first collect the key and visit one of the boxes (A, B, and C) to receive the corresponding reward. All objects are placed in random locations before each episode.

lifetime return $G_t^{\text{life},\phi} \approx G_t^{\text{life}}$ using a *lifetime value function* $V_\phi(\tau)$ parameterized by ϕ, which is learned using a temporal difference learning from n-step trajectory:

$$G_t^{\text{life},\phi} = \sum_{k=0}^{n-1} \gamma^k r_{t+k+1} + \gamma^n V_\phi(\tau_{t+n}) \tag{4.5}$$

$$\phi' = \phi + \alpha'(G_t^{\text{life},\phi} - V_\phi(\tau_t))\nabla_\phi V_\phi(\tau_t), \tag{4.6}$$

where α' is a learning rate. In our empirical work, we found that the lifetime value estimates were crucial to allow the intrinsic reward to perform long-term credit assignments across episodes (Section 4.4.6).

4.4 Empirical Investigations

We present the results from our empirical investigations in two sections. In this section, the experiments and domains are designed to answer the following research questions:

- What kind of knowledge is learned by the intrinsic reward?
- How does the distribution of tasks influence the intrinsic reward?
- What is the benefit of the lifetime return objective over the episode return?
- When is it important to provide the lifetime history as input to the intrinsic reward?

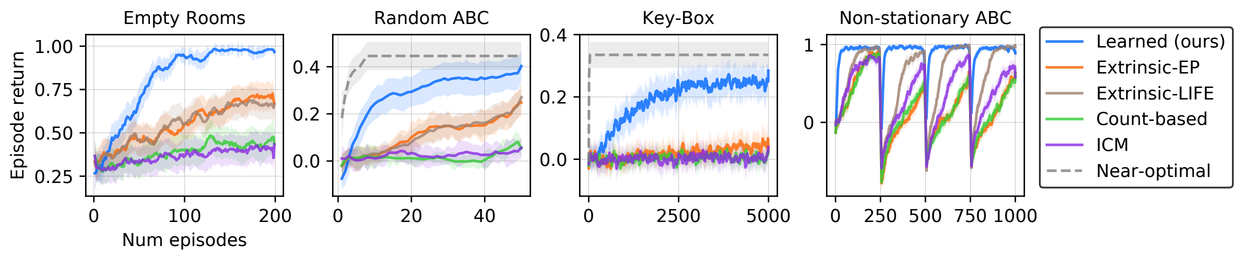


Figure 4.3: Evaluation of different reward functions averaged over 30 seeds. The learning curves show agents trained with our intrinsic reward (blue), with the extrinsic reward using the episodic return objective (orange) or the lifetime return objective (brown), and with a count-based exploration reward (green). The dashed line corresponds to a hand-designed near-optimal exploration strategy.

4.4.1 Experimental Setup

We investigate these research questions in the grid-world domains illustrated in Figure 4.2. For each domain, we trained an intrinsic reward function across many lifetimes and evaluated it by training an agent using the learned reward. We implemented the following baselines.

- Extrinsic-EP: A policy is trained with extrinsic rewards to maximise the episode return.
- Extrinsic-LIFE: A policy is trained with extrinsic rewards to maximise the lifetime return.
- Count-based [Strehl and Littman, 2008]: A policy is trained with extrinsic rewards and count-based exploration bonus rewards.
- ICM [Pathak et al., 2017]: A policy is trained with extrinsic rewards and curiosity rewards based on an inverse dynamics model.

Note that these baselines, unlike the learned intrinsic rewards, do not transfer any knowledge across different lifetimes. Throughout Sections 4.4.2-4.4.5, we focus on analysing what kind of knowledge is learned by the intrinsic reward depending on the nature of environments. We discuss the benefit of using the lifetime return and considering the lifetime history when learning the intrinsic reward in Section 4.4.6.

4.4.2 Exploring Uncertain States

We designed ‘Empty Rooms’ (Figure 4.2a) to see whether the intrinsic reward can learn to encourage exploration of uncertain states like novelty-based exploration methods. The goal is to visit an invisible goal location, which is fixed within each lifetime but varies across lifetimes. An episode terminates when the goal is reached. Each lifetime consists of 200 episodes. From the agent’s perspective, its policy should visit the locations suggested by the intrinsic reward. From the intrinsic

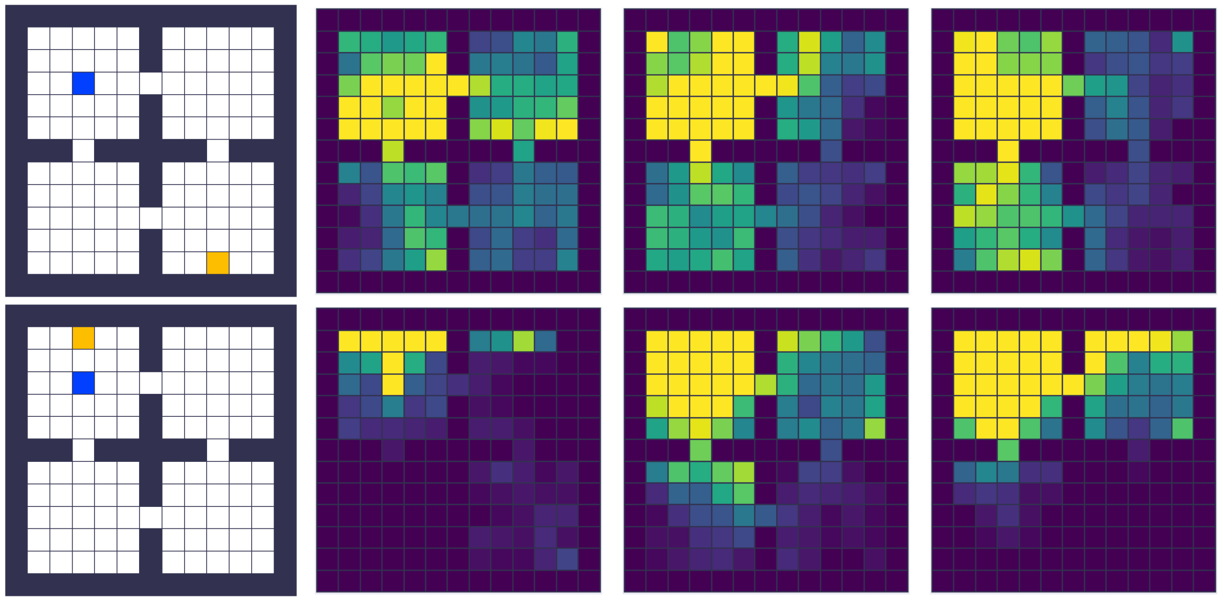


Figure 4.4: Visualisation of the first 3000 steps of an agent trained with different reward functions in Empty Rooms. (a) The blue and yellow squares represent the agent and the *hidden* goal, respectively. (b) The learned reward encourages the agent to visit many locations if the goal is not found (top). However, when the goal is found early, the intrinsic reward makes the agent exploit it without further exploration (bottom). (c-d) Both the count-based and ICM rewards tend to encourage exploration (top) but hinders exploitation when the goal is found (bottom).

reward's perspective, it should encourage the agent to go to unvisited locations to locate the goal, and then to exploit that knowledge for the rest of the lifetime.

Figure 4.3 shows that the learned intrinsic reward was more efficient than extrinsic rewards and count-based exploration when training a new agent. We observed that the intrinsic reward learned two interesting strategies as visualised in Figure 4.4. While the goal is not found, it encourages exploration of unvisited locations, because it learned the knowledge that there exists a rewarding goal location somewhere. Once the goal is found the intrinsic reward encourages the agent to exploit it without further exploration, because it learned that there is only one goal. This result shows that curiosity about uncertain states can naturally emerge when various states can be rewarding in a domain, even when the rewarding states are fixed within an agent's lifetime.

4.4.3 Exploring Uncertain Objects

In the previous domain, we considered uncertainty of where the reward (or goal location) is. We now consider dealing with uncertainty about the value of different objects. In the 'Random ABC' environment (see Figure 4.2b), for each lifetime the rewards for objects A, B, and C are uniformly

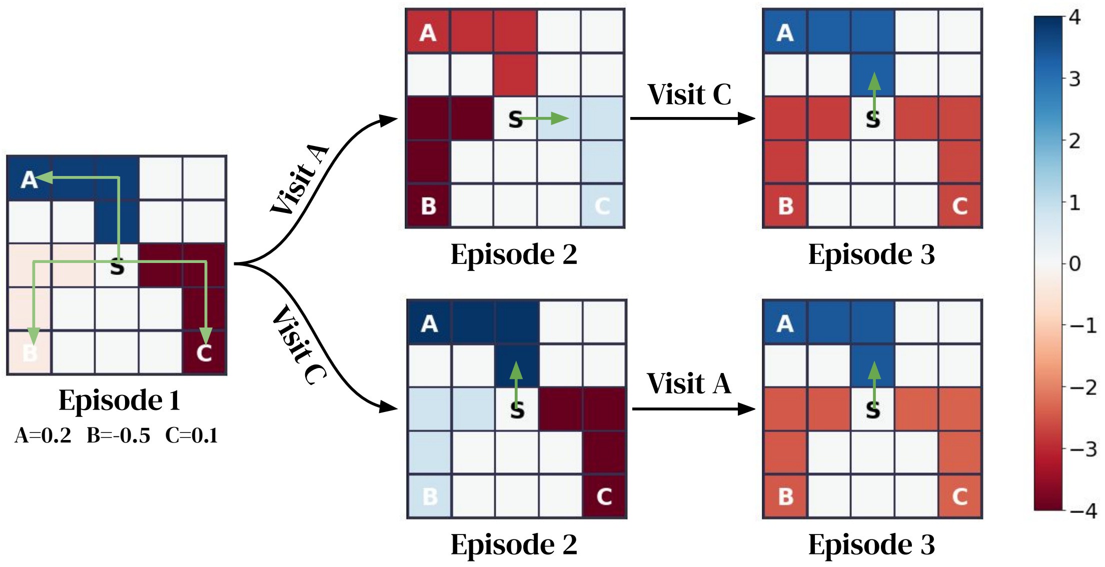


Figure 4.5: Visualisation of the learned intrinsic reward in Random ABC, where the extrinsic rewards for A, B, and C are 0.2, -0.5, and 0.1 respectively. Each figure shows the sum of intrinsic rewards for a trajectory towards each object (A, B, and C). In the first episode, the intrinsic reward encourages the agent to explore A. In the second episode, the intrinsic reward encourages exploring C if A is visited (top) or vice versa (bottom). In episode 3, after both A and C are explored, the intrinsic reward encourages revisiting A (both top and bottom).

sampled from $[-1, 1]$, $[-0.5, 0]$, and $[0, 0.5]$ respectively but are held fixed within the lifetime. A good intrinsic reward should learn that: 1) B should be avoided, 2) A and C have uncertain rewards, hence require systematic exploration (first go to one and then the other), and 3) once it is determined which of the two A or C is better, exploit that knowledge by encouraging the agent to repeatedly go to that object for the rest of the lifetime.

Figure 4.3 shows that the agent learned a near-optimal exploration-then-exploitation method with the learned intrinsic reward. Note that the agent cannot pass information about the reward for objects across episodes, as usual in reinforcement learning. The intrinsic reward can propagate such information across episodes and help the agent explore or exploit appropriately. We visualised the learned intrinsic reward for different actions sequences in Figure 4.5. The intrinsic rewards encourage the agent to explore towards A and C in the first few episodes. Once A and C are explored, the agent exploits the largest rewarding object. Throughout training, the agent is discouraged to visit B through negative intrinsic rewards. These results show that avoidance and curiosity about uncertain objects can potentially emerge if the environment has various or fixed rewarding objects.

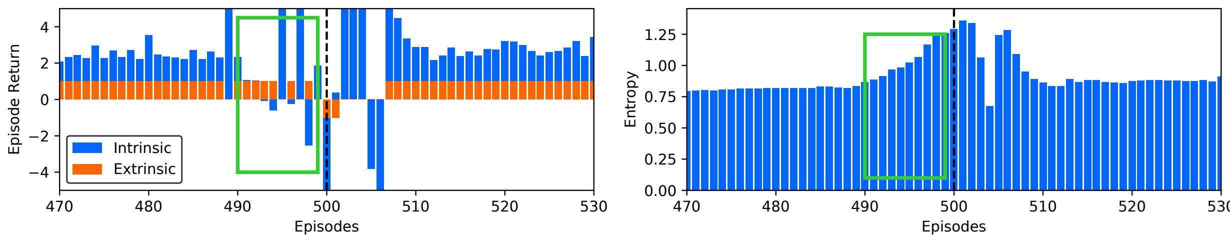


Figure 4.6: Visualisation of the agent's intrinsic and extrinsic rewards (left) and the entropy of its policy (right) on Non-stationary ABC. The task changes at 500th episode (dashed vertical line). The intrinsic reward gives a negative reward even before the task changes (green rectangle) and makes the policy less deterministic (entropy increases). As a result, the agent quickly adapts to the change.

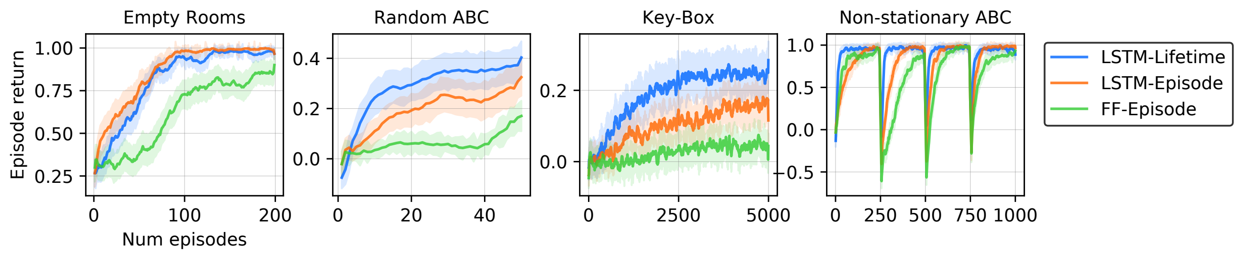


Figure 4.7: Evaluation of different intrinsic reward architectures and objectives. For 'LSTM' the reward network has an LSTM taking the lifetime history as input. For 'FF' a feed-forward reward network takes only the current time-step. 'Lifetime' and 'Episode' means the lifetime and episodic return as objective respectively.

4.4.4 Exploiting Invariant Causal Relationship

To see how the intrinsic reward deals with causal relationship between objects, we designed 'Key-Box', which is similar to Random ABC except that there is a key in the room (see Figure 4.2c). The agent needs to collect the key first to open one of the boxes (A, B, and C) and receive the corresponding reward. The rewards for the objects are sampled from the same distribution as Random ABC. The key itself gives a neutral reward of 0. Moreover, the locations of the agent, the key, and the boxes are randomly sampled for each episode. As a result, the state space contains more than 3 billion distinct states and thus is infeasible to fully enumerate. Figure 4.3 shows that learned intrinsic reward leads to a near-optimal exploration. The agent trained with extrinsic rewards did not learn to open any box. The intrinsic reward captures that the key is necessary to open any box, which is true across many lifetimes of training. This demonstrates that the intrinsic reward can capture causal relationships between objects when the domain has this kind of invariant dynamics.

4.4.5 Dealing with Non-stationarity

We investigated how the intrinsic reward handles non-stationary tasks within a lifetime in our 'Non-stationary ABC' environment. Rewards are as follows: for A is either 1 or -1, for B is -0.5, for C is the negative value of the reward for A. The rewards of A and C are swapped every 250 episodes. Each lifetime lasts 1000 episodes. Figure 4.3 shows that the agent with the learned intrinsic reward quickly recovered its performance when the task changes, whereas the baselines take more time to recover. Figure 4.6 shows how the learned intrinsic reward encourages the learning agent to react to the changing rewards. Interestingly, the intrinsic reward has learned to prepare for the change by giving negative rewards to the exploitation policy of the agent a few episodes before the task changes. In other words, the intrinsic reward reduces the agent's commitment to the current best rewarding object, thereby increasing entropy in the current policy in anticipation of the change, eventually making it easier to adapt quickly. This shows that the intrinsic reward can capture the (regularly) repeated non-stationarity across many lifetimes and make the agent intrinsically motivated not to commit too firmly to a policy, in anticipation of changes in the environment.

4.4.6 Ablation Study

To study relative benefits of the proposed technical ideas, we conducted an ablation study 1) by replacing the long-term lifetime return objective (G^{life}) with the episodic return (G^{ep}) and 2) by restricting the input of the reward network to the current time-step instead of the entire lifetime history. Figure 4.7 shows that the lifetime history was crucial to achieve good performance. This is reasonable because all domains require some past information (e.g., object rewards in Random ABC, visited locations in Empty Rooms) to provide useful exploration strategies. It is also shown that the lifetime return objective was beneficial on Random ABC, Non-stationary ABC, and Key-Box. These domains require exploration across multiple episodes in order to find the optimal policy. For example, collecting an uncertain object (e.g., object A in Random ABC) is necessary even if the episode terminates with a negative reward. The episodic value function would directly penalise such an under-performing exploratory episode when computing meta-gradient, which prevents the intrinsic reward from learning to encourage exploration across episodes. On the other hand, such behaviour can be encouraged by the lifetime value function, as long as it provides useful information to maximise the lifetime return in the long term.

4.5 Generalisation via Rewards

As noted above, rewards capture knowledge about what an agent's goals should be rather than how it should behave. At the same time, transferring the latter in the form of policies is also feasible in

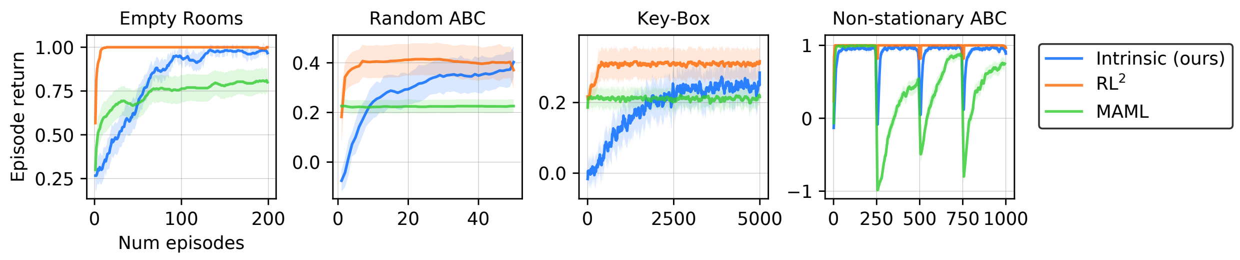


Figure 4.8: Comparison to policy transfer methods.

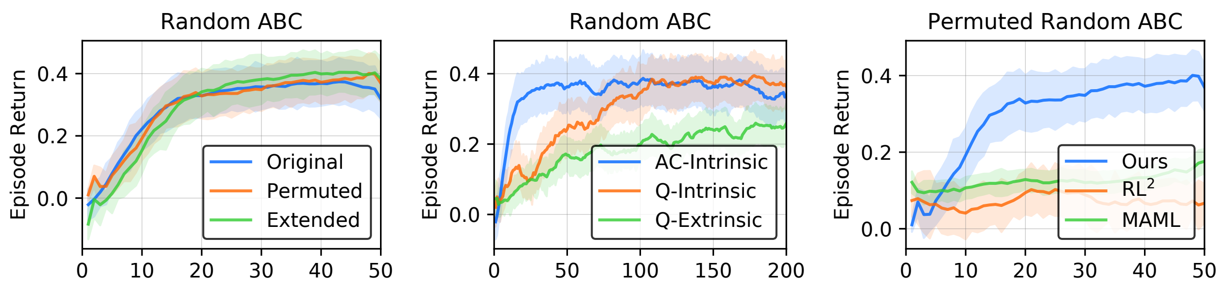
our domains presented above. Here we confirm it by implementing and presenting results for the following two meta-learning methods:

- MAML [Finn et al., 2017a]: A policy meta-learned from a distributions of tasks such that it can adapt quickly to the given task after a few parameter updates.
- RL^2 [Duan et al., 2016b, Wang et al., 2016b]: An RNN policy unrolled over the entire lifetime to maximise the lifetime return, which is pre-trained on a distributions of tasks.

Although all the methods we implemented including ours are designed to learn useful knowledge from a distribution of tasks, they have different objectives. Specifically, the objective of our method is to learn knowledge that is useful for training "randomly-initialised policies" by capturing "what to do", whereas the goal of policy transfer methods is to directly transfer a useful policy for fast task adaptation by transferring "how to do" knowledge. In fact, it can be more efficient to transfer and reuse pre-trained policies instead of restarting from a random policy and learning using the learned rewards given a new task. Figure 4.8 indeed shows that RL^2 performs better than our intrinsic reward approach. It is also shown that MAML and RL^2 achieve good performance from the beginning, as they have already learned how to navigate the grid worlds and how to achieve the goals of the tasks. In our method, on the other hand, the agent starts from a random policy and relies on the learned intrinsic reward which indirectly tells it what to do. Nevertheless, our method outperforms MAML and achieves a comparable asymptotic performance to RL^2.

4.5.1 Generalise to New Agent-Environment Interfaces

In fact, our method can be interpreted as an instance of RL^2 with a particular decomposition of parameters (θ and η), which uses policy gradient as a recurrent update (see Figure 4.1). While this modular structure may not be more beneficial than RL^2 when evaluated with the same agent-environment interface, such a decomposition provides clear semantics of each module: the policy (θ) captures "how to do" while the intrinsic reward (η) captures "what to do", and this enables in-



(a) Action space (b) Algorithm (c) Comparison to baselines

Figure 4.9: Generalisation to new agent-environment interfaces in Random ABC. (a) 'Permuted' agents have different action semantics. 'Extended' agents have additional actions. (b) 'AC-Intrinsic' is the original actor-critic agent trained with the intrinsic reward. 'Q-Intrinsic' is a Q-learning agent with the intrinsic reward learned from actor-critic agents. 'Q-Extrinsic' is the Q-learning agent with the extrinsic reward. (c) The performance of the policy transfer baselines with permuted actions during evaluation.

teresting kinds of generalisations as we show below. Specifically, we show that "what" knowledge captured by the intrinsic reward can be reused by many different learning agents as follows.

Generalise to Unseen Action Spaces We first evaluated the learned intrinsic reward on new action spaces. Specifically, the intrinsic reward was used to train new agents with either 1) permuted actions, where the semantics of left/right and up/down are reversed, or 2) extended actions, with 4 additional actions that move diagonally. Figure 4.9a shows that the intrinsic reward provided useful rewards to new agents with different actions, though it was not trained with those actions. This is feasible because the intrinsic reward assigns rewards to the agent's state changes rather than its actions. The intrinsic reward captures "what to do", which makes it feasible to generalise to new actions, as long as the goal remains the same. On the other hand, it is unclear how to generalise RL^2 and MAML in this way.

Generalise to Unseen Learning Algorithms We further investigated how general the learned intrinsic reward is by evaluating it on agents with different learning algorithms. Specifically, after training the intrinsic reward from actor-critic agents, we evaluated it by training new agents through Q-learning while using the learned intrinsic reward as denoted by 'Q-Intrinsic' in Figure 4.9b. Interestingly, it turns out that the learned intrinsic reward is general enough to be useful for Q-learning agents, even though it was trained for actor-critic agents. Again, it is unclear how to generalise RL^2 and MAML in this way.

