Towards More General and Adaptive Deep Reinforcement Learning Agents

by

Roberta Raileanu

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

New York University

September, 2021

Professor Rob Fergus

© Roberta Raileanu

all rights reserved, 2021

To my loving and supportive parents, Mirela and Ionel.

Acknowledgements

First and foremost, I would like to thank my advisor, Rob Fergus, for his unwavering support and guidance throughout my Ph.D. I am deeply grateful to him for taking me as a Ph.D. student in his lab, as well as for his constant patience, advice, and encouragement over the last 5 years. I would also like to thank Tim Rocktäschel, Lerrel Pinto, Arthur Szlam, and Joan Bruna for serving on my thesis committee and for providing constructive feedback throughout this process.

During my Ph.D., I was fortunate to meet a number of amazing mentors from whom I've learned a lot. In particular, I would like to thank Tim Rocktäschel, who advised me during an internship at Facebook AI Research, for his constant trust, support, honest feedback, and for creating space for growth. Over the past 5 years, I had the chance to collaborate with Arthur Szlam, to whom I'm thankful for sharing his immense knowledge, creative ideas, and valuable insights. I would also like to thank Katja Hofman, who hosted me as an intern at Microsoft Research, for being an inspiration, as well as for the stimulating discussions and career advice. During my DeepMind internship, I was fortunate to be advised by Max Jaderberg, to whom I'm very grateful for giving me the chance to contribute to a very exciting project, for being so welcoming, supportive, and providing clear direction and feedback when needed.

During my time at NYU's CILVR lab, Facebook AI Research, Microsoft Research, and DeepMind, I had the chance of meeting many brilliant mentors, collaborators, and colleagues, to whom I would like to thank for all that they taught me: Kyunghyun Cho, Brenden Lake, Ethan Perez, Cinjon Resnick, Emily Denton, Sainbayar Sukhbaatar, Ilya Kostrikov, Denis Yarats, Max Goldstein, Sean

Welleck, Yiming Zhao, Jake Zhao, Jordan Ash, Nikita Nangia, Kianté Brantley, Michael Henaff, Alexander Rives, Mark Goldstein, Aahlad Puli, Jaan Altosaar, Ilya Kulikov, Sreyas Mohan, Isaac Henrion, Jason Lee, David Brandfonbrener, Aaron Zweig, Elman Mansimov, William Whitney, Martin Arjovski, Sam Devlin, Jaroslaw Rzepecki, Edward Grefenstette, Heinrich Küttler, Nantas Nardelli, Victor Zhong, Alexander Miller, Andres Campero, Adam Stooke, Anuj Mahajan, Jakob Bauer, Jakub Sygnowski, Michael Mathieu, Valentin Dalibard, and Wojciech Marian Czarnecki, and probably others whom I have forgotten that I hope will forgive me.

I would also like to say a special thank you to some of my friends who have been close to me during this time and have made the journey much more enjoyable: Tanya Efremova, Maddie Gilbert, Roxana Pop, Ana Stoica, Tudor Giurgica-Tiron, Ioanna Tzialla, Matthew Morse, Davi Colli Tozoni, Siddartha Krishna, Cynthia Steinhardt, Eve Matthaey, Andrea Ho, Maha Chaudhry, and many others.

Finally, I am extremely grateful to my mother and father for nurturing my desire for knowledge and learning, for allowing me to chase my dreams, and for never telling me a mountain is too high for me to climb. I am also deeply thankful to my wonderful sister for her positive energy, wisdom, and for the most fun conversations about anything in the world.

Last, but not least, this work would not have been possible without the unconditional love, support, and patience from my partner, Liviu, even from across an ocean.

Abstract

Building agents with general skills that can be applied in a wide range of settings has been a long-standing problem in machine learning. The most popular framework for training agents to make sequential decisions in order to maximize reward in a given environment is Reinforcement Learning (RL). Over the last decade, deep reinforcement learning, where RL agents are parameterized by neural networks, has achieved impressive results on a number of tasks, from games such as Atari [Mnih et al. 2013], Go [Silver et al. 2016], StarCraft [Vinyals et al. 2019], or Dota [Berner et al. 2019a], to continuous control tasks with applications in robotics [Lillicrap et al. 2015; Gu et al. 2017].

However, current RL agents are prone to overfitting and struggle to generalize when even minor perturbations are applied to the training environment [Rajeswaran et al. 2017b; Zhang et al. 2018a,d]. This hinders progress on real-world applications such as autonomous vehicles or home robots, where agents need to deal with a large variety of scenarios. In this thesis, we introduce several methods for improving the generalization and adaptation of deep reinforcement learning agents. We start by studying the problem of zero-shot generalization to new instances of a task after training on a limited number of environments. In this setting, agents are trained directly from partial observations of the environment in the form of images. We first propose an approach for regularizing the policy and value function of a RL agent and automatically finding an effective type of data augmentation for a given task. In addition, we demonstrate that a naive application of data augmentation in RL can hurt performance in practice and is not theoretically sound for a certain class of algorithms. We also identify that there is an asymmetry between the information needed to represent the optimal policy and the true value function, which leads to overfitting when using standard deep RL algorithms. As a step towards solving this problem, we propose a method which decouples the optimization of the policy and value, and constrains the representation to be invariant to the task instance. Next, we focus on the problem of learning general exploration strategies for environments with sparse rewards where the layout changes with each episode. We demonstrate that prior exploration methods designed for fixed environments fall short in this setting. To address this limitation, we formulate a new type of intrinsic reward which encourages agents to impact their environments. Then, we discuss a novel approach for fast adaptation to new dynamics. We show that our method, which leverages self-supervised techniques to learn policy and environment embeddings, enables adaptation within a single episode on a number of continuous control tasks. Finally, we investigate how agents can learn more flexible strategies for interacting with different opponents and collaborators.

Contents

List of Figures

List of Tables

1 Introduction

Humans are capable of employing strategies learned in one context to solve problems in many other circumstances. In contrast, current reinforcement learning methods struggle to generalize to new scenarios, often needing hundreds of thousands of diverse environments or task instances [Cobbe et al. 2019a]. In this thesis, we aim to develop reinforcement learning algorithms which generalize across a wider range of settings including completely new ones which have not been used during training.