Comparison to Policy Transfer Although it is impossible to apply the learned policy from RL^2 and MAML when we extend the action space or when we change the learning algorithm, we can do so when we only permute the actions. As shown in Figure 4.9c, both RL^2 and MAML generalise poorly when the action space is permuted for Random ABC, because the transferred policies are highly biased to the original action space. Again, this result highlights the difference between "what to do" knowledge captured by our approach and "how to do" knowledge captured by policies.

4.6 Conclusion

In this chapter, we revisited the original optimal reward problem [Singh et al., 2009] and proposed a more scalable gradient-based method for learning intrinsic rewards across lifetimes. Through several proof-of-concept experiments, we showed that the learned non-stationary intrinsic reward can capture regularities within a distribution of environments or, over time, within a non-stationary environment. As a result, they were capable of encouraging both exploratory and exploitative behaviour across multiple episodes. In addition, some task-independent notions of intrinsic motivation such as curiosity emerged when they were effective for the distribution over tasks across lifetimes the agent was trained on. We also showed that the learned intrinsic rewards can generalise to different agent-environment interfaces such as different action spaces and different learning algorithms, whereas policy transfer methods fail to generalise to such changes. This highlights the difference between the "what" kind of knowledge captured by rewards and the "how" kind of knowledge captured by policies. The flexibility and range of knowledge captured by intrinsic rewards in our proof-of-concept experiments encourages further work towards combining different loci of knowledge to achieve greater practical benefits.

CHAPTER 5

Adaptive Pairwise Weights for Temporal Credit Assignment

The following *umbrella problem* [Osband et al., 2019] illustrates a fundamental challenge in most reinforcement learning (RL) problems, namely the *temporal credit assignment* (TCA) problem. An RL agent takes an umbrella at the start of a cloudy morning and experiences a long day at work filled with various rewards uninfluenced by the umbrella, before needing the umbrella in the rain on the way home. The agent must learn to credit the take-umbrella action in the cloudy-morning state with the very delayed reward at the end of the day, while also learning to not credit the action with the many intervening rewards, despite their occurring much closer in time. More generally, the TCA problem is how much credit or blame should an action taken in a state get for a future reward. One of the earliest and still most widely used heuristics for TCA comes from the celebrated TD(λ) [Sutton, 1988] family of algorithms, and assigns credit based on a scalar coefficient λ raised to the power of the time interval between the state-action and the reward. Note that this is a recency and frequency heuristic, in that it assigns credit based on how recently and how frequently a state-action pair has occurred prior to the reward. It is important, however, to also note that this heuristic has not in any way shown to be the "optimal" way for TCA. In particular, in the umbrella problem the action of taking the umbrella on a cloudy morning will be assigned credit for the rewards achieved during the workday early on in learning and it is only after a lot of learning that this effect will diminish. Nevertheless, the recency and frequency heuristic has been adopted in most modern RL algorithms because it is so simple to implement, with just one hyperparameter, and because it has been shown to allow for asymptotic convergence to the true value function under certain circumstances.

In this chapter, we present two new families of algorithms for addressing TCA: one that generalises TD(λ) and a second that generalises a Monte-Carlo algorithm. Specifically, our generalisation introduces pairwise weightings that are functions of *the state in which the action was taken*, *the state at the time of the reward*, and *the time interval between the two*. Of course, it isn't clear what this pairwise weight function should be, and it is too complex to be treated as a hyperparameter (in contrast to the scalar λ in TD(λ)). We develop a metagradient approach to learning the pairwise weight function at the same time as learning the policy of the agent. Like other metagradient algorithms, our algorithm has two loops: an outer loop that periodically updates the pairwise weight function in order to optimize the usual RL loss (policy gradient loss in our case) and an inner loop where the policy parameters are updated using the pairwise weight function set by the outer loop.

Our main contribution in this chapter is a family of algorithms that contains within it the theoretically well understood TD(λ) and Monte-Carlo algorithms. We show that the additional flexibility of our algorithms can yield benefit analytically in a simple illustrative example intended to build intuition and then empirically in more challenging TCA problems. A second contribution is the metagradient algorithm to learn such the pairwise-weighting function that parameterises our family of algorithms. Our empirical work is geared towards answering two questions: (1) Are the proposed pairwise weight functions able to outperform the best choice of λ and other baselines? (2) Is our metagradient algorithm able to learn the pairwise weight functions fast enough to be worth the extra complexity they introduce?

5.1 Related Work

Several heuristic methods have been proposed to address the long-term credit assignment problem in RL. RUDDER [Arjona-Medina et al., 2019] trains a LSTM [Hochreiter and Schmidhuber, 1997] to predict the return of an episode given the entire state and action sequence and then conducts contribution analysis with the LSTM to redistribute rewards to state-action pairs. Synthetic Returns (SR) [Raposo et al., 2021] directly learns the association between past events and future rewards and use it as a proxy for credit assignment. Different from the predictive approach of RUDDER and SR, Temporal Value Transport (TVT) [Hung et al., 2019] augments the agent with an external memory module and utilizes the memory retrieval as a proxy for transporting future value back to related state-action pairs. We compare against TVT by using their published code, and we take inspiration from the core reward-redistribution idea from RUDDER and implement it within our policy gradient agent as a comparison baseline (because the available RUDDER code is not directly applicable). We do not compare to SR because their source code is not available.

We also compare against two other algorithms that are more closely related to ours in their use of metagradients. Xu et al. [Xu et al., 2018b] adapt λ via metagradients rather than tuning it via hyperparameter search, thereby improving over the use of a fixed-λ algorithm. The Return Generating Model (RGM) [Wang et al., 2019] generalizes the notion of return from exponentially discounted sum of rewards to a more flexibly weighted sum of rewards where the weights are adapted via metagradients during policy learning. RGM takes the entire episode as input and generates one weight for each time step. In contrast, we study pairwise weights as explained below.

Some recent works address counterfactual credit assignment where classic RL algorithms struggle [Harutyunyan et al., 2019, Mesnard et al., 2020, van Hasselt et al., 2020]. Although they are related to our work in that they also address the TCA problem, we do no compare to them because our work does not focus on the counterfactual aspect.

5.2 Pairwise Weights for Advantages

At the core of our contribution are new parameterizations of functions for computing advantage estimators used in policy gradient methods. Therefore, we briefly review advantages estimators in policy gradient methods as our points of departure. For a more comprehensive review, please refer to Chapter 2.

Advantage estimators Recall from Chapter 2 that the Monte-Carlo estimator of the advantage function $\Psi(S_t, A_t)$ is

$$\hat{\Psi}_t^{\text{MC}} = G_t - \hat{v}(S_t), \tag{5.1}$$

and the λ-estimator is

$$\begin{aligned}\hat{\Psi}_t^{(\lambda)} &= G_t^{(\lambda)} - \hat{v}(S_t) \\ &= \sum_{k=t+1}^{T} (\gamma\lambda)^{k-t-1}\delta_k,\end{aligned} \tag{5.2}$$

where $\delta_t = R_t + \gamma\hat{v}(S_t) - \hat{v}(S_{t-1})$ is the TD-error at time t. As a special case, when $\lambda = 1$, it recovers the MC estimator [Schulman et al., 2015]. As noted above, the value for λ is usually manually tuned as a hyperparameter. Adjusting λ provides a way to tradeoff bias and variance in $\hat{\Psi}^{(\lambda)}$ (this is absent in $\hat{\Psi}^{\text{MC}}$). Below we present two new estimators that are analogous in this regard to $\hat{\Psi}^{(\lambda)}$ and $\hat{\Psi}^{\text{MC}}$.

Proposed Heuristic 1: Advantages via Pairwise Weighted Sum of TD-errors. Our first new estimator, denoted PWTD for **P**airwise **W**eighted **TD**-error, is a strict generalization of the λ-estimator above and is defined as follows:

$$\hat{\Psi}_{\eta,t}^{\text{PWTD}} = \sum_{k=t+1}^{T} f_\eta(S_t, S_k, k-t)\delta_k, \tag{5.3}$$

where $f_\eta(S_t, S_k, k-t) \in [0,1]$, parameterized by η, is the scalar weight given to the TD-error δ_k as a function of the state to which credit is being assigned, the state at which the TD-error is obtained, and the time interval between the two. Note that if we choose $f(S_t, S_k, k-t) = (\gamma\lambda)^{k-t-1}$, it recovers the usual λ-estimator $\hat{\Psi}^{(\lambda)}$.

Proposed Heuristic 2: Advantages via Pairwise Weighted Sum of Rewards. Instead of generalizing from the λ-estimator, we can also generalize from the MC estimator via pairwise weighting. Specifically, the new pairwise-weighted return is defined as

$$G^{\text{PWR}}_{\eta,t} = \sum_{k=t+1}^{T} f_\eta(S_t, S_k, k-t) R_k, \tag{5.4}$$

where $f_\eta(S_t, S_k, k-t) \in [0,1]$ is the scalar weight given to the reward R_k. The corresponding advantage estimator, denoted PWR for **P**airwise **W**eighted **R**eward, then is:

$$\hat{\Psi}^{\text{PWR}}_{\eta,t} = G^{\text{PWR}}_{\eta,t} - \hat{v}^{\text{PWR}}(S_t), \tag{5.5}$$

where $v^{\text{PWR}}(s) = \mathbb{E}_\theta[G^{\text{PWR}}_{\eta,t} | S_t = s]$ and $\hat{v}^{\text{PWR}}$ is an approximation of v^{PWR}. Note that if we choose $f(S_t, S_k, k-t) = \gamma^{k-t-1}$, we can recover the MC estimator $\hat{\Psi}^{\text{MC}}$.

The benefit of the additional flexibility provided by these new estimators highly depends on the choice of the pairwise weight function f. As we will demonstrate in the simple example below, the new estimators can yield lower variance and benefit policy learning if the function f captures the underlying credit assignment structure of the problem. On the other hand, the new estimators may not even be well-defined in the infinite-horizon setting if the pairwise weight function is chosen wrongly because the weighted sum of TD-errors/rewards could be unbounded. Designing a good pairwise weight function by hand is challenging because it requires both domain knowledge to capture the credit assignment structure and careful tuning to avoid harmful consequences. Thus we propose a metagradient algorithm to *learn* the pairwise weight function such that it benefits policy learning, as detailed in § 5.3.

An Illustrative Analysis of the Benefit of the PWR Estimator. Consider the simple-MDP version of the umbrella problem in Figure 5.1. Each episode starts at the leftmost state, s_0, and consists of T transitions. The only choice of action is at s_0 and it determines the reward on the last transition. A noisy reward ϵ is sampled for each intermediate transition independently from a distribution with mean μ and variance $\sigma^2 > 0$. These intermediate rewards are independent of the initial action. We consider the undiscounted setting in this example. The expected return for state

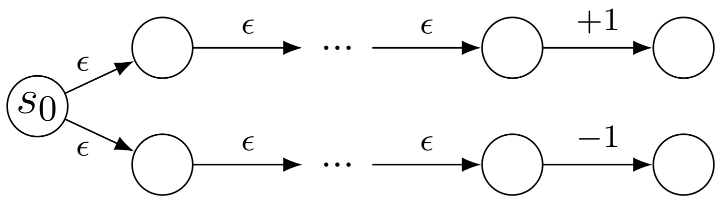


Figure 5.1: A simple illustrative MDP. The initial action determines the final reward but does not impact the intermediate rewards. The consequence of the initial action is delayed.

s_0 under policy π is

$$v(s_0) = \mathbb{E}_\pi[G_0] = (T-1)\mu + \mathbb{E}_\pi[R_T].$$

For any initial action a_0, the advantage is

$$\Psi(s_0, a_0) = \mathbb{E}_\pi[G_0|a_0] - v(s_0) = \mathbb{E}_\pi[R_T|a_0] - \mathbb{E}_\pi[R_T].$$

Consider pairwise weights for computing $\hat{\Psi}^{\text{PWR}}(s_0, a_0)$ that place weight only on the final transition, and zero weight on the noisy intermediate rewards, capturing the notion that the intermediate rewards are not influenced by the initial action choice. More specifically, we choose f such that for any episode, $w_{0T} = 1$ and $w_{ij} = 0$ for other i and j. The shorthand w_{ij} denotes $f(S_i, S_j, j-i)$ for brevity. The expected parameterized reward sum for the initial state s_0 is

$$v^{\text{PWR}}(s_0) = \mathbb{E}_\pi[G_{\eta,0}] = \mathbb{E}_\pi[\sum_{i=t}^{T} w_{0t} R_t] = \mathbb{E}_\pi[R_T].$$

If v^{PWR} is correct, for any initial action a_0, the pairwise-weighted advantage is the same as the regular advantage:

$$\begin{aligned}\mathbb{E}_\pi[\hat{\Psi}^{\text{PWR}}_\eta(s_0, a_0)] &= \mathbb{E}_\pi[G_{\eta,0} - \hat{v}^{\text{PWR}}(s_0)|a_0] \\ &= \mathbb{E}_\pi[\sum_{t=1}^{T} w_{0t} R_t] - v^{\text{PWR}}(s_0) \\ &= \mathbb{E}[R_T|a_0] - \mathbb{E}_\pi[R_T] = \Psi(s_0, a_0).\end{aligned}$$

As for variance, for any initial action a_0, $[G_{\eta,0}|a_0]$ is deterministic because of the zero weight on all the intermediate rewards and thus $\hat{\Psi}^{\text{PWR}}_\eta(s_0, a_0)$ has zero variance. The variance of $\hat{\Psi}^{\text{MC}}(s_0, a_0)$ on the other hand is $(T-1)\sigma^2 > 0$. Thus, in this illustrative example $\hat{\Psi}^{\text{PWR}}$ yields an unbiased advantage estimator with far lower variance than $\hat{\Psi}^{\text{MC}}$.

Our example exploited knowledge of the domain to set weights that would yield an unbiased advantage estimator with reduced variance, thereby providing some intuition on how a more flexible return might in principle yield benefits for learning. Of course, in general RL problems will have the umbrella problem in them to varying degrees. But how can these weights be set by the agent itself, without prior knowledge of the domain? We turn to this question next.

5.3 A Metagradient Algorithm for Adapting Pairwise Weights

Recently metagradient methods have been developed to learn various kinds of parameters that would otherwise be set by hand or by manual hyperparameter search; these include discount factors [Xu et al., 2018b, Zahavy et al., 2020], intrinsic rewards (Chapter 3 and 4; also see [Zheng et al., 2018, Rajendran et al., 2019, Zheng et al., 2020]), auxiliary tasks [Veeriah et al., 2019], constructing general return functions [Wang et al., 2019], and discovering new RL objectives [Oh et al., 2020, Xu et al., 2020]. We use the metagradient algorithm from Xu et al. [2018b] for training the pairwise weights. The algorithm consists of an outer loop learner for the pairwise weight function, which is driven by a conventional policy gradient loss and an inner loop learner driven by a policy-gradient loss based on the new pairwise-weighted advantages. For brevity, we use $\hat{\Psi}_\eta$ to denote $\hat{\Psi}_\eta^{\text{PWTD}}$ or $\hat{\Psi}_\eta^{\text{PWR}}$ unless it causes ambiguity.

Learning in the Inner Loop. In the inner loop, the pairwise-weighted advantage $\hat{\Psi}_\eta$ is used to compute the policy gradient. We rewrite the gradient update from Eq. 2.18 with the new advantage as

$$\nabla_\theta J_\eta(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}[\sum_{t=0}^{T-1} \hat{\Psi}_{\eta,t} \nabla_\theta \log \pi_\theta(A_t|S_t)], \tag{5.6}$$

where τ is a trajectory sampled by executing π_θ. The overall update to θ is

$$\nabla_\theta J^{\text{inner}}(\theta) = \nabla_\theta J_\eta(\theta) + \beta^{\mathcal{H}} \nabla_\theta \mathcal{H}(\pi_\theta), \tag{5.7}$$

where $\mathcal{H}(\theta)$ is the usual entropy regularization term [Mnih et al., 2016] and $\beta^{\mathcal{H}}$ is a mixing coefficient. We apply gradient ascent to update the policy parameters and the updated parameters are denoted by θ'.

Computing $\hat{\Psi}_\eta^{\text{PWR}}$ with Equation 5.5 requires a value function predicting the expectation of pairwise-weighted sums of rewards. We train the value function, $\hat{v}_\psi$ with parameters ψ, along with the policy by minimizing the mean squared error between its output $\hat{v}_\psi(S_t)$ and the pairwise-weighted sum of rewards $G_{\eta,t}$. The objective for training $\hat{v}_\psi$ is

$$J^v_\eta(\psi) = \mathbb{E}_{\tau\sim\pi_\theta}[\sum_{t=0}^{T-1}(G_{\eta,t} - \hat{v}_\psi(S_t))^2]. \tag{5.8}$$

Note that $\hat{\Psi}^{\text{PWTD}}_\eta$ does not need this extra value function.

Updating η via Metagradient in the Outer Loop. To update η, the parameters of the pairwise weight functions, we need to compute the gradient of the usual policy loss w.r.t. η through the effect of η on the inner loop's updates to θ.

$$\nabla_\eta J^{\text{outer}}(\eta) = \nabla_{\theta'} J(\theta')\nabla_\eta\theta'. \tag{5.9}$$

where,

$$\nabla_{\theta'} J(\theta') = \mathbb{E}_{\tau'\sim\pi_{\theta'}}[\sum_{i=0}^{T-1}\Psi_t\nabla_{\theta'}\log\pi_{\theta'}(A_t|S_t)], \tag{5.10}$$

τ' is another trajectory sampled by executing the updated policy $\pi_{\theta'}$ and Ψ_t is the regular advantage.

Note that we need two trajectories, τ and τ', to make one update to the meta-parameters η. The policy parameters θ are updated after collecting trajectory τ. The next trajectory τ' is collected using the updated parameters θ'. The η-parameters are updated on τ'. In order to make more efficient use of the data, we follow [Xu et al., 2018b] and reuse the second trajectory τ' in the next iteration as the trajectory for updating θ. In practice we use modern auto-differentiation tools to compute Equation 5.9 without applying the chain rule explicitly. Computing the regular advantage requires a value function for the regular return. This value function is parameterized by ϕ and updated to minimize the squared loss analogously to $\hat{v}_\phi$.

5.4 Experiments

We present three sets of experiments. The first set (§5.4.1) uses simple tabular MDPs that allow visualization of the pairwise weights learned by Meta-PWTD and -PWR. The results show that the metagradient adaptation both *increases* and *decreases* weights in a way that can be interpreted as reflecting explicit credit assignment and variance reduction. In the second set (§5.4.2) we test Meta-PWTD and -PWR with neural networks in the benchmark credit assignment task *Key-to-Door* [Hung et al., 2019]. We show that Meta-PWTD and -PWR outperform several existing methods for directly addressing credit assignment, as well as TD(λ) methods, and show again that the learned weights reflect domain structure in a sensible way. In the third set (§5.4.3), we evaluate

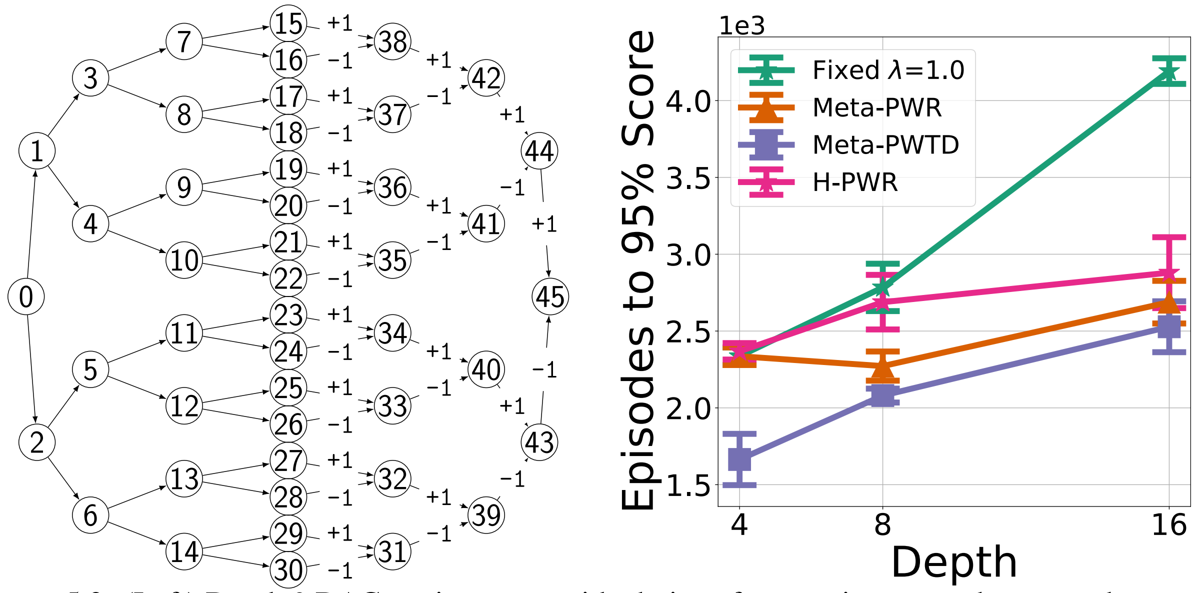


Figure 5.2: (Left) Depth 8 DAG environment with choice of two actions at each state and rewards along transitions. (Right) Learning performance of regular return, handcrafted weights, and fixed meta-learned weights. Results are averaged over 5 independent runs. Low is good.

Meta-PWTD and -PWR in two benchmark RL domains, `bsuite` [Osband et al., 2019] and Atari, and show that our methods do not hinder learning when environments do not pose idealized long-term credit assignment challenges.

5.4.1 Learned Pairwise Weights in A Simple MDP

Consider the environment represented as a DAG in Figure 5.2 (left). In each state in the left part of the DAG (states 0–14, the *first phase*), the agent chooses one of two actions but receives no reward. In the remaining states (states 15–44, the *second phase*) the agent has only one action available and it receives a reward of $+1$ or -1 at each transition. Crucially, the rewards the agent obtains in the second phase are a consequence of the action choices in the first phase because they determine which states are encountered in the second phase. There is an interesting credit assignment problem with a nested structure; for example, the action chosen at state 1 determines the reward received later upon transition into state 44. We refer to this environment as the *Depth* 8 *DAG* and also report results below for depths 4 and 16.

In the DAG environments we use a tabular policy, value function, and meta-parameter representations. The parameters θ, ψ, ϕ, and η represent the policy, baseline for the weighted return, baseline for the regular return, and meta-parameters respectively. The η parameters are a $|S| \times |S|$ matrix. The entry on the ith row and the jth column defines the pairwise weight for computing the contribution of reward at state j to the return at state i. A sigmoid is used to bound the weights to $[0, 1]$, and the η parameters are initialized so that the pairwise weights are close to 0.5. Note that even in a tabular domain such as the DAG, setting the credit assignment weights by random search would be infeasible due to the number of possible weight combinations. This problem is exacerbated by larger domains discussed in the following sections. For this reason, the metagradient algorithm is a promising candidate for setting the weights.

Visualizing the Learned Weights via Inner-loop Reset. To see the most effective weights that metagradient learned for a random policy, we repeatedly reset the policy parameters to a random initialization while continuing to train the meta-parameters until convergence. More specifically: the meta-parameters η are trained repeatedly by randomly initializing θ, ψ, and ϕ and running the inner loop for 16 updates for each outer loop update. Following existing work in metagradient [Veeriah et al., 2019, Zheng et al., 2020], the outer loop objective is evaluated on all 16 trajectories sampled with the updated policies. The gradient of the outer loop objective on the ith trajectory with respect to η is backpropagated through all of the preceding updates to θ.

What pairwise weights would accelerate learning in this domain? Figure 5.2 (top) visualizes a set of *handcrafted* weights for $\hat{\Psi}^{\text{PWR}}$ in the Depth 8 DAG; each row in the grid represents the state in which an action advantage is estimated, and each column the state in which a future reward is experienced. For each state pair (s_i, s_j) the weight is 1 (yellow) only if the reward at s_j depends on the action choice at s_i, else it is zero (dark purple; the white pairs are unreachable). Figure 5.2 (middle) shows the corresponding weights learned by Meta-PWR. Importantly, the learned pairwise weights have been *increased* for those state pairs in which the handcrafted weights are 1 and have been *decreased* for those state pairs in which the handcrafted weights are 0. As in the analysis of the simple domain in §5.2, these weights will result in lower variance advantage estimates.

The same reset-training procedure was applied to Ψ^{PWTD}. Figure 5.2 (bottom) visualizes the resulting weights. Since the TD-errors depend on the value function which are nonstationary during agent learning, we expect different weights to emerge at different points in training; the presented weights are but one snapshot. But a clear contrast to reward weighting can be seen: high weights are placed on transitions in the first phase of the DAG, which yield no rewards—because the TD-errors at these transitions do provide signal once the value function begins to be learned.

Evaluation of the Learned Pairwise Weights. After the θ-reset training of the pairwise-weights completed, we used them to train a new set of θ parameters, fixing the pairwise weights during learning. Figure 5.2 (right) shows the number of episodes to reach 95% of the maximum score in each DAG, for policies trained with regular returns, handcrafted weights (H-PWR), and meta-learned weights. Using the meta-learned weights learned as fast as (indeed faster than) using the handcrafted weights, and both were faster than the regular returns, with the gap increasing for larger DAG-depth. We conjecture that the learned weights performed even better than the

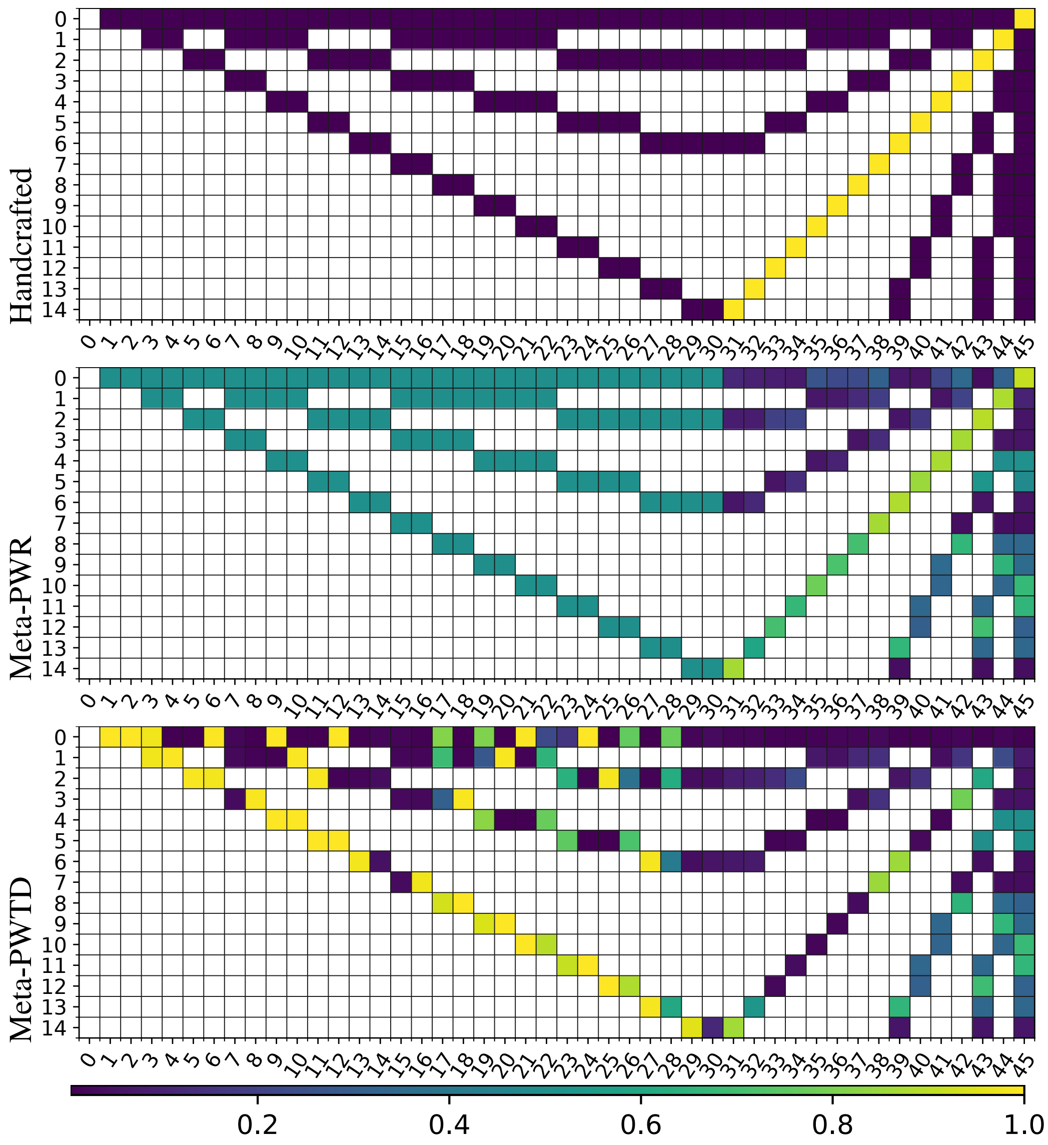


Figure 5.3: Inner loop-reset weight visualization: Top: Handcrafted pairwise weights for Depth 8 DAG; rows and columns correspond to states in Fig. 5.2. Middle: Meta-learned weights for Depth 8 DAG for Meta-PWR and Bottom: Meta-PWTD.

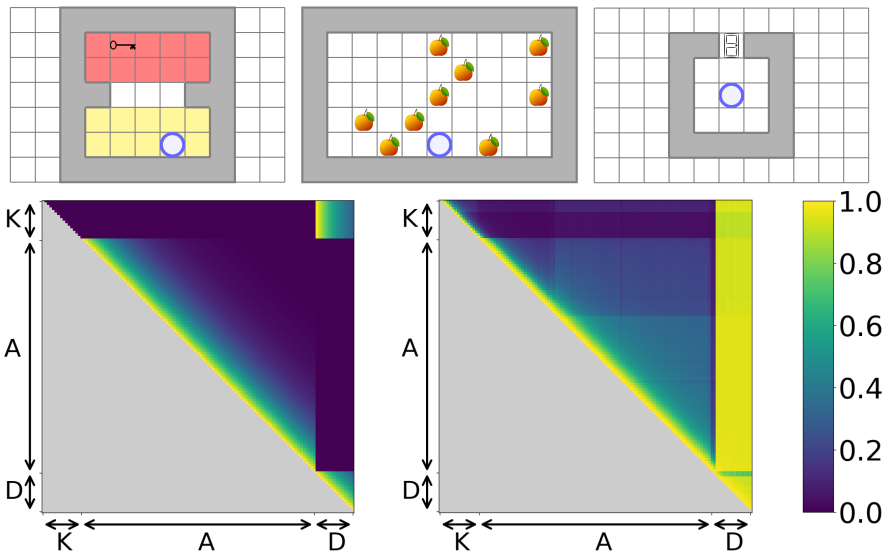


Figure 5.4: (Top) The three phases in KtD. The blue circle denotes the agent. (Bottom left) Visualization of Handcrafted weights in the KtD experiment. (Bottom right) Weights learned by Meta-PWR in the $\mu = 5, \sigma = 5$ configuration.

handcrafted weights because the learned weights adapted to the dynamics of the inner-loop policy learning procedure whereas the handcrafted weights did not.

5.4.2 The Key-to-Door Experiments

We evaluated Meta-PWTD and -PWR the Key-to-Door (KtD) environment [Hung et al., 2019] that is an elaborate umbrella problem that was designed to show-off the TVT algorithm's ability to solve TCA. We varied properties of the domain to vary the credit assignment challenge. We compared the learning performance of our algorithms to a version of $\hat{\Psi}^{\text{PWR}}$ that uses fixed handcrafted pairwise weights and no metagradient adaptation, as well as to the following **five** baselines (see related work in §5.1): **(a)** best fixed-λ: Actor-Critic (A2C) [Mnih et al., 2016] with a best fixed λ found via hyperparameter search; **(b)** TVT [Hung et al., 2019] (using the code accompanying the paper); **(c)** A2C-RR: a reward redistribution method *inspired* by RUDDER [Arjona-Medina et al., 2019]; **(d)** Meta-$\lambda(s)$ [Xu et al., 2018b]: meta-learning a state-dependent function $\lambda(s)$ for λ-returns; and **(e)** RGM [Wang et al., 2019]: meta-learning a *single* set of weights for generating returns as a linear combination of rewards.

Environment and Parametric Variation. KtD is a fixed-horizon task where each episode consists of three phases (Figure 5.4 top). In the *Key phase* (15 steps in duration) there is no reward and the agent must navigate to the key to collect it. In the *Apple phase* (90 steps in duration) the agent collects apples; apples disappear once collected. Each apple yields a noisy reward with mean μ and variance σ^2. In the *Door phase* (15 steps in duration) the agent starts at the center of a room with a door but can open the door only if it has collected the key earlier. Opening the door yields a reward of 10. Crucially, picking up the key or not has no bearing on the ability to collect apple rewards. The apples are the noisy rewards that *distract the agent from learning that picking up the key early on leads to a door-reward later*. In our experiments, we evaluate methods on 9 different environments representing combinations of 3 levels of apple reward mean and 3 levels of apple reward variance.

Implementation. The agent observes the top-down view of the current phase rendered in RGB and a binary variable indicating if the agent collected the key or not. The policy (θ) and the value functions (ψ and ϕ) are implemented by separate convolutional neural networks. The *meta-network* (η) computes the pairwise weight w_{ij} as follows: First, it embeds the observations s_i and s_j and the time difference $(j - i)$ into separate latent vectors. Then it takes the element-wise product of these three vectors to fuse them into a vector h_{ij}. Finally it maps h_{ij} to a scalar output that is bounded to $[0, 1]$ by sigmoid. We tuned hyperparameters for each method on the mid-level configuration $\langle \mu = 5, \sigma = 25 \rangle$ and kept them fixed for the other 8 configurations. Each method has a distinct set of parameters (e.g. outer-loop learning rates, λ). We referred to the original papers for the parameter ranges searched over.

Empirical Results. Figure 5.5 presents learning curves for Meta-PWTD, Meta-PWR, and baselines in three KtD configurations. Learning curves are shown separately for the *total episode return* and the *door phase reward*, the latter a measure of success at the long-term credit assignment. Not unexpectedly, H-PWR which uses handcrafted pairwise weights performs the best. The gap in performance between H-PWR and the best fixed-λ shows that this domain provides a credit assignment challenge that the pairwise-weighted advantage estimate can help with. The TVT and A2C-RR methods used a low discount factor and so relied solely on their heuristics for learning to pick up the key, but neither appears to enable fast learning in this domain. In the door phase, Meta-PWR is generally the fastest learner after H-PWR. Meta-PWTD, though slower, achieves optimal performance. Although RGM performs third best in the door phase, it does not perform well overall, suggesting that the inflexibility of its single set of reward weights (vs. pairwise of Meta-PWR) forces a trade off between short and long-term credit assignment. In summary, Meta-PWR outperforms all the other methods and Meta-PWTD is comparable to the baselines.

Figure 5.4 presents a visualization of the handcrafted weights for H-PWR (bottom left) and weights learned by Meta-PWR (bottom right). In each heatmap, the element on the i-th row and the j-th column denotes w_{ij}, the pairwise weight for computing the contribution of the reward

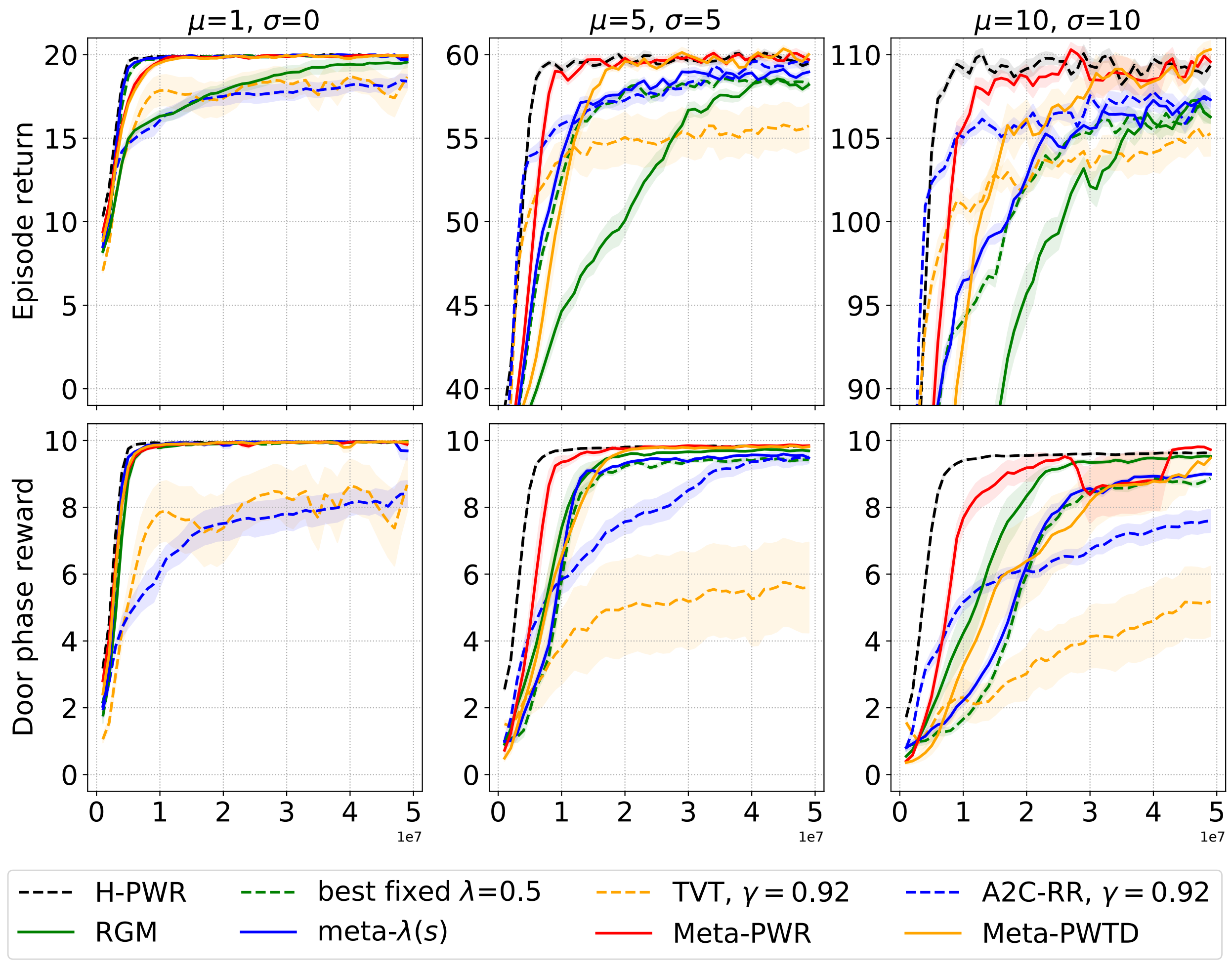


Figure 5.5: Learning curves for the KtD domain. Each column corresponds to a different configuration. The x-axis denotes the number of frames. The y-axis denotes the episode return in top row and the door phase reward in bottom row. The solid curves show the average over 10 independent runs and the shaded area shows the standard errors.

	A2C	A2C-RR	RGM	Meta-PWTD	Meta-PWR
Catch	5975	**5950**	7849	6096	5967
Catch Noise	42221	42295	48268	**41106**	43076
Catch Scale	56800	57033	54421	**48199**	49361
Umbrella Length	38050	38083	40397	**37973**	38168
Umbrella Distract	37524	37433	40159	37226	**36554**
Cartpole	76874	71506	119102	65945	**61752**
Discount Chain	3554	3548	2444	1040	**161**

Table 5.1: Total regret on selected `bsuite` domains (low is good).

upon transition to the j-th state to the return at the i-th state in the episode. In the heatmap of the handcrafted weights, the top-right area has non-zero weights because the rewards in the door phase depend on the actions selected in the key phase. The weights in the remaining part of the top rows are zero because those rewards do not depend on the the actions in the key phase. For the same reason, the weights in the middle-right area are zero as well. The weights in the rest of the area resemble the exponentially discounted weights with a discount factor of 0.92. This steep discounting helps fast learning of collecting apples. The learned weights largely resemble the handcrafted weights, which indicate that the metagradient procedure was able to simultaneously learn (1) the important rewards for the key phase are in the door phase, and (2) a quick-discounting set of weights within the apple phase that allows faster learning of collecting apples.

5.4.3 Experiments on Standard RL Benchmarks

Both the DAG and KtD domains are idealized credit assignment problems. However, in domains outside this idealized class, Meta-PWTD and -PWR may learn slower than baseline methods due to the additional complexity they introduce. To evaluate this possibility we compared them to baseline methods on `bsuite` [Osband et al., 2019] and Atari [Bellemare et al., 2013], both standard RL benchmarks. For these experiments, we did not compare to Meta-$\lambda(s)$ because it performed similarly to the fixed-λ baseline in previous experiments as noted in the original paper [Xu et al., 2018b].

`bsuite` is a set of unit-tests for RL agents: each domain tests one or more specific RL-challenges, such as exploration, memory, and credit assignment, and each contains several versions varying in difficulty. We selected all domains that are tagged by "credit assignment" and at least one other challenge. These domains are not designed solely as idealized credit assignment problems. We ran all methods for $100K$ episodes in each domain except *Cartpole*, which we ran for $50K$ episodes. Each run was repeated 3 times with different random seeds. Table 5.1 shows the total regret. Overall Meta-PWTD or -PWR achieved the lowest total regret in all domains except for *Catch*. It shows that Meta-PWTD and -PWR perform better than or comparably to the baseline methods even in domains without the idealized umbrella-like TCA structure.

To test scalability to high-dimensional environments, we conducted experiments on Atari. Atari games often have long episodes of more than 1000 steps thus episode truncation is required. However, the returns in RGM and Meta-PWR are not in a recursive additive form thus the common way of correcting truncated trajectories by bootstrapping from the value function is not applicable. TVT also requires full episodes for value transportation. Therefore, we excluded RGM, TVT, and Meta-PWR and only ran Meta-PWTD, A2C-RR and A2C. For each method we conducted hyperparameter search on a subset of 6 games and ran each method on 49 games with the fixed set of hyperparameters. An important hyperparameter for the A2C baseline is λ, which was set to 0.95.

Figure 5.6 (inset) shows the median human-normalized score during training. Meta-PWTD performed slightly better than A2C over the entire period, and both performed better than A2C-RR Figure 5.6 shows the relative performance of Meta-PWTD over A2C. Meta-PWTD outperforms A2C in 30 games, underperforms in 14, and ties in 5. These results show that Meta-PWTD can scale to high-dimensional environments like Atari. We conjecture that Meta-PWTD provides a benefit in games with embedded umbrella problems but this is hard to verify directly.

5.5 Conclusion

We presented two new advantage estimators with pairwise weight functions as parameters to be used in policy gradient algorithms, and a metagradient algorithm for learning the pairwise weight functions. Simple analysis and empirical work confirmed that the additional flexibility in our advantage estimators can be useful in domains with delayed consequences of actions, e.g., in umbrella-like problems. Empirical work also confirmed that the metagradient algorithm can learn the pairwise weights fast enough to be useful for policy learning, even in large-scale environments like Atari.

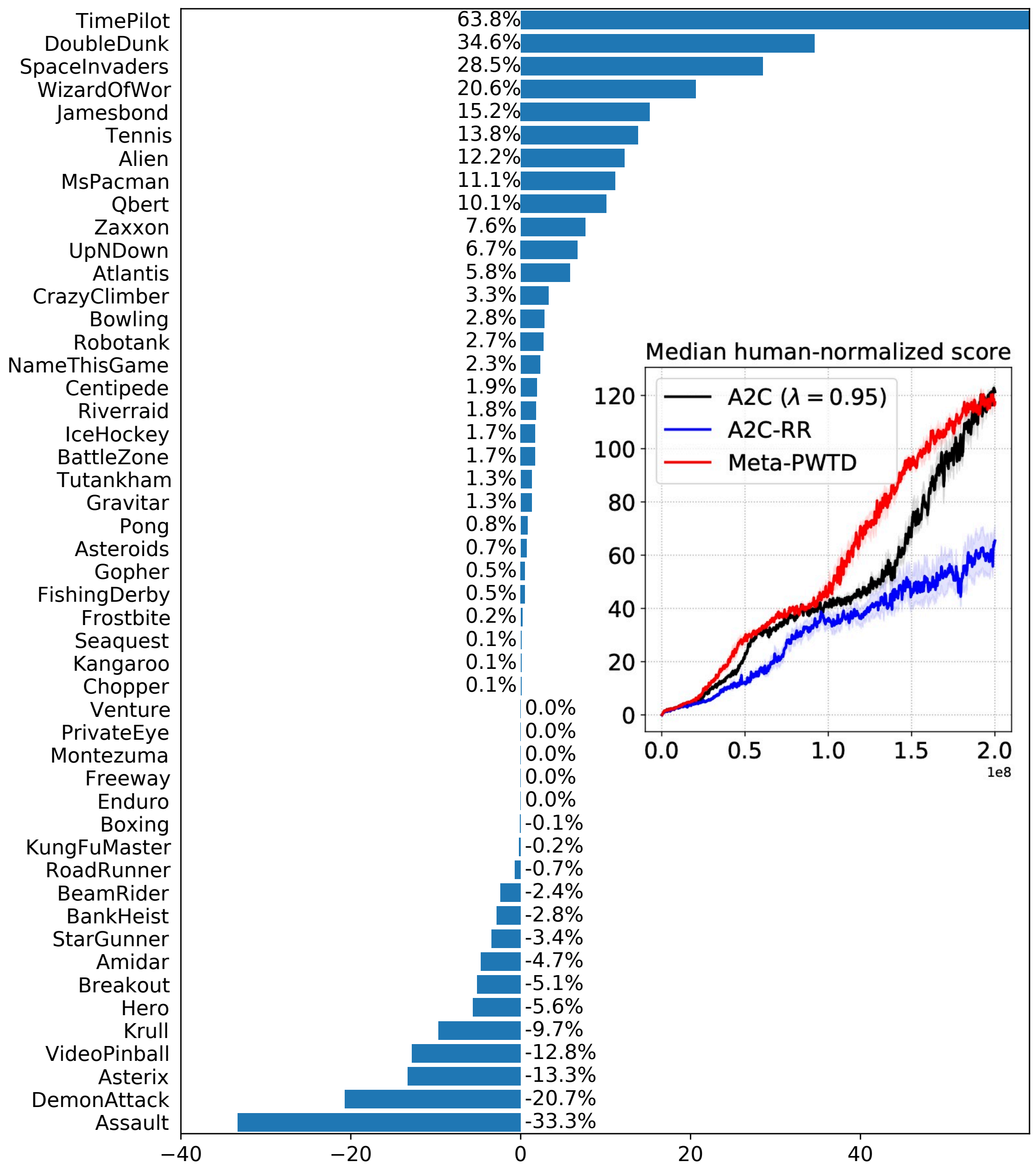


Figure 5.6: Relative performance of Meta-PWTD over A2C ($\lambda = 0.95$). All scores are averaged over 5 independent runs with different random seeds. Inset: Learning curves of median human normalized score of all 49 Atari games. Shaded area shows the standard error over 5 runs.