Over the past decade, deep learning methods have revolutionized many areas of machine learning, such as computer vision [Ciresan et al. 2011; Ciregan et al. 2012; Krizhevsky et al. 2012] speech recognition [Hinton et al. 2012], natural language processing [Bahdanau et al. 2014; Cho et al. 2014; Mikolov et al. 2013; Sutskever et al. 2014; Devlin et al. 2018; Brown et al. 2020], and reinforcement learning. Deep reinforcement learning (RL), a combination of deep learning and reinforcement learning, is one of the most popular frameworks for developing agents that can solve complex tasks which require sequential decision making. RL agents learn to act in new environments through trial and error, in an attempt to maximize their total reward. Recently, RL algorithms have demonstrated impressive results on a number of different tasks, such as achieving superhuman performance on classical Atari games [Mnih et al. 2013], defeating the world champion at Go [Silver et al. 2017, 2018], achieving professional level in popular video games such as Dota [Berner et al. 2019b] or Starcraft [Vinyals et al. 2019], and mastering certain continuous control tasks [Lillicrap et al. 2015]. However, most success stories in single-agent RL evaluate agents on the same environment as the one used for training. Thus, generalization to new scenarios remains a major challenge in deep reinforcement learning. Current methods fail to generalize to unseen environments even when trained on similar settings [Farebrother et al. 2018; Packer et al. 2018b; Zhang et al. 2018b; Cobbe et al. 2018; Gamrian and Goldberg 2019; Cobbe et al. 2019b; Song et al. 2020]. This indicates that standard RL agents memorize specific trajectories rather than learning transferable skills.

Throughout this thesis, we will refer to the general problem that needs to be solved as the *task* (*e.g.* find a goal inside a maze) and to the particular instantiation of this task as the *environment* (*e.g.* maze layout, colors, textures, locations of the objects, environment dynamics etc.). The environment can be *singleton* or *procedurally generated.* Singleton environments are those in which the agent has to solve the same task in the same environment in every episode, *i.e.*., the environment does not change between episodes. In procedurally generated environments, the agent needs to solve the same task, but in every episode the environment is constructed differently (*e.g.* resulting in a different maze layout), making it unlikely for an agent to ever visit the same state twice. Thus, agents in such environments have to learn policies that generalize well across a very large state space.

This thesis aims to improve the generalization and adaptation of deep reinforcement learning agents through advancements in four research areas:

1. zero-shot generalization to new instances of the same task when learning directly from images;
2. exploration of procedurally generated environments with sparse rewards;
3. fast adaptation to new environment dynamics within a single episode;
4. strategic interaction with different opponents and collaborators.

Generalization in Reinforcement Learning. In order to enable real-world applications of deep reinforcement learning, agents need to generalize to new scenarios since the world is constantly changing and it is unfeasible to train on all possible variations an agent might encounter at test time. In reinforcement learning, there are multiple facets of generalization. For example, an agent may need to generalize to new states (*e.g.* layouts), observations (*e.g.* visual appearance such as colors or patterns), dynamics (*e.g.* weather or road conditions), or reward functions (*e.g.* other tasks). In the first two chapters, we focus on zero-shot generalization to new instances of a task which implies generalization to different states, observations, and dynamics, with the reward function being the same across all environments (including training and test ones). We propose two approaches which improve generalization in this setting. The first one uses data augmentation to regularize the policy and value function of an RL agent. We show that a naive application of data augmentation [Cobbe et al. 2018; Ye et al. 2020; Laskin et al. 2020] is not theoretically sound for a certain class of algorithms and can decrease performance in practice. We also identify a particular problem with popular deep RL methods which use a shared network to learn both the policy and the value function. More information is needed to learn the optimal policy than accurately estimate the value function, which leads to overfitting. To alleviate this problem, we propose the use of separate networks for training the policy and value function, as well as an auxiliary loss to avoid capturing instance-specific features which are not necessary for representing the optimal policy. At the time of release, both of our proposed approaches achieved state-of-the-art results on a challenging benchmark of procedurally generated games [Cobbe et al. 2019a]

Exploration in Procedurally Generated Environments. Procedurally generated environments are only now starting to become more widely used as testbeds for RL algorithms. Hence, until now, most exploration methods have been designed for and evaluated on singleton environments. Here, we discuss how these methods have significant limitations in procedurally generated environments even if they are very effective on challenging singleton environments such as Montezuma's Revenge [Bellemare et al. 2013; Burda et al. 2019b]. We also introduce a new type of intrinsic reward which is better suited for training in environments where each episode is different and the reward is very sparse. We demonstrate superior performance on a set of challenging tasks in partially observed and procedurally generated gridworlds [Chevalier-Boisvert et al. 2018]. Exploration of procedurally generated environments is an important problem with consequences for real-world applications. For example, a rescue robot might have to explore new environments when deployed. The test environments might share some similarities with the training environments, but are nevertheless different and cannot be precisely anticipated. In such cases, possessing general exploration strategies (which are effective in a wide variety of settings) is crucial.

Adaptation to New Dynamics. Similarly, the ability to quickly adapt to new environment dynamics is an essential property for an intelligent agent acting in the real-world, whether it acts in the virtual or physical space. For example, a self-driving car needs different policies for controlling the vehicle when it rains, snows, or is sunny and dry. In this thesis, we propose a new framework for fast adaptation to new dynamics. Our approach learns a value in a space of policies and dynamics, which is then used at test time to find an effective behavior for the given environment. Our method proves to be at least as good as popular meta-learning techniques [Finn et al. 2017] on a series of continuous control tasks [Todorov et al. 2012a].

Interacting with Other Agents. Many applications of interest (such as conversational agents or virtual assistants) require agents to interact with others in the environment and adapt their behavior to others' hidden intentions. We study this setting and propose a new method for modeling other agents and inferring their goals. This information is used by the acting agent to adapt its policy accordingly in order to maximize reward. Designing agents which can take into account others' preferences (without the need for explicitly specifying them in the form of instructions) could lead to more flexible and useful agents, thus unlocking important applications such as virtual assistants.

This thesis is organized as follows. Chapter 2 provides a summary of reinforcement learning in single-agent singleton environments, single-agent procedurally generated environments, and multi-agent settings. Chapters 3 and 4 study the problem of zero-shot generalization to new instances of a task, and introduce two approaches which alleviate overfitting in RL. Chapter 5 proposes a new intrinsic reward for efficient exploration of procedurally generated environments with sparse rewards. Chapter 6 discusses a new framework for fast adaptation to new environment dynamics, while Chapter 7 proposes a new method for adapting the agent's policy depending on the behavior of other agents. Finally, in Chapter 8, we summarize our findings and discuss avenues for future research.

1.1 List of Contributions

- **Roberta Raileanu**, Max Goldstein, Denis Yarats, Ilya Kostrikov, Rob Fergus

  Automatic Data Augmentation for Generalization in Reinforcement Learning. *NeurIPS*, 2021.

  Code: `https://github.com/rraileanu/auto-drac`

- **Roberta Raileanu**, Rob Fergus

  Decoupling Value and Policy for Generalization in Reinforcement Learning. *ICML*, 2021.

  Code: `https://github.com/rraileanu/idaac`

- **Roberta Raileanu**, Tim Rocktäschel

  Rewarding Impact-Driven Exploration for Procedurally-Generated Environments. *ICLR*, 2020.