CHAPTER 6

Learning State Representations from Random Deep Action-conditional Predictions

Providing auxiliary tasks to Deep Reinforcement Learning (Deep RL) agents has become an important class of methods for driving the learning of state representations that accelerate learning on the main task. Some notable examples of existing auxiliary tasks include pixel control, reward prediction, termination prediction, and multi-horizon value prediction (these are reviewed in more detail below). In this work, we explore a different approach to providing auxiliary tasks in which a set of *random action-conditional prediction tasks* are generated through a rich space of general value functions (GVFs) defined by a language of predictions of random features of observations conditioned on a random sequence of actions.

Our main, and perhaps surprising, contribution in this chapter is an empirical finding that auxiliary tasks of learning random GVFs—again, random in both predicted features and actions the predictions are conditioned upon— yield state representations that produce control performance that is competitive with state-of-the-art auxiliary tasks with hand-crafted semantics. We demonstrate this competitiveness in Atari games and DeepMind Lab tasks, comparing to multi-horizon value prediction [Fedus et al., 2019], pixel control [Jaderberg et al., 2017], and CURL [Laskin et al., 2020] as our baseline auxiliary tasks. Note that while we present a reasonable approach to generating the semantics of the random GVFs we employ in our experiments, the specifics of our approach is not by itself a contribution (and thus not evaluated against other approaches to producing semantics for random GVFs), and alternative reasonable approaches for generating random GVFs could do as well.

Additionally, through empirical analyses on illustrative domains we show the benefits of exploiting the richness of GVFs—their temporal depth and action-conditionality. We also provide direct evidence that using random GVFs learns useful representations for the main task through *stop-gradient* experiments in which the state representations are trained *solely* via the random-GVF auxiliary tasks without using the usual RL learning with rewards to influence representation learning. We show that, again, surprisingly, these stop-gradient agents outperform the end-to-end-trained actor-critic baseline.

6.1 Related Work

Horde and PSRs. Auxiliary tasks were formalized and introduced to RL in [Sutton et al., 2011] through the Horde architecture. Horde is an off-policy learning framework for learning knowledge represented as GVFs from an agent's experience. Our work is related to Horde in the use of a rich subspace of GVF predictions but differs in that our interest is in the effect of learning these auxiliary predictions on the main task via shared state representations rather than to show the knowledge captured in these GVFs. Our work is also related to predictive state representations (PSRs) [Littman et al., 2001, Singh et al., 2004]. PSRs use predictions *as* state representations whereas our work learns latent state representations from predictions. Recently, in the use of deep neural networks in RL as powerful function approximators, various auxiliary tasks have been proposed to improve the latent state representations of Deep RL agents. We review these auxiliary tasks below. Our work belongs to this family of work in that the auxiliary prediction tasks are used to improve the state representations of Deep RL agents.

Auxiliary tasks using predefined GVF targets. *UNREAL* [Jaderberg et al., 2017] uses reward prediction and pixel control and achieved a significant performance improvement in DeepMind Lab but only marginal improvement in Atari. Termination prediction [Kartal et al., 2019] is shown to be an useful auxiliary task in episodic RL settings. SimCore DRAW [Gregor et al., 2019] learns a generative model of future observations conditioned on action sequences and uses it as an auxiliary task to shape the agent's belief states in partially observable environments. Fedus et al. [Fedus et al., 2019] found that simply predicting the returns with multiple different discount factors (MHVP) serves as effective auxiliary tasks. MHVP relies on the availability of rewards and thus is different from our work and other unsupervised auxiliary tasks.

Information-theoretic auxiliary tasks. Information-theoretic approaches to auxiliary tasks learn representations that are informative about the future trajectory of these representations as the agent interacts with the environment. CPC [van den Oord et al., 2018], CPC|action [Guo et al., 2018], ST-DIM [Anand et al., 2019], DRIML [Mazoure et al., 2020], and ATC [Stooke et al., 2021] apply different forms of temporal contrastive losses to learn predictions in a latent space. CURL [Laskin et al., 2020] ignores the long-term future and applies a contrastive loss on the stack of consecutive frames to learn good visual representations. PBL [Guo et al., 2020] focuses on partially observable environments and introduces a separate target encoder to set the prediction targets. The target encoder is trained to distill the learned state representations. SPR [Schwarzer et al., 2020] replaces the target encoder in PBL with a moving average of the state representation function. In addition to being predictive, PI-SAC [Lee et al., 2020] also enforces the state representations to be compressed. SPR and PI-SAC focuses on data efficiency and only conducted experiments under low data budgets. In these information-theoretic approaches, the targets are not GVFs and despite some empirical success, none could learn long-term predictions effectively. This is in contrast to GVF-like predictions which can be effectively learned via TD as in our work as well as the work presented above.

Theory. A few recent works [Bellemare et al., 2019, Dabney et al., 2020, Lyle et al., 2021] have studied the optimal representation in RL from a geometric perspective and provided theoretical insights into why predicting GVF-like targets is helpful in learning state representations. Our work is consistent with this theoretical motivation.

GVF discovery. Veeriah et al. [Veeriah et al., 2019] used metagradients to discover simple GVFs (discounted sums of features of observations) In this chapter, we show that random choices of features and random but rich GVFs are competitive with the state-of-the art of hand-crafted GVFs as auxiliary tasks.

GVF RNNs. Rather than using GVFs as auxiliary tasks, General Value Function Networks (GVFNs) [Schlegel et al., 2021] are a new family of recurrent neural networks (RNNs) where each dimension of the hidden state is a GVF prediction. GVFNs are trained by TD instead of truncated backprop through time. Our work relates to GVFNs in that both works use GVFs to shape state representations. However, unlike GVFNs, our work uses GVFs as auxiliary tasks and does not enforce any semantics to the state representations. Moreover, our empirical study mainly focuses on the control setting where the agent needs to maximize its long-term cumulative rewards, whereas [Schlegel et al., 2021] mainly focuses on time series modelling tasks and online prediction tasks and demonstrated the superior performance of GVFNs over conventional RNNs when the truncated input sequences are short during training.

6.2 Method

In this section we first describe the specific GVFs we studied in this chapter. Then we describe an algorithm for the construction of random GVFs. We finish this section with a description of the agent architecture used in our empirical work.

6.2.1 GVFs with Interdependent TD Relationships

In this chapter, we study auxiliary prediction tasks where the semantics of the predictions are defined by a set of GVFs with interdependent TD relationships (this family of GVFs are often referred as *temporal-difference networks* in the literature [Sutton et al., 2005, Sutton and Tanner, 2004]). The TD relationships among the GVFs can be described by a graph with directional edges,

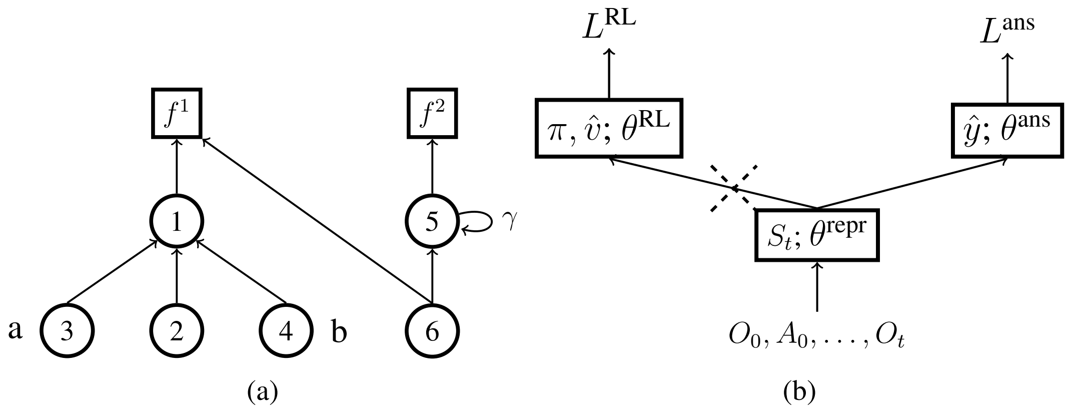

f^1
f^2
1
5
γ
a
3
2
4
b
6
(a)
L^{RL}
L^{ans}
$\pi, \hat{v}; \theta^{\text{RL}}$
$\hat{y}; \theta^{\text{ans}}$
$S_t; \theta^{\text{repr}}$
$O_0, A_0, \ldots, O_t$
(b)

Figure 6.1: **(a)** An example of a question network. The squares represent *feature nodes* and circles represent *prediction nodes*. **(b)** The agent architecture. The dashed cross denotes an optional `stop-gradient` operation.

which we call the *question network* as it defines the semantics of the predictions.

Figure 6.1a shows an example of a question network. The two squares represent two *feature nodes* and the six circles represent six *prediction nodes*. Node 1 (labeled in the circles) predicts the expected value of feature f^1 at the next step. Implicitly, this prediction is conditioned on following the current policy. Node 2 predicts the expected value of node 1 at the next step. Note that we can "unroll" the target of node 2 to ground it on the features. In this example, node 2 predicts the expected value of feature f^1 after two steps when following the current policy. Node 5 has a self-loop and predicts the expectation of the discounted sum of feature f^2 with a discount factor γ. We call node 5 a *discounted sum* prediction node. Node 3 is labeled by action a. It predicts the expected value of node 1 at the next step given action a is taken at the current step. We say node 3 is *conditioned* on action a. Similarly, node 4 predicts the same target but is conditioned on action b. Node 6 has two outgoing edges. It predicts the *sum* (in general a weighted sum, but in this chapter we do not explore the role of these weights and instead fix them to be 1) of feature f^1 and the value of node 5, both at the next step. In this case, it is hard to describe the semantic of node 6's prediction in terms of the features, but we can see that the prediction is still grounded on feature f^1 and f^2.

Generalising from the example above, a question network with n_p prediction nodes and n_f feature nodes defines n_p predictions of n_f features. We use N_p to denote the set of all prediction nodes and N_f to denote the set of all feature nodes. Let W be the adjacency matrix of the question network. W_{ij} denotes the weight on the edge from node i to node j. We define $W_{ij} \triangleq 0$ if there is no edge from node i to node j. Now consider an agent interacting with the environment. At each step t, it receives an observation O_t and takes an action A_t according to its policy π. Then at the next step it receives an observation O_{t+1}. The feature $f^k(O_t, A_t, O_{t+1})$ is a scalar function of the transition. The agent makes a prediction $\hat{y}^i(O_0, A_0, \ldots, O_t)$ for each prediction node i based on its history; this is computed by a neural network in our work. For brevity, we use f^k_{t+1} and $\hat{y}^i_t$ to denote $f^k(O_t, A_t, O_{t+1})$ and $\hat{y}^i(O_0, A_0, \ldots, O_t)$ respectively. The target for prediction i at step t is denoted by y^i_t. If prediction node i is not conditioned on any action, its target is

$$y^i_t = \mathbb{E}_\pi\big[\sum_{j \in N_p} W_{ij} y^j_{t+1} + \sum_{k \in N_f} W_{ik} f^k_{t+1}\big]$$

otherwise, if it is conditioned on action a^i, its target is

$$y^i_t = \mathbb{E}_\pi\big[\sum_{j \in N_p} W_{ij} y^j_{t+1} + \sum_{k \in N_f} W_{ik} f^k_{t+1} | A_t = a^i\big].$$

By the construction of the targets, the agent can learn the prediction $\hat{y}^i_t$ via TD. If i is not conditioned on any action, then $\hat{y}^i_t$ is updated by

$$\hat{y}^i_t \leftarrow \sum_{j \in N_p} W_{ij} \hat{y}^j_{t+1} + \sum_{k \in N_f} W_{ik} f^k_{t+1}$$

otherwise, if i is conditioned on action a^i, then $\hat{y}^i_t$ is updated by

$$\hat{y}^i_t \leftarrow \begin{cases} \sum_{j \in N_p} W_{ij} \hat{y}^j_{t+1} + \sum_{k \in N_f} W_{ik} f^k_{t+1} & \text{if } A_t = a^i \\ \hat{y}^i_t & \text{otherwise} \end{cases}$$

In an episodic setting, if the episode terminates at step T, we define $y^i_T \triangleq 0$ and $\hat{y}^i_T \triangleq 0$ for all prediction nodes i.

These GVFs represent a broad class of predictions. Many existing auxiliary prediction tasks can be expressed by a question network. Reward prediction [Jaderberg et al., 2017] can be represented by a question network with a single feature node representing the reward and a single prediction node predicting the reward. Multi-horizon value prediction [Fedus et al., 2019] can be represented by a similar question network but with multiple self-loop prediction nodes with different discounts. Termination prediction [Kartal et al., 2019] can be represented by a question network with a feature node of constant 1 and a self-loop node with discount 1.

6.2.2 A Random Question Network Generator

In this chapter, instead of hand-crafting a new question network instance as in previous work on the use of predictions for auxiliary tasks, we verify a conjecture that a large number of random deep, action-conditional predictions is enough to drive the learning of good state representations. To test this conjecture, we designed a generator of random question networks from which we can take samples and evaluate their performance as auxiliary tasks. Specifically, we designed a heuristic algorithm that generates question networks with random features and random structures.

Random Features. We use random features, each computed by a scalar function g^k with random parameters. For any transition (O_t, A_t, O_{t+1}), the feature is computed as $f^k_{t+1} = |g^k(O_{t+1}) - g^k(O_t)|$. Instead of directly using the output of g^k as the feature, we use the amount of change in g^k. A similar transformation was used in pixel control [Jaderberg et al., 2017].

Random Structure. We designed the random question network generator based on the following intuition. Each prediction corresponds to first executing an open-loop action sequence then following the agent's policy. Along the trajectory, it accumulates a feature-value (this would be the reward for the standard value function) at each step. Depending on the edges in the question network, the accumulated features can be different for different steps. As we will illustrate in Section 6.3, these predictions can provide rich training signals for learning good representations. Specifically, the generator takes 5 arguments as input: number of features n_f, the discrete action set $\mathcal{A}$, a discount factor γ, depth D, and repeat R. Its output is a question network that contains n_f feature nodes as defined above, and $D+1$ layers, each layer contains $R \times |\mathcal{A}|$ prediction nodes except the first layer which contains n_f prediction nodes. We construct the question network layer by layer incrementally from layer 0 to layer D. First, layer 0 has n_f feature nodes and n_f prediction nodes; each prediction node has an edge to a distinct feature node with weight 1 on the edge and has a self-loop with weight γ. Each prediction node in layer 0 predicts the discounted sum of its corresponding feature and are not conditioned on actions. Then for each layer l $(1 \leq l \leq D)$, we create $R \times |\mathcal{A}|$ prediction nodes. Each prediction node is conditioned on one action and there are exactly R nodes that are conditioned on the same action. Each prediction node has two edges, one to a random node in layer $l-1$ and one to a random feature node in layer 0. Note that prediction nodes in layer 1 do not necessarily connect to a self-loop prediction node in layer 0 - they may only connect to a feature node. A constraint for preventing duplicated predictions is included so that any two prediction nodes in layer l that are conditioned on the same action cannot connect to the same prediction node in layer $l-1$. In our preliminary experiments, we tried adding self-loops to deeper layers and allowing denser connections between nodes. Sometimes those additional loops and dense edges caused instability during training. We leave the study of more sophisticated question network structures to future work. Algorithm 4 shows the pseudocode for the random generator algorithm.

Algorithm 4 A Random Question Network Generator

```
**Input:** number of features $n_p$, discount factor $\gamma$, action set $\mathcal{A}$, depth $D$ and repeat $R$
**Output:** a network $G$
$G \leftarrow$ an empty graph
$roots \leftarrow$ an empty set
$leaves \leftarrow$ an empty set
for $i = 1$ to $n_p$ do
    create a new feature node $f$ in $G$
    $roots \leftarrow roots \cup \{f\}$
    $leaves \leftarrow leaves \cup \{f\}$
    create a new prediction node $v$ in $G$
    $leaves \leftarrow leaves \cup \{v\}$
    add edge $< v, f, 1 >$ to $G$
    add edge $< v, v, \gamma >$ to $G$
end for
for $d = 1$ to $D$ do
    $expanded \leftarrow$ an empty set
    for $a \in A$ do
        $parent \leftarrow$ randomly select $R$ nodes from $leaves$ without replacement
        for $p \in parent$ do
            create a new prediction node $v$ in $G$
            mark $v$ as conditioned on action $a$
            $expanded \leftarrow expanded \cup \{v\}$
            add edge $< v, p, 1 >$ to $G$
            $f \leftarrow$ randomly select a node from $roots$
            add edge $< v, f, 1 >$ to $G$
        end for
    end for
    $leaves \leftarrow expanded$
end for
```

6.2.3 Agent Architecture

We used a standard auxiliary-task-augmented agent architecture, as shown in Figure 6.1b. We base our agent on the actor-critic architecture and it consists of 3 modules. The state representation module, parameterized by θ^{repr}, maps the history of observations and actions $(O_0, A_0, \ldots, O_t)$ to a state vector S_t. The RL module, parameterized by θ^{RL}, maps the state vector S_t to a policy distribution over the available actions $\pi(\cdot|S_t)$ and an approximated value function $\hat{v}(S_t)$. The answer network module, parameterized by θ^{ans}, maps the state vector S_t to a set of predictions $\hat{y}(S_t)$ equal in size to the number of prediction nodes in the question network. Like in previous work, the augmented agent has more parameters than the base A2C agent due to the answer network module. However, the policy space and the value function space remain the same and these auxiliary parameters are

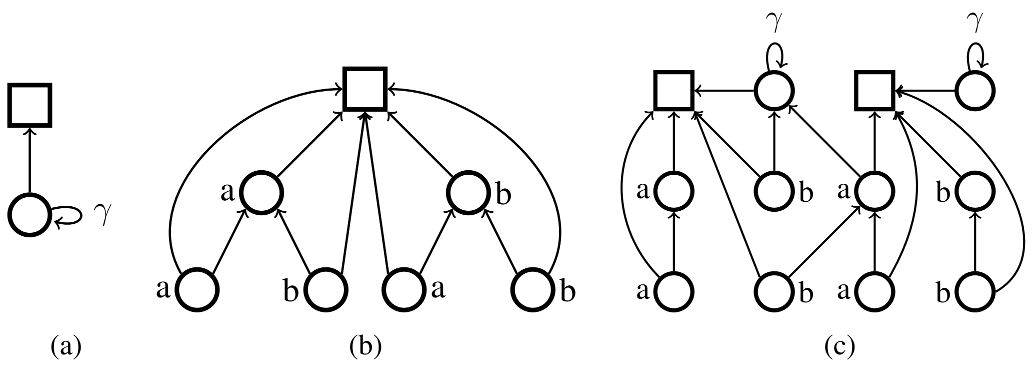


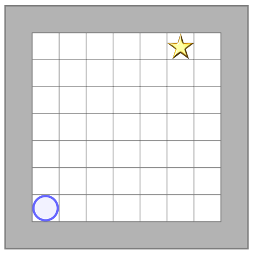
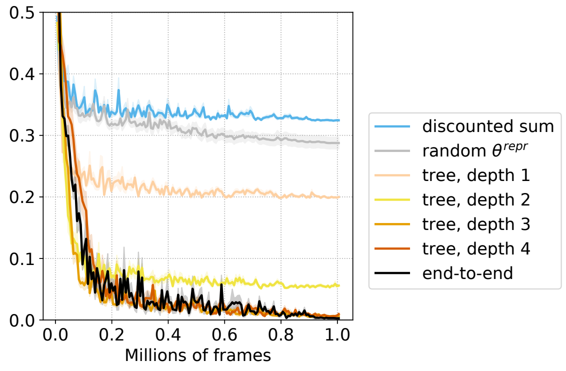
Figure 6.2: The question networks we studied in our illustrative experiment. **(a)** A discounted sum prediction. **(b)** A depth-2 tree question network with 2 actions. The bottom right prediction node predicts the sum of the values of the feature in the next two steps if action b were taken for both the current and the next step. Other prediction nodes have similar semantics. **(c)** A random question network sampled from rGVFs. There are 2 features and 2 actions, with depth and repeat set to 2.

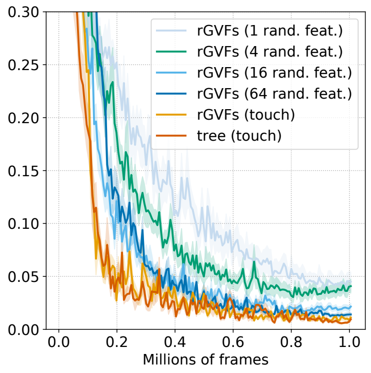
only used for providing richer training signals for the state representation module.

We trained the network in two separate ways. In the auxiliary task setting, the RL loss $\mathcal{L}^{\text{RL}}$ is backpropagated to update the parameters of the state representation (θ^{repr}) and the RL (θ^{RL}) modules, while the answer network loss $\mathcal{L}^{\text{ans}}$ is backpropagated to update the parameters of the answer network (θ^{ans}) and state representation (θ^{repr}) modules. Note that the answer network loss only affects the RL module indirectly through the shared state representation module. In the stop-gradient setting, we stopped the gradients from the RL loss from flowing from $\mathcal{L}^{\text{RL}}$ to θ^{repr}. This allows us to do a harsher and more direct evaluation of how well the auxiliary tasks can train the state representation without any help from the main task. For $\mathcal{L}^{\text{RL}}$, we used the standard actor-critic objective with an entropy regularizer. For $\mathcal{L}^{\text{ans}}$, we used the mean-squared loss for all the targets and predictions.