  Code: `https://github.com/facebookresearch/impact-driven-exploration`

- **Roberta Raileanu**, Max Goldstein, Arthur Szlam, Rob Fergus

  Fast Adaptation to New Environments via Policy-Dynamics Value Functions. *ICML*, 2020.

  Code: `https://github.com/rraileanu/policy-dynamics-value-functions`

- **Roberta Raileanu**, Emily Denton, Arthur Szlam, Rob Fergus

  Modeling Others using Oneself in Multi-Agent Reinforcement Learning. *ICML*, 2018.

2 | Background

In this chapter, we briefly review single-agent and multi-agent reinforcement learning, as well as how the formulation changes when training and testing on procedurally generated environments. The purpose of this section is to give a brief introduction of the class of problems and techniques used throughout the thesis. Each chapter includes more detailed definitions for the corresponding problem setting, thus being self-contained.

2.1 Reinforcement Learning

Reinforcement Learning (RL) is an area of machine learning that concerns learning policies that optimize cumulative rewards based on interactions with learning environments. A common framework for this optimization problem is the discounted infinite horizon Markov Decision Processes (MDPs) [Bellman 1957]. MDPs can be described as a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$, where $\mathcal{S}$ is the state space; $\mathcal{A}$ is the action space; $\mathcal{P}$ is the transition function (or dynamics) such that for any $s_t \in \mathcal{S}, s_{t+1} \in \mathcal{S}, a_t \in \mathcal{A}, \mathcal{P}(s_{t+1}|s_t, a_t)$ represents the probability of transitioning to state s_{t+1} from state s_t by taking action a_t; $\mathcal{R}$ is a reward function such that a reward s_t at time step t is computed as $\mathcal{R}(s_t, a_t)$; and $\gamma \in [0, 1]$ is a discount factor.

A deterministic and stationary policy $\pi : \mathcal{S} \to \mathcal{A}$ specifies a decision-making strategy in which the agent chooses actions based on the current state, *i.e.*, $a_t = \pi(s_t)$. More generally, the agent may also choose actions according to a stochastic policy $\pi : \mathcal{S} \to \Delta(\mathcal{A})$, *i.e.*, $a_t \sim \pi(s_t)$.

The goal of the agent is to choose a policy π to maximize the expected discounted sum of rewards, also called the expected return or value:

$$\mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t \mathcal{R}(s_t, a_t) \mid \pi,\ s_0\right]$$

The expectation is with respect to the randomness of the trajectory, that is, the randomness in state transitions and the stochasticity of π.

Given an MDP, for any state s and policy π, we define the value function $V^\pi : \mathcal{S} \longrightarrow \mathbb{R}$ as:

$$V^\pi(s) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t \mathcal{R}(s_t, a_t) \mid a_t \sim \pi(s_t),\ s_{t+1} \sim \mathcal{P}(s_t, a_t),\ s_0 = s\right],$$

which is the value obtained by following policy π starting at state s.

Similarly, we define the state-action value function or Q function as $Q^\pi : \mathcal{S} \times \mathcal{A} \longrightarrow \mathbb{R}$ as:

$$Q^\pi(s, a) \ = \ \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t \mathcal{R}(s_t, a_t) \mid a_t \sim \pi(s_t),\ s_{t+1} \sim \mathcal{P}(s_t, a_t),\ s_0 = s,\ a_0 = a\right].$$

We can also define the advantage function $A^\pi(s, a)$ in the following way:

$$A^\pi(s, a) = Q^\pi(s, a) - V^\pi(s).$$

The advantage measures how much more return can be expected by taking action a in state s and then following policy π relative to following policy π starting in state s.

Finally, the *optimal policy* is defined as:

$$\pi^\star(\cdot|s) \ = \ \text{argmax}_\pi \ \mathbb{E}_{a\sim\pi(\cdot|s)}\left[\ Q^\pi(s, a)\ \right].$$

There are two large classes of solutions to the problem of finding the optimal policy for a given MDP, namely policy gradient methods and value based methods. In this thesis, we mostly focus on policy gradient approaches, so the following paragraphs will describe the core ideas behind this type of algorithms.

Policy gradient methods aim to model and optimizing the policy directly. The policy is usually parameterized by some function with learnable parameters θ, $\pi_\theta(a|s)$. Various algorithms can be applied to optimize θ so that $\pi_\theta(a|s)$ maximizes the following objective function:

$$J(\theta) \ = \sum_{s\in\mathcal{S}} d^\pi(s)\ V^\pi(s) \ = \sum_{s\in\mathcal{S}} d^\pi(s) \sum_{a\in\mathcal{A}} \pi_\theta(a|s)\ Q^\pi(s,a),$$

where $d^\pi(s)$ is the stationary distribution of Markov chain for π_θ (on-policy state distribution under π), *i.e.*, $d^\pi(s) = \lim_{t\to\infty}\mathcal{P}(s_t = s|s_0, \pi_\theta)$. For simplicity, the parameter θ is omitted for the policy π_θ when the policy is present in the subscript of other functions.

Using gradient ascent, we can move θ toward the direction suggested by the gradient $\nabla_\theta J(\theta)$ to find the parameters θ which lead to the highest return when acting according to policy π_θ.

The gradient can be written as:

$$\begin{aligned}
\nabla_\theta J(\theta) &= \nabla_\theta \sum_{s\in\mathcal{S}} \pi(s) \sum_{a\in\mathcal{A}} \pi_\theta(a|s) Q^\pi(s,a) \\
&\propto \sum_{s\in\mathcal{S}} d^\pi(s) \sum_{a\in\mathcal{A}} Q^\pi(s,a)\nabla_\theta \pi_\theta(a|s) \\
&= \sum_{s\in\mathcal{S}} d^\pi(s) \sum_{a\in\mathcal{A}} \pi_\theta(a|s) Q^\pi(s,a)\frac{\nabla_\theta \pi_\theta(a|s)}{\pi_\theta(a|s)} \\
&= \mathbb{E}_\pi\Big[Q^\pi(s,a)\nabla_\theta \ln \pi_\theta(a|s)\Big],
\end{aligned}$$

where $\mathbb{E}_\pi$ refers to $\mathbb{E}_{s\sim d^\pi, a\sim\pi_\theta}$ when both the action and state distribution follow the policy π_θ. The expected return can be estimated with Monte-Carlo methods by collecting trajectories from the environment using the current policy.

The above derivation of the gradient is called the policy gradient theorem, which lays the theoretical foundation for policy gradient algorithms. This vanilla policy gradient update has no bias but high variance. Many following algorithms were proposed to reduce the variance while keeping the bias unchanged.

Schulman et al. [2015b] unifies policy gradient methods under a general formulation,

$$\nabla_\theta J(\theta) = \mathbb{E}\left[\sum_{t=0}^{\infty} \psi_t \nabla_\theta \log \pi_\theta (a_t | s_t) \right], \tag{2.1}$$

where ψ_t may be one of the following: (i) $\sum_{t=0}^{\infty} r_t$: total reward of the trajectory; (ii) $\sum_{t'=t}^{\infty} r_{t'}$: reward following action a_t; (iii) $\sum_{t'=t}^{\infty} r_{t'} - b(s_t)$: baselined version of the previous formula; (iv) $Q^\pi(s,a)$: state-action value function (v) $A^\pi(s,a)$: advantage function; (vi) $r_t + V^\pi(s_{t+1}) - V^\pi(s_t)$: temporal difference (TD) residual.

Two main components in policy gradient are the policy model and the value function. It makes a lot of sense to learn the value function in addition to the policy, since knowing the value function can assist the policy update, such as by reducing gradient variance in vanilla policy gradients, and that is exactly what the **Actor-Critic** method does.

Actor-critic methods consist of two models, which may optionally share parameters:

1. *Critic* updates the value function parameters ϕ and depending on the algorithm it could be state-action value function $Q_\phi(a|s)$ or value function $V_\phi(s)$.

2. *Actor* updates the policy parameters θ for $\pi_\theta(a|s)$, in the direction suggested by the critic.

Proximal Policy Optimization (PPO) [Schulman et al. 2017] is an actor-critic RL algorithm that learns a policy π_θ and a value function V_θ with the goal of finding an optimal policy for a given MDP. PPO alternates between sampling data through interaction with the environment and optimizing an objective function using stochastic gradient ascent. At each iteration, PPO maximizes the following objective:

$$J_{\text{PPO}} = J_\pi - \alpha_1 J_V + \alpha_2 S_{\pi_\theta}, \tag{2.2}$$

where α_1, α_2 are weights for the different loss terms, S_{π_θ} is the entropy bonus for aiding exploration, J_V is the value function loss defined as

$$J_V = \left(V_\theta(s) - V_t^{\text{target}}\right)^2 .$$

The policy objective term J_π is based on the policy gradient objective which can be estimated using importance sampling in off-policy settings (*i.e.* when the policy used for collecting data is different from the policy we want to optimize):

$$J_{PG}(\theta) = \sum_{a \in \mathcal{A}} \pi_\theta(a|s) \hat{A}_{\theta_{\text{old}}}(s,a) = \mathbb{E}_{a \sim \pi_{\theta_{\text{old}}}} \left[\frac{\pi_\theta(a|s)}{\pi_{\theta_{\text{old}}}(a|s)} \hat{A}_{\theta_{\text{old}}}(s,a) \right], \tag{2.3}$$

where $\hat{A}(\cdot)$ is an estimate of the advantage function, θ_{old} are the policy parameters before the update, $\pi_{\theta_{old}}$ is the behavior policy used to collect trajectories (*i.e.* that generates the training distribution of states and actions), and π_θ is the policy we want to optimize (*i.e.* that generates the true distribution of states and actions).

This objective can also be written as

$$J_{PG}(\theta) = \mathbb{E}_{a \sim \pi_{\theta_{\text{old}}}} \left[r(\theta) \hat{A}_{\theta_{\text{old}}}(s,a) \right], \tag{2.4}$$

where

$$r_\theta = \frac{\pi_\theta(a|s)}{\pi_{\theta_{\text{old}}}(a|s)}$$

is the importance weight for estimating the advantage function.

PPO is inspired by TRPO [Schulman et al. 2015a], which constrains the update so that the policy does not change too much in one step. This significantly improves training stability and leads to better results than vanilla policy gradient algorithms. TRPO achieves this by minimizing the KL divergence between the old (*i.e.* before an update) and the new (*i.e.* after an update) policy.

PPO implements the constraint in a simpler way by using a clipped surrogate objective instead of the more complicated TRPO objective. More specifically, PPO imposes the constraint by forcing $r(\theta)$ to stay within a small interval around 1, precisely $[1-\epsilon, 1+\epsilon]$, where ϵ is a hyperparameter. The policy objective term from equation (2.2) becomes

$$J_{\pi} = \mathbb{E}_{\pi}\left[\min\left(r_{\theta}\hat{A},\ \text{clip}\left(r_{\theta}, 1-\epsilon, 1+\epsilon\right)\hat{A}\right)\right],$$

where $\hat{A} = \hat{A}_{\theta_{\text{old}}}(s, a)$ for brevity. The function clip$(r(\theta), 1-\epsilon, 1+\epsilon)$ clips the ratio to be no more than $1+\epsilon$ and no less than $1-\epsilon$. The objective function of PPO takes the minimum one between the original value and the clipped version so that agents are discouraged from increasing the policy update to extremes for better rewards.

In some of the chapters, we consider the **Partially Observable Markov Decision Process (POMDP)** which is a combination of an MDP to model system dynamics with a hidden Markov model that connects unobservant system states to observations. The agent can perform actions which affect the system or (*i.e.* underlying state of the environment) with the goal to maximize a reward that depends on the sequence of system states and actions taken by the agent. However, the agent cannot directly observe the system state, but at each point in time, the agent makes observations that depend on the state. The agent uses these observations to form a belief of the underlying state of the POMDP.

Formally, a POMDP can be described by a tuple = $(\mathcal{S}, \mathcal{A}, \mathcal{O}, \mathcal{P}, \mathcal{R}, \mathcal{Z}, \gamma)$, where $\mathcal{S}$ is the state space, $\mathcal{A}$ is the action space, $\mathcal{P}(s'|s, a)$ is the state transition probability function, $\mathcal{R} : S \times A \rightarrow \mathbb{R}$ is the reward function, $\mathcal{O}$ is the observation space, $\mathcal{Z}(o|s', a)$ is the observation transition probability function, and γ is the discount factor. At each time step, the environment is in some state $s \in \mathcal{S}$. The agent chooses an action $a \in \mathcal{A}$, which causes the environment to transition to state $s' \in \mathcal{S}$ with probability $\mathcal{P}(s'|s, a)$. At the same time, the agent receives an observation $o \in \mathcal{Z}(o|s', a)$ which depends on the current state and previous action taken by the agent. Finally the agent receives a reward R(s, a). Then, the process repeats. As before, the goal is to select actions so that the expected return is maximized.

2.2 Procedurally Generated Environments

For a long time, it was common to evaluate reinforcement learning agents on the same environment as the one used for training. This setting already posed multiple challenges for RL methods at the time, from credit assignment to exploration in sparse reward environments and learning from high-dimensional observations such as images (where it is important to have useful representations of the inputs in order to generalize across different states). Popular benchmarks such as the Arcade Learning Environment (ALE) [Bellemare et al. 2013], MuJoCo [Todorov et al. 2012b], or DeepMind Control Suite (DMC) [Tassa et al. 2018b], which consist of singleton environments (where the environment does not change throughout training or testing), have led to significant algorithmic progress in all areas mentioned above [Mnih et al. 2013; Schulman et al. 2015a; Lillicrap et al. 2015; Mnih et al. 2016a; Schulman et al. 2017; Espeholt et al. 2018a; Burda et al. 2019a; Ecoffet et al. 2019].