6.3 Illustrating the Benefits of Deep Action-conditional Questions

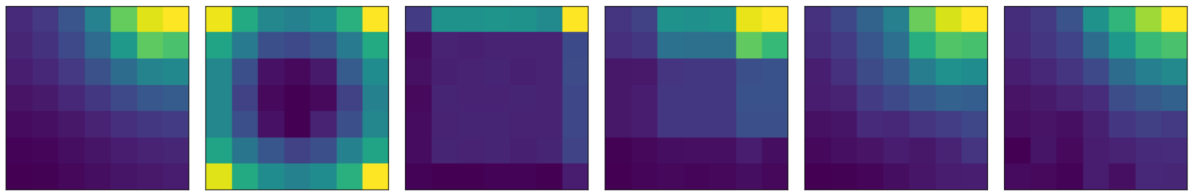
The main aim of the experiments in this section is to illustrate how deep action-conditional predictions can yield good state representations. We first use a simple grid world to visualize the impact of depth and action conditionality on the learned state representations. Then we demonstrate the practical benefit of exploiting both of these two factors by an ablation study on six Atari games. In addition, to test the robustness of the control performance to the hyperparameters of the random GVFs, we conducted a random search experiment on the Atari game Breakout.



(a) (b) (c)

Figure 6.3: **(a)** The illustrative grid world environment. The blue circle denotes the agent and the yellow star denotes the rewarding state. **(b)** MSE between the learned value function and the true value function in the tree question networks experiment. **(c)** MSE between the learned value function and the true value function in the random question networks experiment.



(a) (b) (c) (d) (e) (f)

Figure 6.4: Visualization of the learned value functions in the empty room environment. Bright indicates high value and dark indicates low value. **(a)** The true values. **(b)** The discounted sum predictions of the `touch` feature. **(c)** - **(f)** The prediction are defined by a full-tree-structured question network regarding the `touch` feature. The depth of the tree increases from 1 to 4 from **(c)** to **(f)**.

6.3.1 Benefits of Depth and Action-conditionality: Illustrative Grid World

Although our primary interest (and the focus of subsequent experiments) is learning good policies, in this domain we study *policy evaluation* because this simpler objective is sufficient to illustrate our points and we can compute and visualize the true value function for comparison. Figure 6.3a shows the environment, a 7 by 7 grid surrounded by walls. The observation is a top-down view including the walls. There are 4 available actions that move the agent horizontally or vertically to an adjacent cell. The agent gets a reward of 1 upon arriving at the goal cell located in the top row, and 0 otherwise. This is a continuing environment so achieving the goal does not terminate the interaction. The objective is to learn the state-value function of a *random policy* which selects each action with equal probability. We used a discount factor of 0.98.

Specifying a question network requires specifying both the structure and the features. Later we explore random features, but here we use a single hand-crafted `touch` feature so that every prediction has a clear semantic. `touch` is 1 if the agent's move is blocked by the wall and is 0 otherwise.

Using the `touch` feature we constructed two types of question networks. The first type is the discounted sum prediction of `touch` (we used a discount factor 0.8) (Figure 6.2a). The second type is a *full action-conditional tree* of depth D. There is only one feature node in the tree which corresponds to the `touch` feature. Each internal node has 4 child nodes conditioned on distinct actions. Each prediction node also has a skip edge directly to the feature node (except for the child nodes of the feature node). Figure 6.2b shows an example of a depth-2 tree (the caption describes the semantics of some of the predictions). We also compared to a randomly initialized state representation module as a baseline where the state representation was fixed and only the value function weights were learned during training.

Neural Network Architecture. The empty room environment is fully observable and so the state representation module is a feed-forward neural network that maps the current observation O_t to a state vector S_t. It is parameterized by a 3-layer multi-layer perceptron (MLP) with 64 units in the first two layers and 32 units in the third layer. The RL module has one hidden layer with 32 units and one output head representing the state value. (There is no policy head as the policy was given). The answer network module also has one hidden layer with 32 units and one output layer. We applied a stop-gradient between the state representation module and the RL module (Figure 6.1b).

Results. We measured the performance by the mean-squared error (MSE) between the learned value function and the true value function across all states. The true value function was obtained by solving a system of linear equations [Sutton and Barto, 2018]. Figure 6.3b shows the MSE during training. Both the random baseline and the discounted sum prediction target performed poorly. But even a tree question network of depth 1 (i.e., four prediction targets corresponding to the four action conditional predictions of `touch` after one step) performed much better than these two baselines. Performance increased monotonically with increasing depth until depth 3 when the MSE matched end-to-end training after 1 million frames.

Figure 6.4 shows the different value functions learned by agents with the different prediction tasks. Figure 6.4a visualizes the true values. Figure 6.4b shows the learned value function when the state representations are learned from discounted sum predictions of `touch`. Its symmetric pattern reflects the symmetry of the grid world and the random policy, but is inconsistent with the asymmetric true values. Figure 6.4c shows the learned value function when the state representations are learned from depth-1-tree predictions. It clearly distinguishes 4 corner states, 4 groups of states on the boundary, and states in the center area, as this grouping reflects the different prediction targets for these states.

For the answer network module to make accurate predictions of the targets of the question network, the state representation module must map states with the same prediction target to similar representations and states with different targets to different representations. As the question network tree becomes deeper, the agent learns finer state distinctions, until an MSE of 0 is achieved at depth 3 (Figure 6.4e).

6.3.2 Benefits of Random Question Nets: Illustrative Grid World

The previous experiment demonstrated benefits of temporally deeper action-conditonal prediction tasks. But achieving this by creating deeper and deeper full-branching action-conditional trees is not tractable as the number of prediction targets grows exponentially. The previous experiment also used a single feature formulated using domain knowledge; such feature selection is also not scalable. The random generator described in Section 6.2 provides a method to mitigate both concerns by growing random question networks with random features.

Specifically, we used *discount* 0.8, *depth* 4, and *repeat* equal to the number of features for generating random GVFs. Figure 6.3c shows the MSE of different random GVF variants. The performance of random GVFs with `touch`—that is, random but not necessarily full branching trees of depth 4—performed as well as `touch` with a full tree of depth 4. Random GVFs with a single random feature performed suboptimally; a random feature is likely less discriminative than `touch`. However, as the number of random features increases, the performance improves, and with 64 random features, random GVFs match the final performance of `touch` with a full depth 4 tree.

The results on the grid world provide preliminary evidence that random deep action-conditional GVFs with many random features can yield good state representations. We next test this conjecture on a set of Atari games, exploring again the benefits of depth and action conditionality.

6.3.3 Ablation Study of Benefits of Depth and Action Conditionality: Atari

Here we use six Atari games [Bellemare et al., 2013] (these six are often used for hyperparameter selection for the Atari benchmark [Mnih et al., 2016]) to compare four different kinds of random GVF question networks: **(a)** random GVFs in which all predictions are *discounted sums* of distinct random features (illustrated in Figure 6.2a and denoted rGVFs-*discounted-sum* in Figure 6.5); **(b)** random GVFs in which all predictions are *shallow action-conditional* predictions, a set of depth-1 trees, each for a distinct random feature (denoted rGVFs-*shallow* in Figure 6.5); **(c)** random GVFs without action-conditioning (denoted rGVFs-*no-actions* in Figure 6.5); and **(d)** random GVFs that exploit both action conditionality and depth (illustrated in Figure 6.2c and denoted simply by rGVFs in Figure 6.5).

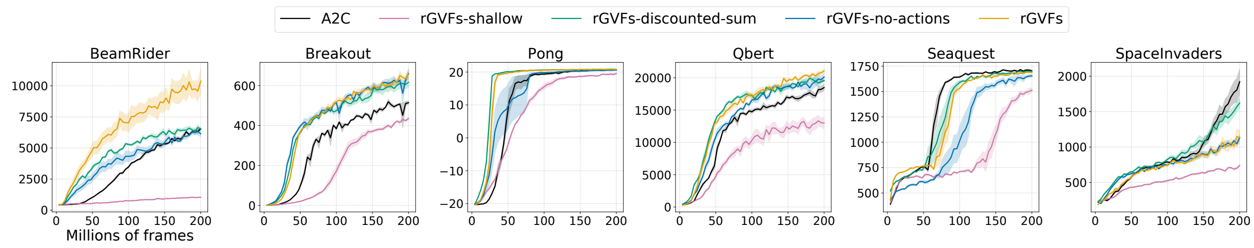


Figure 6.5: Learning curves of different question networks in six Atari games. x-axis denotes the number of frames and y-axis denotes the episode returns. Each curve is averaged over 5 independent runs with different random seeds. Shaded area shows the standard error.

Random Features for Atari. The random function g for computing the random features are designed as follows. The 84×84 observation O_t is divided into 16 disjoint 21×21 patches, and a *shared* random linear function applies to each patch to obtain 16 random features $g_t^1, g_t^2, \ldots, g_t^{16}$. Finally, we process these features as described in §6.2.2.

Neural Network Architecture. We used A2C [Mnih et al., 2016] with a standard neural network architecture for Atari [Mnih et al., 2015] as our base agent. Specifically, the state representation module consists of 3 convolutional layers. The RL module has one hidden dense layer and two output heads for the policy and the value function respectively. The answer network has one hidden dense layer with 512 units followed by the output layer. We stopped the gradient from the RL module to the state representation module.

Hyperparameters. The discount factor, depth, and repeat were set to 0.95, 8, and 16 respectively. Thus there are $16 + 8 * 16 * |\mathcal{A}|$ total predictions. Random GVFs without action-conditioning has the same question network except that no prediction was conditioned on actions. To match the total number of predictions, we used $16 + 8 * 16 * |\mathcal{A}|$ random features for discounted sum and $8 * 16$ features for shallow action-conditional predictions. Additional random features were generated by applying more random linear functions to the image patches. The discount factor for discounted sum predictions is also 0.95.

Results. Figure 6.5 shows the learning curves in the 6 Atari games. rGVFs-*shallow* performed the worst in all the games, providing further evidence for the value of making deep predictions. rGVFs consistently outperformed rGVFs-*no-actions*, providing evidence that action-conditioning is beneficial. And finally, rGVFsperformed better than rGVFs-*discounted-sum* in 3 out of 6 games (large difference in BeamRider and small differences in Breakout and Qbert), was comparable in 2 the other 3 games, and performed worse in one—despite using many fewer features than rGVFs-*discounted-sum*. This suggests that structured deep action-conditional predictions can be more effective than simply making discounted sum predictions about many features.

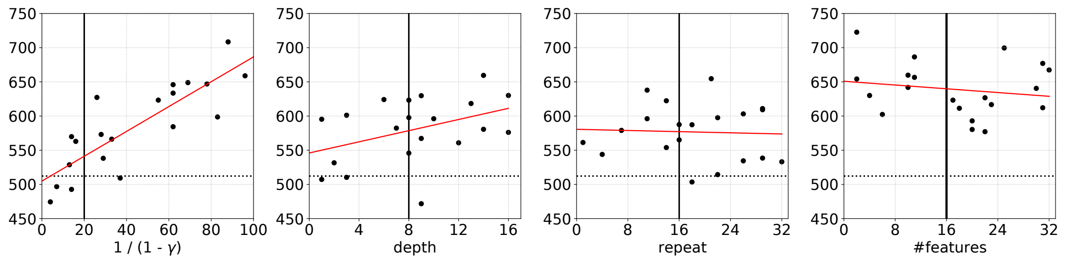


Figure 6.6: Scatter plots of scores in Breakout obtained by rGVFs with different hyperparameters. x-axis denotes the value of the hyperparameter. y-axis denotes the final game score after training for 200 million frames. The red line in each panel is the line of best fit. The dotted horizontal lines denote the performance of the end-to-end A2C baseline. The solid vertical lines denotes the values we used in our final experiments.

6.3.4 Robustness and Stability

We tested the robustness of rGVFs with respect to its hyperparameters, namely discount, depth, repeat, and number of features. We explored different values for each hyperparameter independently while holding the other hyperparameters fixed to the values we used in the previous experiment. For each hyperparameter, we took 20 samples uniformly from a wide interval and evaluated rGVFs on Breakout using the sampled value. The results are presented in Figure 6.6. The lines of best fit (the red lines) in the left two panels indicate a positive correlation between the performance and the depth of the predictions, which is consistent with the previous experiments. Each hyperparameter has a range of values that achieves high performance, indicating that rGVFs are stable and robust to different hyperparameter choices.

6.4 Comparison to Baseline Auxiliary Tasks

In this section, we present the empirical results of comparing the performance of rGVFs against the A2C baseline [Mnih et al., 2016] and three other auxiliary tasks, i.e., multi-horizon value prediction (MHVP) [Fedus et al., 2019], pixel control (PC) [Jaderberg et al., 2017], and CURL [Laskin et al., 2020]. We conducted the evaluation in 49 Atari games [Bellemare et al., 2013] and 12 DeepMind Lab environments [Beattie et al., 2016]. It is unclear how to apply CURL to partially observable environments which require long-term memory because CURL is specifically designed to use the stack of recent frames as the inputs. Thus we did not compare to CURL in the DeepMind Lab environments. Our implementation of rGVFs for this experiment is available at `https://github.com/Hwhitetooth/random_gvfs`.

Atari Implementation. We used the same architecture for rGVFs as in the prior section. For MHVP, we used 10 value predictions following [Fedus et al., 2019]. Each prediction has a

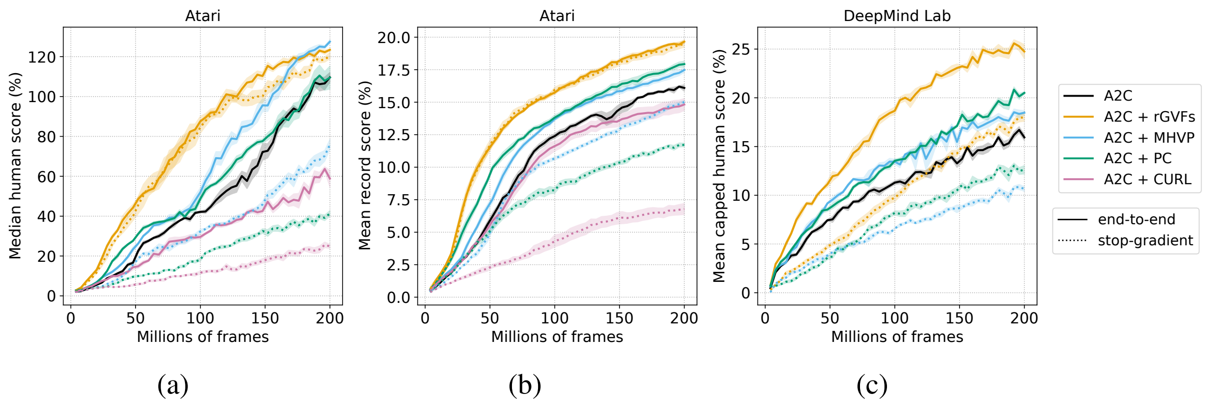


Figure 6.7: **(a)** Median human-normalized score across 49 Atari games. **(b)** Mean record-normalized score across 49 Atari games. **(c)** Mean capped human-normalized score across 12 DeepMind Lab environments. In all panels, the x-axis denotes the number of frames. Each dark curve is averaged over 5 independent runs with different random seeds. The shaded area shows the standard error.

unique discount factor, chosen to be uniform in terms of their effective horizons from 1 to 100 ($\{0, 1-\frac{1}{10}, 1-\frac{1}{20}, \ldots, 1-\frac{1}{90}\}$). The architecture for MHVP is the same as rGVFs. For PC, we followed the architecture design and hyperparameters in [Jaderberg et al., 2017]. For CURL, we implemented it in our experiment setup by using the code accompanying the paper as a reference [1]. When not stopping gradient from the RL loss, we mixed the RL updates and the answer network updates by scaling the learning rate for the answer network with a coefficient c. We searched c in $\{0.1, 0.2, 0.5, 1, 2\}$ on the 6 games in the previous section. $c = 1$ worked the best for all methods.

DeepMind Lab Implementation. We used the same RL module and answer network module as Atari but used a different state representation module to address the partial observability. Specifically, the convolutional layers in the state representation module were followed by a dense layer with 512 units and a GRU core [Cho et al., 2014, Chung et al., 2014] with 512 units.

Results. Figure 6.7a and Figure 6.7b shows the results for both the stop-gradient and end-to-end architectures on Atari, comparing to two standard human-normalized score measures (median human-normalized score [Mnih et al., 2015] and mean record-normalized score [Hafner et al., 2020]). When training representations end-to-end through a combined main task and auxiliary task loss, the performance of rGVFs matches or substantially exceeds the three baselines. Although the original paper shows that CURL improves agent performance in the data-efficient regime (i.e., $100K$ interactions in Atari), our results indicate that it hurts the performance in the long run. We conjecture that CURL is held back by failing to capture long-term future in representation learning. Surprisingly, the stop-gradient rGVFs agents outperform the end-to-end A2C baseline, unlike stop-

[1] https://github.com/aravindsrinivas/curl_rainbow

gradient versions of the baseline auxiliary task agents. Figure 6.7c shows the results for both stop-gradient and end-to-end architectures on 12 DeepMind Lab environments (using mean capped human-normalized scores). Again, rGVFs substantially outperforms both auxiliary task baselines, and the stop-gradient version matches the final performance of the end-to-end A2C. Taken together the results from these 61 tasks provide substantial evidence that rGVFs drive the learning of good state representations, outperforming auxiliary tasks with fixed hand-crafted semantics.

6.5 Conclusion and Future Work

In this chapter we provided evidence that learning random deep action-conditional predictions can drive the learning of good state representations. We explored a rich space of GVFs that can be learned efficiently with TD methods. Our empirical study on the Atari and DeepMind Lab benchmarks shows that learning state representations solely via auxiliary prediction tasks defined by random GVFs outperforms the end-to-end trained A2C baseline. Random GVFs also outperformed pixel control, multi-horizon value prediction, and CURL when being used as part of a combined loss function with the main RL task.

In this chapter, the question network was sampled before learning and was held fixed during learning. An interesting goal for future research is to find methods that can adapt the question network and discover useful questions during learning. The question networks we studied are limited to discrete actions. It is unclear how to condition a prediction on a continuous action. Thus another future direction to explore is to extend action-conditional predictions to continuous action spaces.

CHAPTER 7

G-VUZero: Planning with Models Learned Using Generalized Value-equivalence Updates

Model-based reinforcement learning (RL) methods such as Value Prediction Networks and the state of the art MuZero use value-equivalence to learn models. Their implementation of value-equivalence updates the model by equating (a) the value of the state reached by taking a sequence of actions in the real world with (b) the value of the state reached by taking the *same* action-sequence in the model. We present Generalized Value-equivalence Updates (G-VU) that equate (a) the value of a state reached by taking an action sequence in the real world followed by an action sequence in the model with (b) the value of a state reached by taking the *concatenation* of the two action sequences entirely in the model. Crucially, this means that G-VU is not restricted to update only on action sequences (and thus states) experienced in the world. Our algorithm G-VUZero uses MuZero's method for acting in the world via MCTS-planning, but uses G-VU model-updates instead of the more restricted model-updates used by MuZero. In particular, we explore updating the model for actions likely to be queried by the MCTS-planner while acting. Our empirical results show that G-VUZero can sometimes outperform the strong MuZero baseline.

7.1 Related Work

Our work falls into the category of model-based approaches in RL. Learning a model of the environment can benefit the agent in various ways such as value learning or policy optimization by simulating trajectories [Sutton, 1991, Oh et al., 2015, Racanière et al., 2017, Ha and Schmidhuber, 2018, Hafner et al., 2019a, 2020], guiding real-time action selection via lookahead search [Oh et al., 2017, Hafner et al., 2019b, Hessel et al., 2021], improving state representation learning [Zhang et al., 2018, Hessel et al., 2021], and encouraging exploration [Oh et al., 2015, Pathak et al., 2017, Filos et al., 2021]. See [Moerland et al., 2020] for a survey of model-based RL.

Models learned by different objectives capture different aspects of the environments. In this chapter, we focus on a specific class of models called value-equivalent (VE) models [Grimm et al., 2020] (previously known as “value-aware” model learning [Farahmand et al., 2017]). VE models focus on the task-related aspects of the environment, i.e., rewards and values. Examples of VE models include value iteration networks [Tamar et al., 2016], TreeQN and ATreeC [Farquhar et al., 2018], the Predictron [Silver et al., 2017], value prediction networks [Oh et al., 2017], and MuZero [Schrittwieser et al., 2020]. By combining a learned VE model with Monte-Carlo tree search (MCTS) [Kocsis and Szepesvári, 2006], MuZero demonstrates strong empirical success in a wide range of environments including classic board games Go, Chess, and Shogi, Atari video games, and continuous control tasks [Schrittwieser et al., 2020, Hubert et al., 2021]. Our work proposes a novel general value-equivalence update (G-VU) for learning VE models. In particular, we implement G-VU with MuZero and build a new agent called G-VUZero. However, G-VU is not tied to MuZero and can be implemented with other VE models as well.

Recently, Farquhar et al. [2021] studied self-consistency (SC) in VE models and proposed an update to enforce joint self-consistency between the model and the value function. Our work is related to SC in that both G-VU and SC can update the model on action sequences that are not experienced in the environment and thus can provide additional learning signals to the model. But our work also differs from SC significantly. G-VU is a complete method for learning a VE model with respect to the environment. In contrast, self-consistency is only a necessary condition for value-equivalence but is not a sufficient condition. Therefore, SC must be combined with other VE model learning methods such as MuZero. In our experiments, we compare G-VUZero to an agent that combines MuZero and SC.

7.2 Background

In this section, we provide some background knowledge that our work builds upon. We first do a brief review of the value-equivalence principle. Then we introduce the MuZero agent which is used for our empirical study.

7.2.1 Value-equivalent models

Model-based reinforcement learning (MBRL) algorithms is a class of RL algorithms that learn an approximated *model* of the environment as an intermediate step. We use $m^* = (r, p)$ to denote the environment dynamics and $\hat{m} = (\hat{r}, \hat{p})$ to denote the approximated model. In this chapter, we are interested in a particular class of models called *value-equivalent* (VE) models [Grimm et al., 2020]. A model $\hat{m}$ is said to be value-equivalent to the environment with respect to the policy π and a function f if

$$\mathcal{T}^{\pi}_{\hat{m}} f = \mathcal{T}^{\pi}_{m^*} f, \tag{7.1}$$

where $\mathcal{T}_m^\pi$ denotes the Bellman operator [Bellman, 1966] induced by the policy π and the model m. For brevity, we will omit the superscript π unless it causes ambiguity. By applying the Bellman operator multiple times, we can generalize the definition of value-equivalence to *order-k value-equivalence* [Grimm et al., 2021]. A model $\hat{m}$ is order-k value-equivalent to the environment with respect to the policy π and a function f if

$$\mathcal{T}_{\hat{m}}^{(k)} f = \mathcal{T}_{m^*}^{(k)} f, \tag{7.2}$$

where $\mathcal{T}^{(k)}$ denotes k applications of the Bellman operator. By taking k to infinity, we obtain *proper value-equivalence* (PVE) [Grimm et al., 2021].