However, RL agents trained on such singleton environments are prone to overfitting to a specific environment and often struggle to generalize to even slightly different settings [Rajeswaran et al. 2017b; Zhang et al. 2018a,d]. This is problematic for real-world applications such as autonomous vehicles or home robots, where agents need to deal with a very large variety of scenarios. In addition, the world is constantly changing, so agents will need to generalize or quickly adapt to new circumstances at test time. Procedurally generated environments are a promising path towards developing more flexible agents and testing the generalization abilities of various RL methods. Here, the game state is generated programmatically in every episode, making it extremely unlikely for an agent to visit the exact state more than once during its lifetime. Recently, a number of such environments have been released, such as DeepMind Lab [Beattie et al. 2016], Sokoban [Racanière et al. 2017], Malmö [Johnson et al. 2016], CraftAssist [Jernite et al. 2019], MiniGrid [Chevalier-Boisvert et al. 2018], Sonic [Nichol et al. 2018a], CoinRun [Cobbe et al. 2019c], Obstacle Tower [Juliani et al. 2019a], Capture the Flag [Jaderberg et al. 2019], Procgen [Cobbe et al. 2019a], or The NetHack Learning Environment (NLE) [Kuttler et al. 2020]. Training agents on procedurally generated environments is becoming increasingly more common [Raileanu and Rocktäschel 2020; Zhang et al. 2020d; Campero et al. 2020; Laskin et al. 2020; Raileanu et al. 2020; Jiang et al. 2020; Mazoure et al. 2020; Cobbe et al. 2020; Wang et al. 2020; Raileanu and Fergus 2021; Mazoure et al. 2021]. Moreover, these types of environments provide a systematic way for probing generalization by holding out a number of the generated environments for testing, similar to the standard practice in supervised learning.

When training on procedurally generated environments, we consider a distribution $q(m)$ of POMDPs $m \in \mathcal{M}$, with m defined by the tuple $m = (\mathcal{S}_m, \mathcal{A}, \mathcal{O}_m, \mathcal{P}_m, \mathcal{R}_m, \mathcal{Z}_m, \gamma)$, where $\mathcal{S}_m$ is the state space, $\mathcal{O}_m$ is the observation space, $\mathcal{A}$ is the action space, $\mathcal{P}_m(s'|s, a)$ is the state transition probability distribution, $\mathcal{Z}_m(o|s', a)$ is the observation probability distribution, $\mathcal{R}_m(s, a)$ is the reward function, and γ is the discount factor. During training, we restrict access to a fixed set of POMDPs, $\mathcal{M}_{\text{train}} = \{m_1, \dots, m_n\}$, where $m_i \sim q$, $\forall\, i = \overline{1, n}$. The goal is to find a policy π_θ which maximizes the expected discounted reward over the entire distribution of POMDPs, $J(\pi_\theta) = \mathbb{E}_{q,\pi,\mathcal{P}_m}\left[\sum_{t=0}^{\infty} \gamma^t \nabla_m(s_t, a_t)\right]$.

In practice, each POMDP m is determined by a seed or integer which is used to generate the corresponding environment. When we want to evaluate zero-shot generalization to new POMDPs from the same family, we can train agents on a fixed set of n POMDPs (generated using seeds from 1 to n), and test them on the full distribution of levels (generated using any computer integer seed).

2.3 Multi-Agent Reinforcement Learning

In Chapter 7, we study the problem of adapting an agent's policy depending on the behavior of other agents in order to effectively cooperate or compete with them. Thus, here we formalize the multi-agent setting, emphasizing how it differs from the single-agent framework.

In multi-agent reinforcement learning (MARL), multiple agents take actions in the environment, each of them aiming to maximize their individual return. This can be formalized as a *stochastic game* [Shapley 1953], G, defined by a tuple $G = (\mathcal{S}, \mathcal{U}, O, \mathcal{P}, \mathcal{R}, \mathcal{Z}, \gamma, n)$, in which n agents identified by $a \in \mathcal{A} \equiv \{1, \ldots, n\}$ choose sequential actions. As in the single-agent setting, the environment has a true state $s \in S$. At each time step, each agent takes an action $u^a \in \mathcal{U}$, forming a joint action $\mathbf{u} \in \mathbf{U} \equiv \mathcal{U}^n$ which induces a transition in the environment according to the transition function $\mathcal{P}(s'|s, \mathbf{u}) : \mathcal{S} \times \mathbf{U} \times \mathcal{S} \to \mathbb{R}$. Here $\mathcal{U}$ is the action space, the reward function specifies a reward for each agent, $\mathcal{R}(s, \mathbf{u}, a) : \mathcal{S} \times \mathbf{U} \times \mathcal{A} \to \mathbb{R}$ and, as before, $\gamma \in [0, 1]$ is a discount factor. We denote joint quantities over agents in bold, e.g., u, and joint quantities over agents other than a given agent a with the superscript a, *e.g.*, $\mathbf{u}^a$.

3 | Automatic Augmentation for Generalization in Reinforcement Learning

Deep reinforcement learning (RL) agents often fail to generalize beyond their training environments. To alleviate this problem, recent work has proposed the use of data augmentation. However, different tasks tend to benefit from different types of augmentations and selecting the right one typically requires expert knowledge. In this chapter, we introduce three approaches for automatically finding an effective augmentation for any RL task. These are combined with two new regularization terms for the policy and value function, required to make the use of data augmentation theoretically sound for actor-critic algorithms. Relative to other approaches, our agent learns policies and representations which are more robust to changes in the environment that are irrelevant for solving the task, such as the background. At the time of releasing this work, our method achieved a new state-of-the-art on the Procgen benchmark [Cobbe et al. 2019a] and outperformed popular RL algorithms on DeepMind Control tasks with distractors [Zhang et al. 2020b].

3.1 Introduction

Generalization to new environments remains a major challenge in deep reinforcement learning (RL). Current methods fail to generalize to unseen environments even when trained on similar settings [Farebrother et al. 2018; Packer et al. 2018b; Zhang et al. 2018b; Cobbe et al. 2018; Gamrian and Goldberg 2019; Cobbe et al. 2019b; Song et al. 2020]. This indicates that standard RL agents memorize specific trajectories rather than learning transferable skills. Several strategies have been proposed to alleviate this problem, such as the use of regularization [Farebrother et al. 2018; Zhang et al. 2018b; Cobbe et al. 2018; Igl et al. 2019], data augmentation [Cobbe et al. 2018; Lee et al. 2020; Ye et al. 2020; Kostrikov et al. 2020; Laskin et al. 2020], or representation learning [Zhang et al. 2020a,c; Mazoure et al. 2020; Stooke et al. 2020; Agarwal et al. 2021]. In this work, we focus on the use of data augmentation in RL. We identify key differences between supervised learning and reinforcement learning which need to be taken into account when using data augmentation in RL.

More specifically, we show that a naive application of data augmentation can lead to both theoretical and practical problems with standard RL algorithms, such as unprincipled objective estimates and poor performance. As a solution, we propose **Data-regularized Actor-Critic** or **DrAC**, a new algorithm that enables the use of data augmentation with actor-critic algorithms in a theoretically sound way. Specifically, we introduce two regularization terms which constrain the agent's policy and value function to be invariant to various state transformations. Empirically, this approach allows the agent to learn useful behaviors (outperforming strong RL baselines) in settings in which a naive use of data augmentation completely fails or converges to a sub-optimal policy. While we use Proximal Policy Optimization (PPO, Schulman et al. [2017]) to describe and validate our approach, the method can be easily integrated with any actor-critic algorithm with a discrete stochastic policy such as A3C [Mnih et al. 2013], SAC [Haarnoja et al. 2018], or IMPALA [Espeholt et al. 2018a].

The current use of data augmentation in RL either relies on expert knowledge to pick an

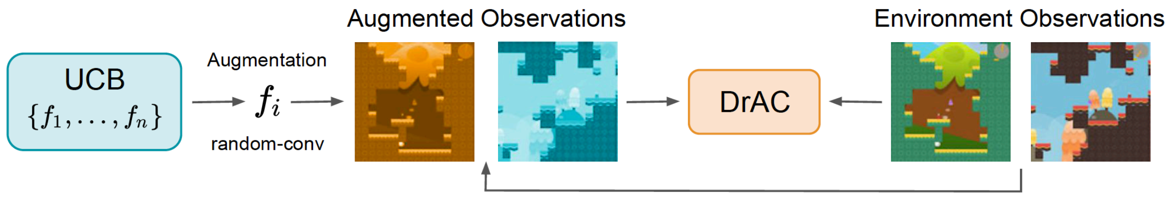


Figure 3.1: Overview of UCB-DrAC. A UCB bandit selects an image transformation (*e.g.* random-conv) and applies it to the observations. The augmented and original observations are passed to a regularized actor-critic agent (*i.e.* DrAC) which uses them to learn a policy and value function which are invariant to this transformation.

appropriate augmentation [Cobbe et al. 2018; Lee et al. 2020; Kostrikov et al. 2020] or separately evaluates a large number of transformations to find the best one [Ye et al. 2020; Laskin et al. 2020]. In this chapter, we propose three methods for automatically finding a useful augmentation for a given RL task. The first two learn to select the best augmentation from a fixed set, using either a variant of the upper confidence bound algorithm (UCB, Auer [2002]) or meta-learning (RL^2, Wang et al. [2016b]). We refer to these methods as **UCB-DrAC** and **RL2-DrAC**, respectively. The third method, **Meta-DrAC**, directly meta-learns the weights of a convolutional network, without access to predefined transformations (MAML, Finn et al. [2017]). Figure 3.1 gives an overview of UCB-DrAC.

We evaluate these approaches on the *Procgen* generalization benchmark [Cobbe et al. 2019b] which consists of 16 procedurally generated environments with visual observations. Our results show that UCB-DrAC is the most effective among these at finding a good augmentation, and is comparable or better than using DrAC with the best augmentation from a given set. UCB-DrAC also outperforms baselines specifically designed to improve generalization in RL [Igl et al. 2019; Lee et al. 2020; Laskin et al. 2020] on both train and test. In addition, we show that our agent learns policies and representations that are more invariant to changes in the environment which do not alter the reward or transition function (*i.e.* they are inconsequential for control), such as the background theme.

To summarize, our work makes the following contributions: (i) we introduce a principled way of using data augmentation with actor-critic algorithms, (ii) we propose a practical approach for automatically selecting an effective augmentation in RL settings, (iii) we show that the use of data augmentation leads to policies and representations that better capture task invariances, and (iv) we demonstrate state-of-the-art results on the Procgen benchmark and outperform popular RL methods on four DeepMind Control tasks with natural and synthetic distractors.

3.2 Related Work

Generalization in Deep RL. A recent body of work has pointed out the problem of overfitting in deep RL [Rajeswaran et al. 2017a; Machado et al. 2018c; Justesen et al. 2018a; Packer et al. 2018b; Zhang et al. 2018b,e; Nichol et al. 2018b; Cobbe et al. 2018, 2019b; Juliani et al. 2019b; Raileanu and Rocktäschel 2020; Kuttler et al. 2020; Grigsby and Qi 2020]. A promising approach to prevent overfitting is to apply regularization techniques originally developed for supervised learning such as dropout [Srivastava et al. 2014; Igl et al. 2019] or batch normalization [Ioffe and Szegedy 2015; Farebrother et al. 2018; Igl et al. 2019]. For example, Igl et al. [2019] use selective noise injection with a variational information bottleneck, while Lee et al. [2020] regularize the agent's representation with respect to random convolutional transformations. The use of state abstractions has also been proposed for improving generalization in RL [Zhang et al. 2020a,c; Agarwal et al. 2021]. Similarly, Roy and Konidaris [2020] and Sonar et al. [2020] learn domain-invariant policies via feature alignment, while Stooke et al. [2020] decouple representation from policy learning. Igl et al. [2020] reduce non-stationarity using policy distillation, while Mazoure et al. [2020] maximize the mutual information between the agent's representation of successive time steps. Jiang et al. [2020] improve efficiency and generalization by sampling levels according to their learning potential, while Wang et al. [2020] use mixtures of observations to impose linearity constraints between the agent's inputs and the outputs. More similar to our work, Cobbe et al. [2018], Ye et al. [2020] and Laskin et al. [2020] add augmented observations to the training buffer of an RL agent. However, as we show here, naively applying data augmentation in RL can lead to both theoretical and practical issues. Our algorithmic contributions alleviate these problems while still benefitting from the regularization effect of data augmentation.

Data Augmentation has been extensively used in computer vision for both supervised [LeCun et al. 1989; Becker and Hinton 1992; LeCun et al. 1998; Simard et al. 2003; Cireşan et al. 2011; Ciresan et al. 2011; Krizhevsky et al. 2012] and self-supervised [Dosovitskiy et al. 2016; Misra and van der Maaten 2019] learning. More recent work uses data augmentation for contrastive learning, leading to state-of-the-art results on downstream tasks [Ye et al. 2019; Hénaff et al. 2019; He et al. 2019; Chen et al. 2020]. Domain randomization can also be considered a type of data augmentation, which has proven useful for transferring RL policies from simulation to the real world [Tobin et al. 2017]. However, it requires access to a physics simulator, which is not always available. Recently, a few papers propose the use of data augmentation in RL [Cobbe et al. 2018; Lee et al. 2020; Srinivas et al. 2020; Kostrikov et al. 2020; Laskin et al. 2020], but all of them use a fixed (set of) augmentation(s) rather than automatically finding the most effective one. The most similar work to ours is that of Kostrikov et al. [2020], who propose to regularize the Q-function in Soft Actor-Critic (SAC) [Haarnoja et al. 2018] using random shifts of the input image. Our work differs from theirs in that it automatically selects an augmentation from a given set, regularizes both the actor and the critic, and focuses on the problem of generalization rather than sample efficiency. While there is a body of work on the automatic use of data augmentation [Cubuk et al. 2019a,b; Fang et al. 2019; Shi et al. 2019; Li et al. 2020], these approaches were designed for supervised learning and, as we explain here, cannot be applied to RL without further algorithmic changes.