7.2.2 MuZero: VE models in practice

Many existing model-based RL agents can be viewed as different implementations of VE models. MuZero [Schrittwieser et al., 2020] is a particular instance that combines VE models with Monte-Carlo tree search (MCTS) and demonstrates great empirical success. A MuZero agent consists of three functions. In practice, the environment state s_t is rarely available and the agent only receives an observation o_t instead. Thus the *representation* function $\hat{s}_t = h_{\theta^r}(h_t)$ encodes a history $h_t = (o_0, a_0, o_1, \ldots, o_t)$ to a compact state vector $\hat{s}_t$. The *dynamic* function $\hat{r}_t^{k+1}, \hat{s}_t^{k+1} = g_{\theta^d}(\hat{s}_t^k, a_t^k)$ takes a state $\hat{s}_t^k$ and an action a_t^k as inputs and outputs a reward prediction $\hat{r}_t^{k+1}$ and a new state $\hat{s}_t^{k+1}$. To distinguish real time steps and simulated time steps, we use superscripts to denote simulated time steps. For example, $\hat{s}_t^k$ denotes the k-th simulated step starting from the state at real time step t. Note that $\hat{s}_t^0 = \hat{s}_t$. The *prediction* function $\hat{v}_t^k, \pi_t^k = f_{\theta^p}(\hat{s}_t^k)$ maps a state $\hat{s}_t^k$ to a value prediction $\hat{v}_t^k$ and a policy prediction π_t^k. Let θ^r, θ^d, and θ^p denote the parameters of the three functions respectively. The agent parameters $\theta = (\theta^r, \theta^d, \theta^p)$ is the union of the parameters of the three functions.

The parameters are updated as follows during training. For a trajectory o_t, a_t, r_{t+1}, o_{t+1}, $\ldots$, o_{t+K}, we *unroll* the model from $\hat{s}_t$ over the action sequence a_t, $\ldots$, a_{t+K-1} and obtain the reward predictions, the value predictions, and the policy predictions along the way. Then the parameters are updated by stochastic gradient descent to minimize the loss function

$$\mathcal{L}^{\text{MZ}}(\theta) = \sum_{k=1}^{K} \mathcal{L}^r(\hat{r}_t^k, r_{t+k}) + \sum_{k=0}^{K} \mathcal{L}^v(\hat{v}_t^k, v_{t+k}^{\text{target}}) + \sum_{k=0}^{K} \mathcal{L}^\pi(\pi_t^k, \pi_{t+k}^{\text{MCTS}}), \tag{7.3}$$

where $\mathcal{L}^r$, $\mathcal{L}^v$, and $\mathcal{L}^\pi$ denote suitable loss functions for the reward prediction, the value prediction, and the policy prediction respectively. The value target is the n-step return $v_{t+k}^{\text{target}} = \sum_{i=1}^n \gamma^{i-1} r_{t+k+i} + \gamma^n v_{t+k+n}^{\text{MCTS}}$. The policy target π^{MCTS} and the bootstrap value v^{MCTS} are com-

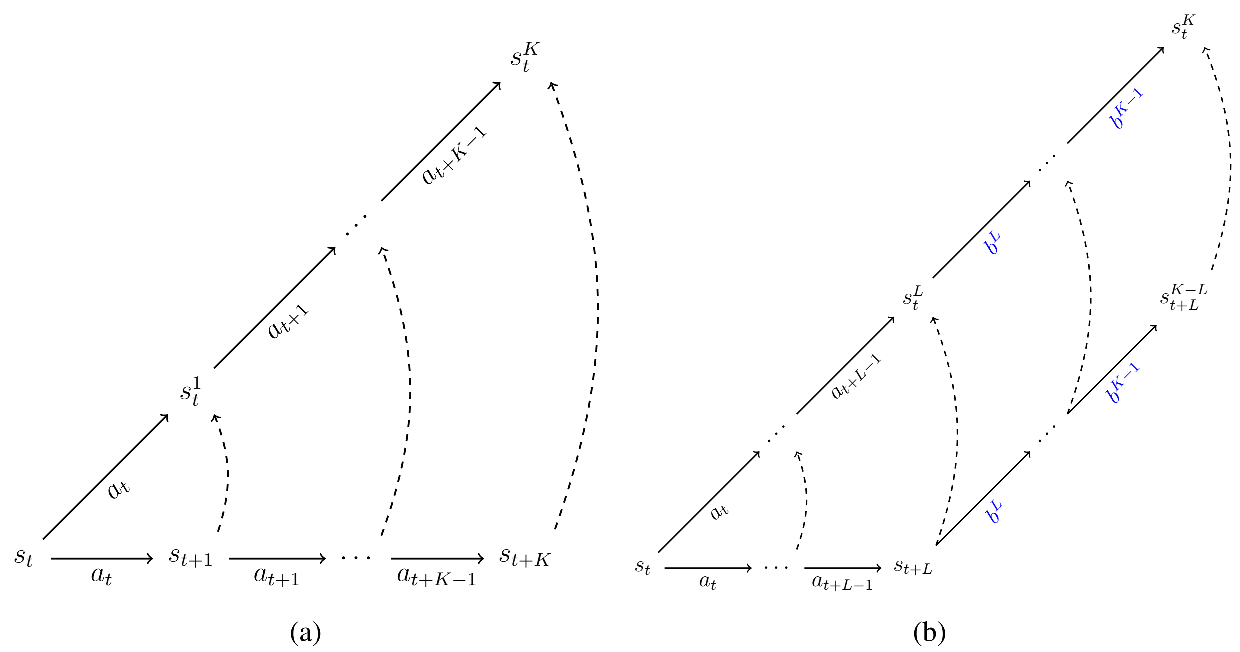


Figure 7.1: Illustration of VE updates. (a) Direct VE updates. (b) Generalized VE updates.

puted based on the outputs of MCTS.

7.3 Generalized Value-equivalence Updates

In this section, we first provide a brief review on the direct value-equivalence update (D-VU) used by existing agents. Then we propose a generalized value-equivalence update (G-VU) which is a strict generalization of D-VU. After that, we discuss the additional flexibility provided by G-VU and its potential benefit. We finish this section by presenting a new agent that combines G-VU with MuZero, which we name G-VUZero.

7.3.1 Direct Value-equivalence Update

Many recently MBRL agents can be viewed as learning an approximated value function $\hat{v}$ and a VE model $\hat{m}$ with respect to $\hat{v}$ simultaneously. In this chapter, we are interested in the VE model learning part. As suggested by Grimm et al. [2021], most of these existing methods are effectively minimizing the order-k value-equivalence loss for all $k = 1, \dots, K$:

$$\mathcal{L}^{\text{D-VU}}(\hat{m}) = \sum_{k=1}^{K} \|\mathcal{T}_{\hat{m}}^{(k)}\hat{v} - \mathcal{T}_{m^*}^{(k)}\hat{v}\|. \tag{7.4}$$

However, it is often infeasible to compute this loss in practice because it requires the transition function of the environment which is typically unknown. Therefore, we often approximate Eq. 7.4 by the following loss computed on a real state s_t and the subsequent action sequence $a_{t:t+K-1}$:

$$\mathcal{L}^{\text{D-VU}}(\hat{m}|a_{t:t+K-1}) = \sum_{k=1}^{K} \big[\mathcal{L}^r(\hat{r}_t^k, r_{t+k}) + \mathcal{L}^v(\hat{v}_t^k, \hat{v}_{t+k})\big]. \tag{7.5}$$

Note that the action sequence $a_{t:t+K-1}$ can be sampled from any policy because Eq. 7.5 enforces consistency on each individual action sequence. We call this the *direct* value-equivalence update (D-VU) because Eq. 7.4 directly equates k applications of the model Bellman operator with k applications of the environment Bellman operator. Figure 7.1a illustrates how the practical D-VU loss is computed.

7.3.2 Generalized Value-equivalence Update

Note that if a model is order-k VE to the environment with respect to $\hat{v}$ for all $k = 1, \ldots, K$, it also satisfies the following equation:

$$\mathcal{T}_{\hat{m}}^{(k)} \hat{v} = \mathcal{T}_{m^*}^{(l_k)} \mathcal{T}_{\hat{m}}^{(k-l_k)} \hat{v}, \tag{7.6}$$

for all $k = 1, \ldots, K$ and any $l_k \in \{1, \ldots, k\}$. In fact, the other direction is also true, i.e., if Eq. 7.7 is satisfied for all $k = 1, \ldots, K$ and some $l_k \in \{1, \ldots, k\}$, then Eq. 7.4 is also satisfied for all $k = 1, \ldots, K$. Based on this observation, we propose a *general* value-equivalence update (G-VU) that minimizes the following loss:

$$\mathcal{L}^{\text{G-VU}}(\hat{m}) = \sum_{k=1}^{K} \|\mathcal{T}_{\hat{m}}^{(k)} \hat{v} - \mathcal{T}_{m^*}^{(l_k)} \mathcal{T}_{\hat{m}}^{(k-l_k)} \hat{v}\|. \tag{7.7}$$

Different from D-VU, G-VU equates k applications of the model Bellman operator with $k - l_k$ applications of the model Bellman operator followed by l_k applications of the environment Bellman operator. Note that D-VU is a special case of G-VU where $l_k = k$ for all k.

Like D-VU, the loss in Eq. 7.7 is often infeasible to compute in practice but we can approximate it as follows. Starting from a real state s_t, we first execute an action sequence $a_{t:t+L-1}$ in the environment and reach s_{t+L}, where $L \leq K$ is a hyperparameter. Then we unroll the model from s_{t+L} along a second action sequence $b_{L:K-1}$ and reach s_{t+L}^{K-L}. After that, we unroll the model from s_t along the *concatenation* of $a_{t:t+L-1}$ and $b_{L:K-1}$ and reach s_t^K. Finally, we update the model to minimize the following loss:

$$\begin{aligned}\mathcal{L}^{\text{G-VU}}(\hat{m}|a_{t:t+L-1}, b_{L:K-1}) = &\sum_{k=1}^{L} \left[\mathcal{L}^r(\hat{r}_t^k, r_{t+k}) + \mathcal{L}^v(\hat{v}_t^k, \hat{v}_{t+k})\right] \\ &+ \sum_{k=L+1}^{K} \left[\mathcal{L}^r(\hat{r}_t^k, \hat{r}_{t+L}^{k-L}) + \mathcal{L}^v(\hat{v}_t^k, \hat{v}_{t+L}^{k-L})\right].\end{aligned} \tag{7.8}$$

The first summation measures the errors between the reward predictions and the value predictions along the simulated trajectory in the model and the rewards and the value predictions along the real trajectory. This corresponds to the applications of the environment Bellman operator on the RHS of Eq. 7.7. The second summation measures the errors between the reward predictions and the value predictions along the simulated trajectory starting from s_t and the reward predictions and the value predictions along the simulated trajectory starting from s_{t+L}. This corresponds to the applications of the model Bellman operators on the RHS of 7.7. This loss function corresponds to setting $l_k = \min\{k, L\}$ for all k. Similar to D-VU, $a_{t:t+L-1}$ can be sampled from any policy and $b_{L:K-1}$ can be any action sequence of length $K - L$ (there are exponentially many of them). Figure 7.1b provides an illustration of the practical G-VU loss is computed.

7.3.3 Additional flexibility and potential benefits of G-VU

The flexibility that G-VU provides is that the second action sequence $b_{L:K-1}$ does not need to coincide with the agent's actions in the environment. This enables updating the model on action sequences that are not experienced in the environment. For D-VU, the updates are restricted to the action sequence $a_{t:t+K-1}$ that is executed in the environment following s_t. But for G-VU, given the same observed action sequence $a_{t:t+L-1}$, we can update the model on more action sequences by sampling different suffixes $b_{L:K-1}$. Therefore, G-VU could potentially improve the data efficiency of model learning as it provides additional learning signals using the same amount of data. In Section 7.4, we will provide empirical evidence supporting this hypothesis.

7.3.4 G-VUZero: Implementing G-VU with MuZero

Now we present a concrete implementation of G-VU with MuZero. We chose to adapt MuZero for its strong empirical performance.

One important question we need to address is how to choose the suffix action sequence b. Recall that there are exponentially many choices of b for a given L and K. But in practice we can only choose a small set of bs for each model parameter update due to resource constraints, thus we want to choose them wisely. Here we propose a heuristic that takes the downstream usage of the model

Algorithm 5 G-VUZero

```
Input Initial parameters $\theta$
repeat
    Collect a $K$-step trajectory $s_t, a_t, r_{t+1}, s_{t+1}, \ldots, s_{t+K}$
    Compute the MuZero loss $\mathcal{L}^{\text{MZ}}$ on $a_{t:t+K-1}$ by Eq. 7.3
    Unroll the model for one step by $a_t$ and obtain $s_t^1$
    Run MCTS from $s_t^1$ for $M$ expansions and store the queried action sequences $\tau^1, \ldots, \tau^M$
    Compute the G-VU loss $\mathcal{L}^{\text{G-VU}} = \frac{1}{M}\sum_{i=1}^{M} \mathcal{L}^{\text{G-VU}}(\theta|\overline{a_t\tau^i})$ by Eq. 7.9
    Compute the total loss $\mathcal{L} = \mathcal{L}^{\text{MZ}} + \mathcal{L}^{\text{G-VU}}$
    Update the parameters $\theta$ by SGD to minimize the total loss $\mathcal{L}$
until done
```

into account. In MuZero, the model is used by a MCTS planner to expand a search tree. Each node in the search tree corresponds to a unique action sequence that the model is queried on. Our heuristic is to update the model on the action sequences that are queried by the MCTS planner. Intuitively, the predictions along these action sequences are more important because their accuracy influences the outcome of the planner, whereas the predictions along other action sequences are less important because they do not impact the planner as much. Therefore, we want to prioritize updating the model on these important action sequences over the less important ones.

Concretely, to compute the G-VU loss, we first unroll the model from s_t for one step along a_t to reach s_t^1. Then we run MCTS for M expansions and store the corresponding action sequences $\{\tau^1, \ldots, \tau^M\}$. The length of τ^i is denoted by $k(i)$. Instead of choosing a fixed prefix length L, we choose an adaptive prefix length $l(i)$ for each τ^i as the longest common prefix between $\overline{a_t\tau^i}$ and $a_{t:t+K-1}$. Here $\overline{a_t\tau^i}$ denotes the concatenation of a_t and τ^i. Then the G-VU loss along $\overline{a_t\tau^i}$ is computed as:

$$\mathcal{L}^{\text{G-VU}}(\theta|\overline{a_t\tau^i}) = \mathcal{L}^r(\hat{r}_t^{k(i)}, \hat{r}_{t+k(i)}^{k(i)-l(i)}) + \mathcal{L}^v(\hat{v}_t^{k(i)}, \hat{v}_{t+k(i)}^{k(i)-l(i)}) + \mathcal{L}^\pi(\pi_t^{k(i)}, \pi_{t+k(i)}^{k(i)-l(i)}). \tag{7.9}$$

The overall G-VU loss is the average over all τ^i. Note that there are two differences between Eq. 7.8 and Eq. 7.9. First, we add a policy loss term to accommodate the policy component of MuZero. Second, instead of accumulating the losses along the action sequence, we only compute the loss on the final state of the model unrolling. This is to adapt to the tree structure of the MCTS search tree so that we do not accumulate the loss on the same node multiple times.

Overall, the model is updated by stochastic gradient decent to minimize the sum of the MuZero loss and the G-VU loss. Algorithm 5 provides an overview of the model learning algorithm. We name this new agent G-VUZero.

7.3.5 VE Models as Sequence-value functions

Here we provide a new interpretation of VE models as recurrent implementations of *sequence-value functions*. This interpretation provides a different perspective for understanding D-VU and G-VU.

Similar to the commonly used state-value function and action-value function, we can define *sequence-value functions*. For any policy π, its sequence-value function is defined as

$$v^k(s, a_{0:k-1}) = \mathbb{E}_\pi \big[\sum_{i=0}^{\infty} \gamma^i R_{t+i+1} | S_t = s, A_t = a_0, \ldots, A_{t+k-1} = a_{k-1} \big]. \tag{7.10}$$

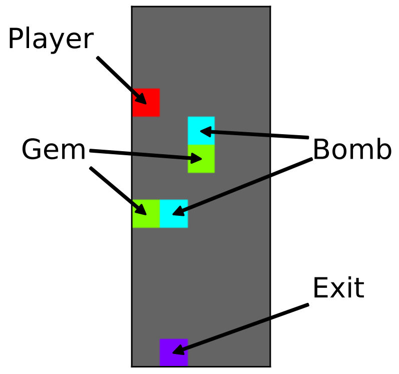
The superscript $\cdot^k$ indicates the length of the action sequence. Note that v^0 is the state-value function and v^1 is the action-value function.

A VE model can be interpreted as a recurrent implementation of sequence-value functions. The VE model shares the same inputs as the sequence-value function. It takes the state input s as the initial hidden state and unrolls over the action sequence input $a_{0:k-1}$ one step at a time. The value prediction of the final state represents the sequence value.

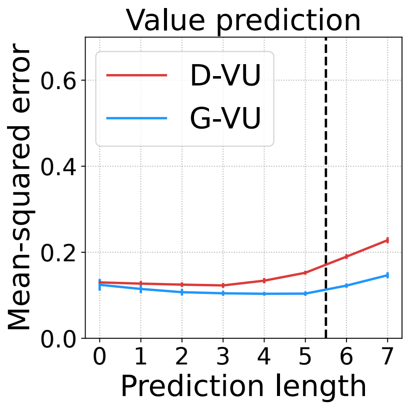
Based on this correspondence, both D-VU and G-VU can be viewed as learning a set of sequence-value functions via bootstrapping. The difference is in the bootstrapping targets. To update $\hat{v}^k(s_t, a_{0:k-1})$, D-VU bootstraps from $\hat{v}^0(s_{t+k})$ where s_{t+k} is the state the agent arrives at after executing $a_{0:k-1}$ from s_t. In contrast, G-VU bootstraps from $\hat{v}^{k-L}(s_{t+L}, a_{L:k-1})$ where s_{t+L} is the state the agent arrives at after executing the first L actions, $a_{0:L-1}$, from s_t. Note that the action sequence suffix $a_{L:k-1}$ is not executed in the environment.

7.4 Experiments

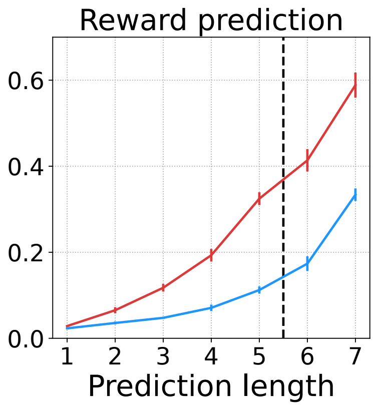
We present two sets of experiments. The first set is in the prediction setting where the task is to learn a VE model that predicts the rewards and the value of a fixed policy. We use a grid world domain called `Canal` for this experiment. The main goal of this set of experiments is to demonstrate that G-VU is more data efficient than D-VU because it can provide additional learning signals to the model on the same real trajectory. The second set of experiments evaluates G-VUZero in two planning-focused environments, Sokoban and Minipacman [Racanière et al., 2017, Guez et al., 2019]. We compare G-VUZero to MuZero and self-consistent MuZero (MuZero-SC) [Farquhar et al., 2021] to show that G-VU can further benefit the downstream planning and improve the overall control performance. We also compare G-VUZero to a variant where the action sequence suffices b are randomly sampled rather than being guided by MCTS to show that it is important to choose the action sequence suffix b properly.



(a)



(b)



(c)

Figure 7.2: The prediction experiment. (a) A snapshot of the `Canal` environment. (b) and (c) Mean-squared errors of value prediction and reward prediction on the test set. The x-axis denotes the length of the input action sequences to the model which we term "prediction length". The y-axis denotes the mean-squared errors of the predictions. Each curve is an average of 10 independent runs with different random seeds. Standard errors are marked by the short bars. The dashed vertical line separates the prediction lengths that the model was updated on during training and the longer prediction lengths that the model was never updated on.

7.4.1 Prediction

We first evaluate G-VU in a prediction setting. We designed a simple grid world environment called `Canal`. Figure 7.2a provides a snapshot of the environment. There are four different kinds of entities in this environment: the agent, two gems, two bombs, and an exit. The initial locations for the agent, the gems, and the bombs are randomly sampled at the beginning of each episode. The column of the exit is also randomly sampled but it is always on the bottom row. The agent moves down a row at every time step. There are 3 actions, one keeps the agent in the same column, a second moves to the left column, and the third moves to the right column respectively. The agent cannot move beyond the boundary of the screen and it will remain in the same column if it attempts to do so. The agent gets a reward of 5 if it collects a gem and a reward of -5 if the collects a bomb. An episode terminates when the agent hits the bottom row. It receives the reward of 1 if it hits the exit or a reward of -1 otherwise.

The task is to learn a model that predicts the reward and the value of a random policy upon executing an action sequence from a state. During training we generate trajectories by choosing actions uniformly and learn a model by either D-VU or G-VU. After training, the model is evaluated on a set of 10000 pre-sampled trajectories. The ground truth value for each state is solved by solving the Bellman equation system analytically [Sutton and Barto, 2018].

Neural network architecture. The model consists of three functions: a representation function, a transition function, and a prediction function. All of them are parameterized by neural networks. The representation function consists of two convolutional layers and a fully-connected layer. The transition function consists of two fully-connected layers. The input state and input action are concatenated before being fed into the neural network. A skip connection adds the input state to the output of the transition function. The prediction function consists of one hidden fully-connected layer and one output layer. All convolutional layers contain 32 channels with 3×3 kernels. All fully-connected layers contain 512 units. ReLU is applied after every hidden layer.

Hyperparameters. The discount factor was set to 0.97 through the experiment. The value function was updated by 1-step TD. We used a batch of 32 trajectories per update. We used the Adam optimizer [Kingma and Ba, 2015] and searched the learning rate in $\{0.001, 0.0003, 0.0001, 0.00003\}$ and selected 0.0001 as it performed the best for both D-VU and G-VU. We set $K = 5$ for both D-VU and G-VU and $L = 1$ for G-VU. We use a sliding window on the interaction stream to sample the training trajectories so the total number of parameter updates are the same for D-VU and G-VU.

Results. We train each model for 10 million environment steps and report the evaluation errors of the final model after training. The value prediction errors are shown in Figure 7.2b and the reward prediction errors are shown in Figure 7.2c. In both figures, the x-axis denotes the length of the input action sequences to the model which we term "prediction length". The y-axis denotes the mean-squared errors of the predictions. We aggregate the prediction errors by the prediction lengths. Each curve is an average of 10 independent runs with different random seeds. Standard errors are marked by the short bars. G-VU (the blue curve) consistently achieves lower prediction errors than D-VU (the red curve) on all prediction lengths. This shows that G-VU can indeed improve the sample efficiency over D-VU by providing richer learning signals. Specifically, D-VU only updates K predictions per real trajectory whereas G-VU updates $|A|^{K-1}$ predictions per trajectory. Another interesting observation is that G-VU performs better than D-VU even when the prediction lengths are greater than $K = 5$, as shown by the errors on the right of the dashed vertical line. In other words, G-VU generalizes better than D-VU to action sequences that are longer than what the model is trained on. We hypothesis that G-VU serves as a regularization by enforcing consistency between predictions at consecutive steps thus yields better generalization.