3.3 Background

We consider the problem setting described in Section 2 where the goal is to find the optimal policy for a faimly of POMDPs.

In practice, we use the Procgen benchmark which contains 16 procedurally generated games. Each game corresponds to a distribution of POMDPs $q(m)$, and each level of a game corresponds to a POMDP sampled from that game's distribution $m \sim q$. The POMDP m is determined by the seed (*i.e.* integer) used to generate the corresponding level. Following the setup from Cobbe et al. [2019b], agents are trained on a fixed set of $n = 200$ levels (generated using seeds from 1 to 200) and tested on the full distribution of levels (generated using any computer integer seed).

3.3.1 Naive Data Augmentation in Reinforcement Learning

Image augmentation has been successfully applied in computer vision for improving generalization on object classification tasks [Simard et al. 2003; Cireşan et al. 2011; Ciregan et al. 2012; Krizhevsky et al. 2012]. As noted by Kostrikov et al. [2020], those tasks are invariant to certain image transformations such as rotations or flips, which is not always the case in RL. For example, if your observation is flipped, the corresponding reward will be reversed for the left and right actions and will not provide an accurate signal to the agent. While data augmentation has been previously used in RL settings without other algorithmic changes [Cobbe et al. 2018; Ye et al. 2020; Laskin et al. 2020], we argue that this approach is not theoretically sound.

We consider that the policy is learned using an actor-critic algorithm such as PPO [Schulman et al. 2017], which was described in Section 2. If transformations are naively applied to observations in PPO's buffer, as done in Laskin et al. [2020], the PPO objective changes and equation (2.3) is replaced by

$$J_{\text{PG}}(\theta) = \mathbb{E}_{a\sim\pi_{\theta_{\text{old}}}}\left[\frac{\pi_\theta(a|f(s))}{\pi_{\theta_{\text{old}}}(a|s)}\hat{A}_{\theta_{\text{old}}}(s,a)\right], \tag{3.1}$$

where $f : \mathcal{S} \times \mathcal{H} \to \mathcal{S}$ is the image transformation. However, the right hand side of the above equation is not a sound estimate of the left hand side because $\pi_\theta(a|f(s)) \neq \pi_\theta(a|s)$, since nothing constrains $\pi_\theta(a|f(s))$ to be close to $\pi_\theta(a|s)$. Note that in the on-policy case when $\theta = \theta_{old}$, the ratio used to estimate the advantage should be equal to one, which is not necessarily the case when using equation (3.1). In fact, one can define certain transformations $f(\cdot)$ that result in an arbitrarily large ratio $\pi_\theta(a|f(s))/\pi_{\theta_{old}}(a|s)$.

Figure 3.8 (from Page 34) shows examples where a naive use of data augmentation prevents PPO from learning a good policy in practice, suggesting that this is not just a theoretical concern. In the following section, we propose an algorithmic change that enables the use of data augmentation with actor-critic algorithms in a principled way.

3.4 Approach

3.4.1 Policy and Value Function Regularization

Inspired by the recent work of Kostrikov et al. [2020], we propose two novel regularization terms for the policy and value functions that enable the proper use of data augmentation for actor-critic algorithms. Our algorithmic contribution differs from that of Kostrikov et al. [2020] in that it constrains both the actor and the critic, as opposed to only regularizing the Q-function. This allows our method to be used with a different (and arguably larger) class of RL algorithms, namely those that learn a policy and a value function.

Following Kostrikov et al. [2020], we define an optimality-invariant state transformation $f : \mathcal{S} \times \mathcal{H} \to \mathcal{S}$ as a mapping that preserves both the agent's policy π and its value function V such that $V(s) = V(f(s, \nu))$ and $\pi(a|s) = \pi(a|f(s, \nu)), \quad \forall s \in \mathcal{S}, \ \nu \in \mathcal{H}$, where ν are the parameters of $f(\cdot)$, drawn from the set of all possible parameters $\mathcal{H}$.

To ensure that the policy and value functions are invariant to such transformation of the input state, we propose an additional loss term for regularizing the policy,

$$G_\pi = KL\left[\pi_\theta(a|s) \mid \pi_\theta(a|f(s, \nu))\right], \tag{3.2}$$

as well as an extra loss term for regularizing the value function,

$$G_V = \left(V_\phi(s) - V_\phi\left(f(s,\nu)\right)\right)^2. \tag{3.3}$$

Thus, our **data-regularized actor-critic** method, or **DrAC**, maximizes the following objective:

$$J_{\text{DrAC}} = J_{\text{PPO}} - \alpha_r(G_\pi + G_V), \tag{3.4}$$

where α_r is the weight of the regularization term (see Algorithm 1).

The use of G_π and G_V ensures that the agent's policy and value function are invariant to the transformations induced by various augmentations. Particular transformations can be used to impose certain inductive biases relevant for the task (*e.g.* invariance with respect to colors or translations). In addition, G_π and G_V can be added to the objective of any actor-critic algorithm with a discrete stochastic policy (*e.g.* A3C, TRPO, ACER, SAC, or IMPALA) without any other changes.

Note that when using DrAC, as opposed to the method proposed by Laskin et al. [2020], we still use the correct importance sampling estimate of the left hand side objective in equation (2.3) (instead of a wrong estimate as in equation (3.1)). This is because the transformed observations $f(s)$ are only used to compute the regularization losses G_π and G_V, and thus are not used for the main PPO objective. Without these extra terms, the only way to use data augmentation is as explained in Section 3.3.1, which leads to inaccurate estimates of the PPO objective. Hence, DrAC benefits from the regularizing effect of using data augmentation, while mitigating adverse consequences on the RL objective. Hence, DrAC benefits from the regularizing effect of using data augmentation, while mitigating adverse consequences on the RL objective.