7.4.2 Control

In the second set of experiments, we evaluate G-VUZero in two planning-focused environments: Sokoban and Minipacman [Racanière et al., 2017, Guez et al., 2019]. We choose these two environments because deep lookahead search is often required to achieve strong performance in these environments. Thus they are suitable for demonstrating the benefits of G-VU on planning. We compare G-VUZero to two baseline. The first one is MuZero. The second one is MuZero-SC which augments MuZero by an additional loss that enforces joint self-consistency between the model and the value function [Farquhar et al., 2021]. To evaluate the heuristic of guiding G-VU with MCTS, we also include a variant of G-VUZero where the action sequence suffices b are chosen by random sampling instead of MCTS. We denote this variant as G-VUZero-RS.

Environment specifications. Sokoban is a procedurally generated environment. Instead of generating the game instances on-the-fly, we sampled a random configuration from a pre-generated set for each episode. We used the "unfiltered" set from [Guez et al., 2019]. We refer the readers to the original articles for the full details of the environment [Racanière et al., 2017, Guez et al., 2019]. The only difference is that we did not apply the penalty at every step because we found MuZero worked slightly better without the penalty. Minipacman is grid world that mimics the video game MsPacman. We adopted the implementation from [Guez et al., 2019] where the progress bar for eating a power pill was replaced by the color change of the ghosts. We refer the readers to the original articles for the full details of both environments [Racanière et al., 2017, Guez et al., 2019].

Neural network architectures. G-VUZero, MuZero, and MuZero-SC share the same neural network architecture and only differ in how the model is updated. For Sokoban, the representation function consists of two convolutional layers with kernel size 8 and 4 and strides 4 and 2 respectively and two residual blocks [He et al., 2016]. For the transition function, we applied a pool-and-inject layer [Racanière et al., 2017] to the input abstract state to help capture global information. The action is first encoded as a one-hot vector and then broadcast to match the spatial dimensions of the abstract states. We concatenated the output of the pool-and-inject layer and the action encoding and fed them into a convolutional layer. We applied a skip connection and added the input abstract to the output of this convolutional layer and feeding it into a residual block. The prediction function consists of three separate networks with identical architectures for reward, value, and policy respectively. Each network consists of a convolutional layer, a fully-connected layer with 256 units, and an output layer. All convolutional layers used 64 3×3 kernels with stride 1 if not otherwise stated. We applied layer normalization [Ba et al., 2016] and ReLU activation after every hidden layer. The agent architecture for Minipacman is mostly the same as for Sokoban, except that the first two convolutional layers of the representation function are replaced by a single convolutional layer with 3×3 kernels and stride 1.

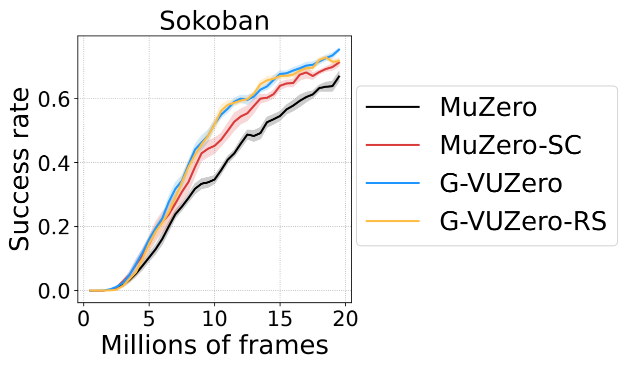


Figure 7.3: Learning curves for the first 20 million frames of training in Sokoban. The x-axis denotes the number of environment steps during training. The y-axis denotes the success rate on the test set. Each curve shows the average success rate over 10 independent trials with different random seeds. The shaded area shows the standard error.

Table 7.1: Success rates on the test set for different methods in Sokoban. We evaluated each agent at 10, 20, 50, and 100 million frames during training. The corresponding success rates are shown in different columns. Each entry shows the mean success rate across 10 independent trials with different random seeds. The number in the parenthesis is the standard error. Best performance at each evaluation point is highlighted in **bold**.

	@10M	@20M	@50M	@100M
MuZero	34.1(1.6)	68.5(1.5)	84.4(0.6)	87.5(0.8)
MuZero-SC	44.6(2.8)	72.6(1.1)	85.2(0.6)	87.5(0.6)
G-VUZero	**50.0**(1.5)	**77.0**(0.5)	**88.9**(0.6)	**89.3**(0.4)
G-VUZero-RS	49.7(1.8)	73.6(0.8)	86.4(0.7)	88.3(0.4)

Hyperparameters. Following Guez et al. [2019], we used a discount factor of 0.97 for Sokoban. K was set to 5. The value function was updated by 5-step returns. We used a batch size of 256. We used the Adam optimizer [Kingma and Ba, 2015] and searched the learning rate in $\{0.003, 0.001, 0.0003, 0.0001\}$ and selected 0.001 as it worked the best for MuZero. We did not do a separate search for other methods. For planning with MCTS, we set the search budget to 15 steps. The temperature for action selection was initialized to 1 and decayed by a factory of 0.95 after 10000 parameter updates. For MuZero-SC, we searched the number of self-consistent steps in $\{3, 5, 8\}$ and selected 5. For G-VUZero and G-VUZero, we searched M in $\{3, 5, 8, 10\}$ and selected 8. The hyperparameters for Minipacman is mostly the same as for Sokoban, except the batch size was 128 instead of 256.

Results. Figure 7.3 and Table 7.1 summarize the results for Sokoban. Figure 7.3 shows the learning curves during the first 20 million frames of training. The x-axis shows the number of frames

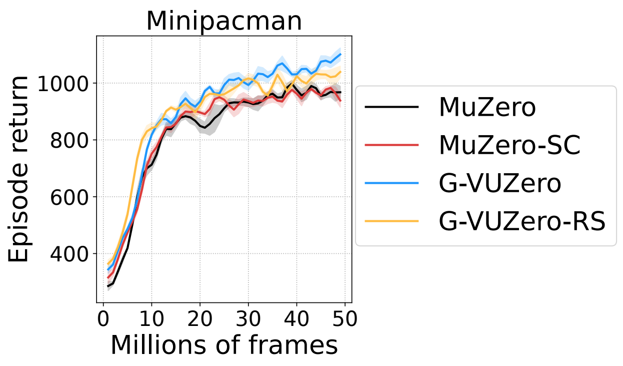


Figure 7.4: Learning curves in Minipacman. The x-axis denotes the number of environment steps during training. The y-axis denotes the episode return. Each curve shows the average success rate over 5 independent trials with different random seeds. The shaded area shows the standard error.

and the y-axis shows the success rate on the test set. Each curve shows the average success rate over 10 independent trials with different random seeds. The shaded area shows the standard error. G-VUZero (the blue curve) achieves a higher success rate than both MuZero (the black curve) and MuZero-SC (the red curve) at the early stage of training. Table 7.1 shows the agents' success rates at 10 million, 20 million, 50 million, and 100 million frames of training. The corresponding success rates are shown in different columns. Each entry shows the average success rate across 10 independent trials with different random seeds. The numbers in the parenthesis are standard errors. As highlighted by the bold numbers, G-VUZero performs better than MuZero and MuZero-SC throughout 100 million frames of training, though the differences become small as all methods converge. Figure 7.4 shows the learning curves in Minipacman. The x-axis shows the number of frames and the y-axis shows the episode return. Each curve is an average over 5 independent trials with different random seeds. The shaded area shows the standard error. Again, G-VUZero (the blue curve) performs better than both MuZero (the black curve) and MuZero-SC (the red curve). These results show that the benefit provided by G-VU in model learning also carries over to the downstream planning and improve the overall control performance of the agent. G-VUZero-RS (the yellow curve) performs slightly worse than G-VUZero in Sokoban (Figure 7.3) but is clearly subpar in Minipacman (Figure 7.4). This performance gap is due to the difference in choosing the action sequence suffix b to apply G-VU on. By guiding G-VU with MCTS, G-VUZero can update the model on the action sequences that can potentially impact the planner. Whereas G-VUZero-RS chooses the action sequences randomly thus rendering the G-VU updates less effective. This result suggests that G-VU needs to be guided by its downstream usage in order to maximize the benefits of flexibly updating the model on action sequences that do not coincide with the real trajectories.

7.5 Conclusion and Future Directions

In this chapter we proposed the general value-equivalent update (G-VU) for learning VE models. G-VU is a generalization of the direct value-equivalent update (D-VU) used by existing methods. G-VU is more flexible than D-VU in that it can update the model on action sequences that are not experienced in the environment. Our empirical results showed that G-VU is more data efficient at learning VE models than D-VU. We combined G-VU with the state-of-the-art MuZero agent and showed that the new agent, G-VUZero, outperformed MuZero in Sokoban and Minipacman. We also proposed a heuristic that uses the downstream planner to guide G-VU for choosing which action sequences to update the model on. We evaluated this heuristic in Sokoban and Minipacman and showed that it performed better than a simple random sampling baseline. Although more empirical work is needed, our results suggest that G-VU is an effective VE-model learning method and can potentially benefit VE-model-based RL agents, including but not limited to MuZero.

One future direction is to apply G-VU to continuous action spaces. The main benefit of G-VU is that it can update the model on more action sequences than D-VU with the same real trajectory. In continuous control, there are effectively infinite action sequences for any length, which provides more room for G-VU to improve. Meanwhile, it may be more important to choose the action sequences wisely in the continuous control setting because randomly sampled action sequences may have very little overlap with the ones queried by the planner. Another future direction is to extend G-VU to other types of models. Although we derived G-VU from the value-equivalence principle, the idea of bootstrapping from a later model prediction is more general and not limited to VE models. It will be interesting to apply the idea of G-VU on other types of models, especially models learned by self-supervised methods.

CHAPTER 8

Conclusion

In this chapter, we summarize the main contributions of this thesis and discuss some future directions for DeepRL research.

8.1 Summary of Contributions

Building on the recent success of DeepRL, this thesis attempted to further advance DeepRL techniques by addressing challenges in the following directions: 1) reward design, 2) temporal credit assignment, 3) state representation learning, and 4) model learning.

In Chapter 3, we built on the Optimal Rewards Framework and derived a novel meta-learning algorithm, LIRPG, for learning intrinsic rewards for policy-gradient-based agents. Our empirical study in the Atari domain and the Mujoco continuous control domain showed that LIRPG could learn useful intrinsic reward functions that improved the performance of the policy learner. In addition, we also showed that the learned intrinsic reward function performed better than two handcrafted heuristic intrinsic reward functions.

In Chapter 4, we focused on understanding what can be captured by the learned intrinsic reward functions. To investigate this, we proposed a scalable metagradient framework for learning useful intrinsic reward functions across multiple lifetimes of experience. Through a set of proof-of-concept experiments, we showed that the learned intrinsic reward functions could capture knowledge about long-term exploration and exploitation. Furthermore, we showed that the learned reward functions could generalise to different RL algorithms and to changes in the environment dynamics because they captured “what to do” instead of “how to do.”

In Chapter 5, in order to overcome the limitations of existing temporal credit assignment mechanisms, we explored heuristics based on more general pairwise weightings that are functions of the state in which the action was taken, the state at the time of the reward, as well as the time interval between the two. We developed a metagradient algorithm for learning these weight functions during the usual policy optimization procedure. Our empirical work showed that it is often possible to learn these pairwise weight functions during policy learning to achieve better performance than competing approaches.

In Chapter 6, we investigated using random prediction tasks as auxiliary tasks for state representation learning. We found that random general value functions (GVFs), i.e., deep action-conditional predictions, form good auxiliary tasks for RL agents. Our empirical study on the Atari benchmark and the DeepMind Lab benchmarks showed that learning state representations solely via auxiliary prediction tasks defined by random GVFs outperformed the end-to-end trained A2C baseline. Random GVFs also outperformed other handcrafted auxiliary tasks when being used as part of a combined loss function with the main RL loss.

In Chapter 7, we presented generalized value-equivalence updates (G-VU) for learning value-equivalent models. G-VU is more flexible than existing method because it can update the model on action sequences that are not experienced in the environment. Our experiments showed that G-VU was more sample efficient than existing methods in a prediction setting. When combined with MuZero, the resulting agent G-VUZero demonstrated better the control performance than MuZero in two planning-focused environments Sokoban and Minipacman. Moreover, we explored applying G-VU on action sequences queried by the downstream MCTS planner and showed that this version of guided G-VU performed better than applying G-VU on randomly sampled action sequences. It is worth noting that learning a VE model itself serves as a good auxiliary task for state representation learning [Hessel et al., 2021]. Similar to the random GVFs in Chapter 6, the VE model predictions are both deep and action-conditional. In contrast to the random GVFs, a VE model predicts rewards and values rather than unsupervised random features of the observations. Moreover, G-VU is similar to the TD update in Chapter 6 in that they both update a prediction by bootstrapping from other predictions at later steps. This connection suggests a future direction of combining unsupervised off-policy prediction learning with model learning to improve both the state representation and the model.

8.2 Future Directions

Learning from signals beyond the rewards Most existing DeepRL agents still learn solely from rewards. Since rewards are usually sparse, learning only happens occasionally and is inefficient. In Chapter 6, we demonstrated that random GVFs can form useful auxiliary tasks that provide additional learning signals to an agent and help it learn better state representations. However, random GVFs are hardly a principled solution to this problem. Due to their task-agnostic nature, they may fail to capture important aspects of the environment and yield poor state representations. One solution to this problem is to let the agents *discover* useful prediction tasks for themselves. Veeriah et al. [2019] proposed a metagradient algorithm similar to LIRPG in Chapter 3 for discovering useful auxiliary prediction tasks. However, the proposed method only demonstrated marginal performance improvement in their empirical study. How to discover useful auxiliary tasks effectively remains an open question. Another promising approach is to take advantage of the recent advances in self-supervised learning. Recent work has shown that it is possible to learn good feature representations for images [Oord et al., 2018, Chen et al., 2020, He et al., 2020, Chen and He, 2021] or languages [Devlin et al., 2018, Brown et al., 2020] without any label. Adapting these techniques to DeepRL may enable the agent to learn good state representations without reward signals. Some recently work already demonstrated promise in this direction [Laskin et al., 2020, Lee et al., 2020] but further research is still needed.

Temporal abstractions of behaviors Temporal abstraction of behaviors is still largely missing in DeepRL agents. The option framework [Sutton et al., 1999] provides a theoretical framework for studying temporal abstractions in RL. However, we have only seen limited success of combining options with DeepRL so far. The biggest challenge is how to discover useful options. It is unclear under what pressure useful abstractions of temporally extended behaviors will emerge. Some early attempts used the same RL objective of maximizing cumulative rewards in a single-task setting [Bacon et al., 2017]. But the discovered options often degraded to either primitive actions or a policy that solves the task completely. Other works avoided this problem by setting a fixed temporal resolution [Vezhnevets et al., 2017, Nachum et al., 2018]. But this approach limits the expressiveness of the options and cannot scale well as the task horizons continue to increase. Therefore, how to discover useful temporal abstractions of behaviors remains an open question to the RL community.

Interfaces for goal specification In Chapter 3 we studied the reward design problem. Part of the reason why reward design is challenging is that humans do not communicate their goals via reward functions. The reward design process can be interpreted as translating human desire to a form of goal specification that RL agents can understand. This translation is often challenging and error-prone. In this thesis we made an attempt to address this problem and proposed a method to mitigate this gap. But it still requires a reward function, though not necessarily an easy-to-optimize one, to be given by the agent designer. It is still unclear what is the optimal interface for communicating goals from human agent designers to RL agents. Humans are most comfortable with communicating via languages. Therefore, we may prefer an interface that allows human to specify goals via natural language. Recently, large language models (LLMs) like BERT [Devlin et al., 2018] and GPT [Brown et al., 2020] show promise in comprehensive language understanding and compositional generalization. Inspired by these recent success, one potential direction is to explore how to design a natural-language-to-reward interface by using these LLMs.

BIBLIOGRAPHY

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. {TensorFlow}: A system for {Large-Scale} machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 265–283, 2016.

Dario Amodei, Chris Olah, Jacob Steinhardt, Paul Christiano, John Schulman, and Dan Mané. Concrete problems in ai safety. *arXiv preprint arXiv:1606.06565*, 2016.

Ankesh Anand, Evan Racah, Sherjil Ozair, Yoshua Bengio, Marc-Alexandre Côté, and R. Devon Hjelm. Unsupervised state representation learning in atari. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 8766–8779, 2019.

Marcin Andrychowicz, Misha Denil, Sergio Gomez, Matthew W Hoffman, David Pfau, Tom Schaul, Brendan Shillingford, and Nando De Freitas. Learning to learn by gradient descent by gradient descent. In *Advances in Neural Information Processing Systems*, pages 3981–3989, 2016.

OpenAI: Marcin Andrychowicz, Bowen Baker, Maciek Chociej, Rafal Jozefowicz, Bob McGrew, Jakub Pachocki, Arthur Petron, Matthias Plappert, Glenn Powell, Alex Ray, et al. Learning dexterous in-hand manipulation. *The International Journal of Robotics Research*, 39(1):3–20, 2020.

Jose A Arjona-Medina, Michael Gillhofer, Michael Widrich, Thomas Unterthiner, Johannes Brandstetter, and Sepp Hochreiter. Rudder: Return decomposition for delayed rewards. In *Advances in Neural Information Processing Systems*, pages 13544–13555, 2019.

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

Pierre-Luc Bacon, Jean Harb, and Doina Precup. The option-critic architecture. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 31, 2017.

Dzmitry Bahdanau, Kyung Hyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In *3rd International Conference on Learning Representations, ICLR 2015*, 2015.

Dzmitry Bahdanau, Felix Hill, Jan Leike, Edward Hughes, Arian Hosseini, Pushmeet Kohli, and Edward Grefenstette. Learning to understand goal specifications by modelling reward. In *International Conference on Learning Representations*, 2019.

Charles Beattie, Joel Z. Leibo, Denis Teplyashin, Tom Ward, Marcus Wainwright, Heinrich Küttler, Andrew Lefrancq, Simon Green, Víctor Valdés, Amir Sadik, Julian Schrittwieser, Keith Anderson, Sarah York, Max Cant, Adam Cain, Adrian Bolton, Stephen Gaffney, Helen King, Demis Hassabis, Shane Legg, and Stig Petersen. Deepmind lab. *CoRR*, abs/1612.03801, 2016. URL `http://arxiv.org/abs/1612.03801`.

Sarah Bechtle, Artem Molchanov, Yevgen Chebotar, Edward Grefenstette, Ludovic Righetti, Gaurav Sukhatme, and Franziska Meier. Meta-learning via learned loss. *arXiv preprint arXiv:1906.05374*, 2019.

Marc Bellemare, Sriram Srinivasan, Georg Ostrovski, Tom Schaul, David Saxton, and Remi Munos. Unifying count-based exploration and intrinsic motivation. In *Advances in Neural Information Processing Systems*, pages 1471–1479, 2016.

Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

Marc G. Bellemare, Will Dabney, Robert Dadashi, Adrien Ali Taïga, Pablo Samuel Castro, Nicolas Le Roux, Dale Schuurmans, Tor Lattimore, and Clare Lyle. A geometric perspective on optimal representations for reinforcement learning. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 4360–4371, 2019.

Richard Bellman. Dynamic programming. *Science*, 153(3731):34–37, 1966.

Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Debiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, et al. Dota 2 with large scale deep reinforcement learning. *arXiv preprint arXiv:1912.06680*, 2019.

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

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. A simple framework for contrastive learning of visual representations. In *International conference on machine learning*, pages 1597–1607. PMLR, 2020.

Xinlei Chen and Kaiming He. Exploring simple siamese representation learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 15750–15758, 2021.

Kyunghyun Cho, Bart van Merrienboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder-decoder approaches. In Dekai Wu, Marine Carpuat, Xavier Carreras, and Eva Maria Vecchi, editors, *Proceedings of SSST@EMNLP 2014, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation, Doha, Qatar, 25 October 2014*, pages 103–111. Association for Computational Linguistics, 2014. doi: 10.3115/v1/W14-4012.

Junyoung Chung, Çaglar Gülçehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *CoRR*, abs/1412.3555, 2014.

Jack Clark and Dario Amodei. Faulty reward functions in the wild. *CoRR*, 2016. URL `https://blog.openai.com/`.

Ignasi Clavera, Anusha Nagabandi, Simin Liu, Ronald S. Fearing, Pieter Abbeel, Sergey Levine, and Chelsea Finn. Learning to adapt in dynamic, real-world environments through meta-reinforcement learning. In *International Conference on Learning Representations*, 2019. URL `https://openreview.net/forum?id=HyztsoC5Y7`.

Jonathan D Cohen, Samuel M McClure, and Angela J Yu. Should i stay or should i go? how the human brain manages the trade-off between exploitation and exploration. *Philosophical Transactions of the Royal Society B: Biological Sciences*, 362(1481):933–942, 2007.

Will Dabney, André Barreto, Mark Rowland, Robert Dadashi, John Quan, Marc G. Bellemare, and David Silver. The value-improvement path: Towards better representations for reinforcement learning. *CoRR*, abs/2006.02243, 2020.

Jonas Degrave, Federico Felici, Jonas Buchli, Michael Neunert, Brendan Tracey, Francesco Carpanese, Timo Ewalds, Roland Hafner, Abbas Abdolmaleki, Diego de las Casas, Craig Donner, Leslie Fritz, Cristian Galperti, Andrea Huber, James Keeling, Maria Tsimpoukelli, Jackie Kay, Antoine Merle, Jean-Marc Moret, Seb Noury, Federico Pesamosca, David Pfau, Olivier Sauter, Cristian Sommariva, Stefano Coda, Basil Duval, Ambrogio Fasoli, Pushmeet Kohli, Koray Kavukcuoglu, Demis Hassabis, and Martin Riedmiller. Magnetic control of tokamak plasmas through deep reinforcement learning. *Nature*, 602(7897):414–419, Feb 2022. ISSN 1476-4687. doi: 10.1038/s41586-021-04301-9. URL `https://doi.org/10.1038/s41586-021-04301-9`.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, and Yuhuai Wu. Openai baselines. `https://github.com/openai/baselines`, 2017.

Yan Duan, Xi Chen, Rein Houthooft, John Schulman, and Pieter Abbeel. Benchmarking deep reinforcement learning for continuous control. In *International Conference on Machine Learning*, pages 1329–1338, 2016a.

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

Yan Duan, Marcin Andrychowicz, Bradly Stadie, OpenAI Jonathan Ho, Jonas Schneider, Ilya Sutskever, Pieter Abbeel, and Wojciech Zaremba. One-shot imitation learning. In *Advances in neural information processing systems*, pages 1087–1098, 2017.

Rachit Dubey and Thomas L Griffiths. Reconciling novelty and complexity through a rational analysis of curiosity. *Psychological Review*, 2019.

Amir-massoud Farahmand, Andre Barreto, and Daniel Nikovski. Value-aware loss function for model-based reinforcement learning. In *Artificial Intelligence and Statistics*, pages 1486–1494. PMLR, 2017.

Greg Farquhar, Kate Baumli, Zita Marinho, Angelos Filos, Matteo Hessel, Hado P van Hasselt, and David Silver. Self-consistent models and values. *Advances in Neural Information Processing Systems*, 34, 2021.

Gregory Farquhar, Tim Rocktäschel, Maximilian Igl, and Shimon Whiteson. Treeqn and atreec: Differentiable tree-structured models for deep reinforcement learning. In *International Conference on Learning Representations*, 2018.

William Fedus, Carles Gelada, Yoshua Bengio, Marc G. Bellemare, and Hugo Larochelle. Hyperbolic discounting and learning over multiple horizons. *CoRR*, abs/1902.06865, 2019.

Angelos Filos, Eszter Vértes, Zita Marinho, Gregory Farquhar, Diana Borsa, Abram Friesen, Feryal Behbahani, Tom Schaul, André Barreto, and Simon Osindero. Model-value inconsistency as a signal for epistemic uncertainty. *arXiv preprint arXiv:2112.04153*, 2021.

Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1126–1135. JMLR. org, 2017a.

Chelsea Finn, Tianhe Yu, Tianhao Zhang, Pieter Abbeel, and Sergey Levine. One-shot visual imitation learning via meta-learning. In *Conference on Robot Learning*, pages 357–368, 2017b.

John Gittins. A dynamic allocation index for the sequential design of experiments. *Progress in statistics*, pages 241–266, 1974.

John C Gittins. Bandit processes and dynamic allocation indices. *Journal of the Royal Statistical Society: Series B (Methodological)*, 41(2):148–164, 1979.

Goren Gordon and Ehud Ahissar. Reinforcement active learning hierarchical loops. In *The 2011 International Joint Conference on Neural Networks*, pages 3008–3015. IEEE, 2011.

Anirudh Goyal, Riashat Islam, DJ Strouse, Zafarali Ahmed, Hugo Larochelle, Matthew Botvinick, Yoshua Bengio, and Sergey Levine. Infobot: Transfer and exploration via the information bottleneck. In *International Conference on Learning Representations*, 2018.

Karol Gregor, Danilo Jimenez Rezende, Frederic Besse, Yan Wu, Hamza Merzic, and Aäron van den Oord. Shaping belief states with generative environment models for RL. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 13475–13487, 2019.

Christopher Grimm, André Barreto, Satinder Singh, and David Silver. The value equivalence principle for model-based reinforcement learning. *Advances in Neural Information Processing Systems*, 33:5541–5552, 2020.

Christopher Grimm, André Barreto, Greg Farquhar, David Silver, and Satinder Singh. Proper value equivalence. *Advances in Neural Information Processing Systems*, 34, 2021.

Arthur Guez, Mehdi Mirza, Karol Gregor, Rishabh Kabra, Sébastien Racanière, Théophane Weber, David Raposo, Adam Santoro, Laurent Orseau, Tom Eccles, et al. An investigation of model-free planning. In *International Conference on Machine Learning*, pages 2464–2473. PMLR, 2019.

Xiaoxiao Guo, Satinder Singh, Richard Lewis, and Honglak Lee. Deep learning for reward design to improve monte carlo tree search in atari games. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, pages 1519–1525. AAAI Press, 2016.

Zhaohan Daniel Guo, Mohammad Gheshlaghi Azar, Bilal Piot, Bernardo A. Pires, Toby Pohlen, and Rémi Munos. Neural predictive belief representations. *CoRR*, abs/1811.06407, 2018.

Zhaohan Daniel Guo, Bernardo Ávila Pires, Bilal Piot, Jean-Bastien Grill, Florent Altché, Rémi Munos, and Mohammad Gheshlaghi Azar. Bootstrap latent-predictive representations for multitask reinforcement learning. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pages 3875–3886. PMLR, 2020.

David Ha and Jürgen Schmidhuber. Recurrent world models facilitate policy evolution. *Advances in neural information processing systems*, 31, 2018.

Danijar Hafner, Timothy Lillicrap, Jimmy Ba, and Mohammad Norouzi. Dream to control: Learning behaviors by latent imagination. In *International Conference on Learning Representations*, 2019a.

Danijar Hafner, Timothy Lillicrap, Ian Fischer, Ruben Villegas, David Ha, Honglak Lee, and James Davidson. Learning latent dynamics for planning from pixels. In *International conference on machine learning*, pages 2555–2565. PMLR, 2019b.

Danijar Hafner, Timothy P. Lillicrap, Mohammad Norouzi, and Jimmy Ba. Mastering atari with discrete world models. *CoRR*, abs/2010.02193, 2020.

Anna Harutyunyan, Sam Devlin, Peter Vrancx, and Ann Nowe. Expressing arbitrary reward functions as potential-based advice. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, pages 2652–2658. AAAI Press, 2015.

Anna Harutyunyan, Will Dabney, Thomas Mesnard, Mohammad Gheshlaghi Azar, Bilal Piot, Nicolas Heess, Hado P van Hasselt, Gregory Wayne, Satinder Singh, Doina Precup, et al. Hindsight credit assignment. In *Advances in neural information processing systems*, pages 12467–12476, 2019.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

Kaiming He, Haoqi Fan, Yuxin Wu, Saining Xie, and Ross B. Girshick. Momentum contrast for unsupervised visual representation learning. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2020, Seattle, WA, USA, June 13-19, 2020*, pages 9726–9735. IEEE, 2020. doi: 10.1109/CVPR42600.2020.00975.

Matteo Hessel, Ivo Danihelka, Fabio Viola, Arthur Guez, Simon Schmitt, Laurent Sifre, Theophane Weber, David Silver, and Hado Van Hasselt. Muesli: Combining improvements in policy optimization. In *International Conference on Machine Learning*, pages 4214–4226. PMLR, 2021.

Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal processing magazine*, 29(6):82–97, 2012.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8): 1735–1780, 1997.

Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Mohammadamin Barekatain, Simon Schmitt, and David Silver. Learning and planning in complex action spaces. In *International Conference on Machine Learning*, pages 4476–4486. PMLR, 2021.

Chia-Chun Hung, Timothy Lillicrap, Josh Abramson, Yan Wu, Mehdi Mirza, Federico Carnevale, Arun Ahuja, and Greg Wayne. Optimizing agent behavior over long time scales by transporting value. *Nature communications*, 10(1):1–12, 2019.

Laurent Itti and Pierre F Baldi. Bayesian surprise attracts human attention. In *Advances in neural information processing systems*, pages 547–554, 2006.

Max Jaderberg, Volodymyr Mnih, Wojciech Marian Czarnecki, Tom Schaul, Joel Z. Leibo, David Silver, and Koray Kavukcuoglu. Reinforcement learning with unsupervised auxiliary tasks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

Bilal Kartal, Pablo Hernandez-Leal, and Matthew E. Taylor. Terminal prediction as an auxiliary task for deep reinforcement learning. In Gillian Smith and Levi Lelis, editors, *Proceedings of the Fifteenth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, AIIDE 2019, October 8-12, 2019, Atlanta, Georgia, USA*, pages 38–44. AAAI Press, 2019.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

Louis Kirsch, Sjoerd van Steenkiste, and Juergen Schmidhuber. Improving generalization in meta reinforcement learning using learned objectives. In *International Conference on Learning Representations*, 2019.

Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer, 2006.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.

Michael Laskin, Aravind Srinivas, and Pieter Abbeel. CURL: contrastive unsupervised representations for reinforcement learning. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pages 5639–5650. PMLR, 2020.

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

Kuang-Huei Lee, Ian Fischer, Anthony Liu, Yijie Guo, Honglak Lee, John Canny, and Sergio Guadarrama. Predictive information accelerates learning in RL. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-end training of deep visuomotor policies. *The Journal of Machine Learning Research*, 17(1):1334–1373, 2016.

Cam Linke, Nadia M Ady, Martha White, Thomas Degris, and Adam White. Adapting behaviour via intrinsic reward: A survey and empirical study. *arXiv preprint arXiv:1906.07865*, 2019.

Michael L. Littman, Richard S. Sutton, and Satinder Singh. Predictive representations of state. In Thomas G. Dietterich, Suzanna Becker, and Zoubin Ghahramani, editors, *Advances in Neural Information Processing Systems 14 [Neural Information Processing Systems: Natural and Synthetic, NIPS 2001, December 3-8, 2001, Vancouver, British Columbia, Canada]*, pages 1555–1561. MIT Press, 2001.

Clare Lyle, Mark Rowland, Georg Ostrovski, and Will Dabney. On the effect of auxiliary tasks on representation dynamics. In Arindam Banerjee and Kenji Fukumizu, editors, *The 24th International Conference on Artificial Intelligence and Statistics, AISTATS 2021, April 13-15, 2021, Virtual Event*, volume 130 of *Proceedings of Machine Learning Research*, pages 1–9. PMLR, 2021. URL http://proceedings.mlr.press/v130/lyle21a.html.

Bogdan Mazoure, Remi Tachet des Combes, Thang Doan, Philip Bachman, and R. Devon Hjelm. Deep reinforcement and infomax learning. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

Thomas Mesnard, Théophane Weber, Fabio Viola, Shantanu Thakoor, Alaa Saade, Anna Harutyunyan, Will Dabney, Tom Stepleton, Nicolas Heess, Arthur Guez, et al. Counterfactual credit assignment in model-free reinforcement learning. *arXiv preprint arXiv:2011.09464*, 2020.

Luke Metz, Niru Maheswaranathan, Brian Cheung, and Jascha Sohl-Dickstein. Meta-learning update rules for unsupervised representation learning. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=HkNDsiC9KQ.

Marco Mirolli and Gianluca Baldassarre. Functions and mechanisms of intrinsic motivations. In *Intrinsically Motivated Learning in Natural and Artificial Systems*, pages 49–72. Springer, 2013.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*, pages 1928–1937, 2016.

Thomas M Moerland, Joost Broekens, and Catholijn M Jonker. Model-based reinforcement learning: A survey. *arXiv preprint arXiv:2006.16712*, 2020.

Ofir Nachum, Shixiang Shane Gu, Honglak Lee, and Sergey Levine. Data-efficient hierarchical reinforcement learning. *Advances in neural information processing systems*, 31, 2018.

Andrew Y Ng, Daishi Harada, and Stuart J Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *Proceedings of the Sixteenth International Conference on Machine Learning*, pages 278–287. Morgan Kaufmann Publishers Inc., 1999.

Alex Nichol and John Schulman. On first-order meta-learning algorithms. *arXiv preprint arXiv:1803.02999*, 2018.

Junhyuk Oh, Xiaoxiao Guo, Honglak Lee, Richard L Lewis, and Satinder Singh. Action-conditional video prediction using deep networks in atari games. *Advances in neural information processing systems*, 28, 2015.

Junhyuk Oh, Satinder Singh, and Honglak Lee. Value prediction network. *Advances in neural information processing systems*, 30, 2017.

Junhyuk Oh, Matteo Hessel, Wojciech M Czarnecki, Zhongwen Xu, Hado P van Hasselt, Satinder Singh, and David Silver. Discovering reinforcement learning algorithms. *Advances in Neural Information Processing Systems*, 33, 2020.

Aaron van den Oord, Yazhe Li, and Oriol Vinyals. Representation learning with contrastive predictive coding. *arXiv preprint arXiv:1807.03748*, 2018.

Ian Osband, Yotam Doron, Matteo Hessel, John Aslanides, Eren Sezener, Andre Saraiva, Katrina McKinney, Tor Lattimore, Csaba Szepezvari, Satinder Singh, et al. Behaviour suite for reinforcement learning. *arXiv preprint arXiv:1908.03568*, 2019.

Georg Ostrovski, Marc G Bellemare, Aäron van den Oord, and Rémi Munos. Count-based exploration with neural density models. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 2721–2730. JMLR. org, 2017.

Pierre-Yves Oudeyer and Frederic Kaplan. What is intrinsic motivation? a typology of computational approaches. *Frontiers in Neurorobotics*, 1:6, 2009.

Pierre-Yves Oudeyer, Frdric Kaplan, and Verena V Hafner. Intrinsic motivation systems for autonomous mental development. *IEEE transactions on evolutionary computation*, 11(2):265–286, 2007.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

Deepak Pathak, Pulkit Agrawal, Alexei A Efros, and Trevor Darrell. Curiosity-driven exploration by self-supervised prediction. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 2778–2787. JMLR. org, 2017.

Pascal Poupart, Nikos Vlassis, Jesse Hoey, and Kevin Regan. An analytic solution to discrete bayesian reinforcement learning. In *Proceedings of the 23rd International Conference on Machine Learning*, pages 697–704. ACM, 2006.

Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

Sébastien Racanière, Théophane Weber, David Reichert, Lars Buesing, Arthur Guez, Danilo Jimenez Rezende, Adrià Puigdomènech Badia, Oriol Vinyals, Nicolas Heess, Yujia Li, et al. Imagination-augmented agents for deep reinforcement learning. *Advances in neural information processing systems*, 30, 2017.

Janarthanan Rajendran, Richard Lewis, Vivek Veeriah, Honglak Lee, and Satinder Singh. How should an agent practice? *arXiv preprint arXiv:1912.07045*, 2019.

Aravind Rajeswaran, Kendall Lowrey, Emanuel V Todorov, and Sham M Kakade. Towards generalization and simplicity in continuous control. In *Advances in Neural Information Processing Systems*, pages 6553–6564, 2017.

Jette Randlöv and Preben Alström. Learning to drive a bicycle using reinforcement learning and shaping. In *Proceedings of the Fifteenth International Conference on Machine Learning*, pages 463–471. Morgan Kaufmann Publishers Inc., 1998.

David Raposo, Sam Ritter, Adam Santoro, Greg Wayne, Theophane Weber, Matt Botvinick, Hado van Hasselt, and Francis Song. Synthetic returns for long-term credit assignment. *arXiv preprint arXiv:2102.12425*, 2021.

Adam Santoro, Sergey Bartunov, Matthew Botvinick, Daan Wierstra, and Timothy Lillicrap. Meta-learning with memory-augmented neural networks. In *International conference on machine learning*, pages 1842–1850, 2016.

Matthew Schlegel, Andrew Patterson, Adam White, and Martha White. Discovery of predictive representations with a network of general value functions, 2018. URL `https://openreview.net/forum?id=ryZElGZ0Z`.

Matthew Schlegel, Andrew Jacobsen, Zaheer Abbas, Andrew Patterson, Adam White, and Martha White. General value function networks. *J. Artif. Intell. Res.*, 70:497–543, 2021. doi: 10.1613/jair.1.12105. URL `https://doi.org/10.1613/jair.1.12105`.

Jüergen Schmidhuber, Jieyu Zhao, and MA Wiering. Simple principles of metalearning. *Technical report IDSIA*, 69:1–23, 1996.

Jürgen Schmidhuber. Curious model-building control systems. In *Proc. international joint conference on neural networks*, pages 1458–1463, 1991a.

Jürgen Schmidhuber. A possibility for implementing curiosity and boredom in model-building neural controllers. In *Proc. of the international conference on simulation of adaptive behavior: From animals to animats*, pages 222–227, 1991b.

Jürgen Schmidhuber. Formal theory of creativity, fun, and intrinsic motivation (1990–2010). *IEEE Transactions on Autonomous Mental Development*, 2(3):230–247, 2010.

Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, et al. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, 2020.

John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015.

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

Max Schwarzer, Ankesh Anand, Rishab Goel, R Devon Hjelm, Aaron Courville, and Philip Bachman. Data-efficient reinforcement learning with self-predictive representations. *arXiv preprint arXiv:2007.05929*, 2020.

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

David Silver, Hado Hasselt, Matteo Hessel, Tom Schaul, Arthur Guez, Tim Harley, Gabriel Dulac-Arnold, David Reichert, Neil Rabinowitz, Andre Barreto, et al. The predictron: End-to-end learning and planning. In *International conference on machine learning*, pages 3191–3199. PMLR, 2017.

Satinder Singh, Michael R. James, and Matthew R. Rudary. Predictive state representations: A new theory for modeling dynamical systems. In David Maxwell Chickering and Joseph Y. Halpern, editors, *UAI '04, Proceedings of the 20th Conference in Uncertainty in Artificial Intelligence, Banff, Canada, July 7-11, 2004*, pages 512–518. AUAI Press, 2004.

Satinder Singh, Richard L Lewis, and Andrew G Barto. Where do rewards come from. In *Proceedings of the annual conference of the cognitive science society*, pages 2601–2606. Cognitive Science Society, 2009.

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, 2010.

Jonathan Sorg, Richard L Lewis, and Satinder Singh. Reward design via online gradient ascent. In *Advances in Neural Information Processing Systems*, pages 2190–2198, 2010.

Bradly Stadie, Ge Yang, Rein Houthooft, Peter Chen, Yan Duan, Yuhuai Wu, Pieter Abbeel, and Ilya Sutskever. The importance of sampling inmeta-reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 9280–9290, 2018.

Bradly C Stadie, Sergey Levine, and Pieter Abbeel. Incentivizing exploration in reinforcement learning with deep predictive models. *arXiv preprint arXiv:1507.00814*, 2015.

Adam Stooke, Kimin Lee, Pieter Abbeel, and Michael Laskin. Decoupling representation learning from reinforcement learning. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pages 9870–9879. PMLR, 2021. URL http://proceedings.mlr.press/v139/stooke21a.html.

Alexander L Strehl and Michael L Littman. An analysis of model-based interval estimation for markov decision processes. *Journal of Computer and System Sciences*, 74(8):1309–1331, 2008.

Richard S Sutton. Learning to predict by the methods of temporal differences. *Machine learning*, 3(1):9–44, 1988.

Richard S. Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proceedings of the Seventh International Conference on Machine Learning*, pages 216–224. Morgan Kaufmann, 1990.

Richard S Sutton. Dyna, an integrated architecture for learning, planning, and reacting. *ACM Sigart Bulletin*, 2(4):160–163, 1991.

Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

Richard S. Sutton and Brian Tanner. Temporal-difference networks. In *Advances in Neural Information Processing Systems 17 [Neural Information Processing Systems, NIPS 2004, December 13-18, 2004, Vancouver, British Columbia, Canada]*, pages 1377–1384, 2004.

Richard S Sutton, Doina Precup, and Satinder Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1-2):181–211, 1999.

Richard S Sutton, David A McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*, pages 1057–1063, 2000.

Richard S. Sutton, Eddie J. Rafols, and Anna Koop. Temporal abstraction in temporal-difference networks. In *Advances in Neural Information Processing Systems 18 [Neural Information Processing Systems, NIPS 2005, December 5-8, 2005, Vancouver, British Columbia, Canada]*, pages 1313–1320, 2005. URL https://proceedings.neurips.cc/paper/2005/hash/12311d05c9aa67765703984239511212-Abstract.html.

Richard S. Sutton, Joseph Modayil, Michael Delp, Thomas Degris, Patrick M. Pilarski, Adam White, and Doina Precup. Horde: a scalable real-time architecture for learning knowledge from unsupervised sensorimotor interaction. In Liz Sonenberg, Peter Stone, Kagan Tumer, and Pinar Yolum, editors, *10th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2011), Taipei, Taiwan, May 2-6, 2011, Volume 1-3*, pages 761–768. IFAAMAS, 2011.

Aviv Tamar, Yi Wu, Garrett Thomas, Sergey Levine, and Pieter Abbeel. Value iteration networks. *Advances in neural information processing systems*, 29, 2016.

Haoran Tang, Rein Houthooft, Davis Foote, Adam Stooke, OpenAI Xi Chen, Yan Duan, John Schulman, Filip DeTurck, and Pieter Abbeel. # exploration: A study of count-based exploration for deep reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 2750–2759, 2017.

William R Thompson. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, 25(3/4):285–294, 1933.

Sebastian Thrun and Lorien Pratt. Learning to learn: Introduction and overview. In *Learning to learn*, pages 3–17. Springer, 1998.

Aäron van den Oord, Yazhe Li, and Oriol Vinyals. Representation learning with contrastive predictive coding. *CoRR*, abs/1807.03748, 2018.

Hado van Hasselt, Sephora Madjiheurem, Matteo Hessel, David Silver, André Barreto, and Diana Borsa. Expected eligibility traces. *arXiv preprint arXiv:2007.01839*, 2020.

Vivek Veeriah, Matteo Hessel, Zhongwen Xu, Janarthanan Rajendran, Richard L Lewis, Junhyuk Oh, Hado P van Hasselt, David Silver, and Satinder Singh. Discovery of useful questions as auxiliary tasks. In *Advances in Neural Information Processing Systems*, pages 9306–9317, 2019.

Alexander Sasha Vezhnevets, Simon Osindero, Tom Schaul, Nicolas Heess, Max Jaderberg, David Silver, and Koray Kavukcuoglu. Feudal networks for hierarchical reinforcement learning. In *International Conference on Machine Learning*, pages 3540–3549, 2017.

Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, 2019.

Jane X Wang, Zeb Kurth-Nelson, Dhruva Tirumala, Hubert Soyer, Joel Z Leibo, Remi Munos, Charles Blundell, Dharshan Kumaran, and Matt Botvinick. Learning to reinforcement learn. *arXiv preprint arXiv:1611.05763*, 2016a.

Jane X. Wang, Zeb Kurth-Nelson, Dhruva Tirumala, Hubert Soyer, Joel Z. Leibo, Rémi Munos, Charles Blundell, Dharshan Kumaran, and Matthew M Botvinick. Learning to reinforcement learn. *ArXiv*, abs/1611.05763, 2016b.

Yufei Wang, Qiwei Ye, and Tie-Yan Liu. Beyond exponentially discounted sum: Automatic learning of return function. *arXiv preprint arXiv:1905.11591*, 2019.

C. J. C. H. Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge, England, 1989.

Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

Robert C Wilson, Andra Geana, John M White, Elliot A Ludvig, and Jonathan D Cohen. Humans use directed and random exploration to solve the explore–exploit dilemma. *Journal of Experimental Psychology: General*, 143(6):2074, 2014.

Kelvin Xu, Ellis Ratner, Anca Dragan, Sergey Levine, and Chelsea Finn. Learning a prior over intent via meta-inverse reinforcement learning. In *Proceedings of the 36th International Conference on Machine Learning*, pages 6952–6962, 2019.

Tianbing Xu, Qiang Liu, Liang Zhao, and Jian Peng. Learning to explore via meta-policy gradient. In *International Conference on Machine Learning*, pages 5459–5468, 2018a.

Zhongwen Xu, Hado P van Hasselt, and David Silver. Meta-gradient reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 2396–2407, 2018b.

Zhongwen Xu, Hado van Hasselt, Matteo Hessel, Junhyuk Oh, Satinder Singh, and David Silver. Meta-gradient reinforcement learning with an objective discovered online. *arXiv preprint arXiv:2007.08433*, 2020.

Tom Zahavy, Zhongwen Xu, Vivek Veeriah, Matteo Hessel, Junhyuk Oh, Hado P van Hasselt, David Silver, and Satinder Singh. A self-tuning actor-critic algorithm. *Advances in Neural Information Processing Systems*, 33, 2020.

Amy Zhang, Harsh Satija, and Joelle Pineau. Decoupling dynamics and reward for transfer learning. *arXiv preprint arXiv:1804.10689*, 2018.

Zeyu Zheng, Junhyuk Oh, and Satinder Singh. On learning intrinsic rewards for policy gradient methods. In *Advances in Neural Information Processing Systems*, pages 4644–4654, 2018.

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 *International Conference on Machine Learning*, pages 11436–11446. PMLR, 2020.