Algorithm 1 DrAC: **D**ata-**r**egularized **A**ctor-**C**ritic applied to PPO
Black: unmodified actor-critic algorithm.
Cyan: image transformation.
Red: policy regularization.
Blue: value function regularization.

```
1:  Hyperparameters: image transformation $f$, regularization loss coefficient $\alpha_r$, minibatch size
    M, replay buffer size T, number of updates K.
2:  for $k = 1, \dots, K$ do
3:      Collect a new set of transitions $\mathcal{D} = \{(s_i, a_i, r_i, s_{i+1})\}_{i=1}^{T}$ using $\pi_\theta$.
4:      for $j = 1, \dots, *\frac{T}{M}$ do
5:          $\{(s_i, a_i, r_i, s_{i+1})\}_{i=1}^{M} \sim \mathcal{D}$                 ▹ Sample a minibatch of transitions
6:          for $i = 1, \dots, M$ do
7:              $\nu_i \sim \mathcal{H}$                                   ▹ Sample the augmentation parameters
8:              $\hat{\pi}_i \leftarrow \pi_\phi(\cdot|s_i)$                          ▹ Compute the policy targets
9:              $\hat{V}_i \leftarrow V_\phi(s_i)$                            ▹ Compute the value function targets
10:         end for
11:         $G_\pi(\theta) = \frac{1}{M}\sum_{i=1}^{M} KL\left[\hat{\pi}_i \mid \pi_\theta(\cdot|f(s_i, \nu_i))\right]$        ▹ Regularize the policy
12:         $G_V(\phi) = \frac{1}{M}\sum_{i=1}^{M}\left(\hat{V}_i - V_\phi\left(f(s_i, \nu_i)\right)\right)^2$        ▹ Regularize the value function
13:         $J_{\mathrm{DrAC}}(\theta, \phi) = J_{\mathrm{PPO}}(\theta, \phi) - \alpha_r(G_\pi(\theta) + G_V(\phi))$   ▹ Compute the total loss function
14:         $\theta \leftarrow_\theta J_{\mathrm{DrAC}}$                                      ▹ Update the policy
15:         $\phi \leftarrow_\phi J_{\mathrm{DrAC}}$                              ▹ Update the value function
16:     end for
17: end for
```

3.4.2 Automatic Data Augmentation

Since different tasks benefit from different types of transformations, we would like to design a method that can automatically find an effective transformation for any given task. Such a technique would significantly reduce the computational requirements for applying data augmentation in RL. In this section, we describe three approaches for doing this. In all of them, the augmentation learner is trained at the same time as the agent learns to solve the task using DrAC. Hence, the distribution of rewards varies significantly as the agent improves, making the problem highly nonstationary.

Upper Confidence Bound. The problem of selecting a data augmentation from a given set can be formulated as a multi-armed bandit problem, where the action space is the set of available transformations $\mathcal{F} = \{f^1, \ldots, f^n\}$. A popular algorithm for such settings is the upper confidence bound or UCB [Auer 2002], which selects actions according to the following policy:

$$f_t = \text{argmax}_{f \in \mathcal{F}} \left[Q_t(f) + c\sqrt{\frac{\log(t)}{N_t(f)}} \right], \tag{3.5}$$

where f_t is the transformation selected at time step t, $N_t(f)$ is the number of times transformation f has been selected before time step t and c is UCB's exploration coefficient. Before the t-th DrAC update, we use equation (3.5) to select an augmentation f. Then, we use equation (3.4) to update the agent's policy and value function. We also update the counter: $N_t(f) = N_{t-1}(f) + 1$. Next, we collect rollouts with the new policy and update the Q-function: $Q_t(f) = \frac{1}{K} \sum_{i=t-K}^{t} \mathcal{R}(f_i = f)$, which is computed as a sliding window average of the past K episodes after using augmentation f to train the agent. We refer to this approach as **UCB-DrAC** (Algorithm 2). Note that UCB-DrAC's estimation of $Q(f)$ differs from that of a typical UCB algorithm which uses rewards from the entire history. However, the choice of estimating $Q(f)$ using only more recent rewards is crucial due to the nonstationarity of the problem.

Algorithm 2 UCB-DrAC

```
1:  Hyperparameters: Set of image transformations $\mathcal{F} = \{f^1, \dots, f^n\}$, exploration coefficient c, window for estimating the Q-functions W, number of updates K, initial policy parameters $\pi_\theta$, initial value function $V_\phi$.
2:  $N(f) = 1,\ \forall f \in \mathcal{F}$    ▷ Initialize the number of times each augmentation was selected
3:  $Q(f) = 0,\ \forall f \in \mathcal{F}$    ▷ Initialize the Q-functions for all augmentations
4:  $R(f) = \text{FIFO}(W),\ \forall f \in \mathcal{F}$    ▷ Initialize the lists of returns for all augmentations
5:  for $k = 1, \dots, K$ do
6:      $f_k = \text{argmax}_{f \in \mathcal{F}} \left[ Q(f) + c \sqrt{\frac{\log(k)}{N(f)}} \right]$    ▷ Use UCB to select an augmentation
7:      Update the policy and value function according to Algorithm 1 with $f = f_k$ and $K = 1$:
8:      $\theta \leftarrow_\theta J_{\text{DrAC}}$    ▷ Update the policy
9:      $\phi \leftarrow_\phi J_{\text{DrAC}}$    ▷ Update the value function
10:     Compute the mean return obtained by the new policy $r_k$.
11:     Add $r_k$ to the $R(f_k)$ list using the first-in-first-out rule.
12:     $Q(f_k) \leftarrow \frac{1}{|R(f_k)|} \sum_{r \in R(f_k)} r$
13:     $N(f_k) \leftarrow N(f_k) + 1$
14: end for
```

Meta-Learning the Selection of an Augmentation. Alternatively, the problem of selecting a data augmentation from a given set can be formulated as a meta-learning problem. Here, we consider a meta-learner like the one proposed by Wang et al. [2016b]. Before each DrAC update, the meta-learner selects an augmentation, which is then used to update the agent using equation (3.4). We then collect rollouts using the new policy and update the meta-learner using the mean return of these trajectories. We refer to this approach as **RL2-DrAC**.

Meta-Learning the Weights of an Augmentation. Another approach for automatically finding an appropriate augmentation is to directly learn the weights of a certain transformation rather than selecting an augmentation from a given set. In this work, we focus on meta-learning the weights of a convolutional network which can be applied to the observations to obtain a perturbed image. We meta-learn the weights of this network using an approach similar to the one proposed by Finn et al. [2017]. For each agent update, we also perform a meta-update of the transformation function by splitting DrAC's buffer into meta-train and meta-test sets. We refer to this approach as **Meta-DrAC**.

3.5 Experiments

In this section, we evaluate our methods on **four DeepMind Control environments with natural and synthetic distractors** and the **full Procgen benchmark** [Cobbe et al. 2019b] which consists of 16 procedurally generated games (see Figure 3.2). Procgen has a number of attributes that make it a good testbed for generalization in RL: (i) it has a diverse set of games in a similar spirit with the ALE benchmark [Bellemare et al. 2013], (ii) each of these games has procedurally generated levels which present agents with meaningful generalization challenges, (iii) agents have to learn motor control directly from images, and (iv) it has a clear protocol for testing generalization.

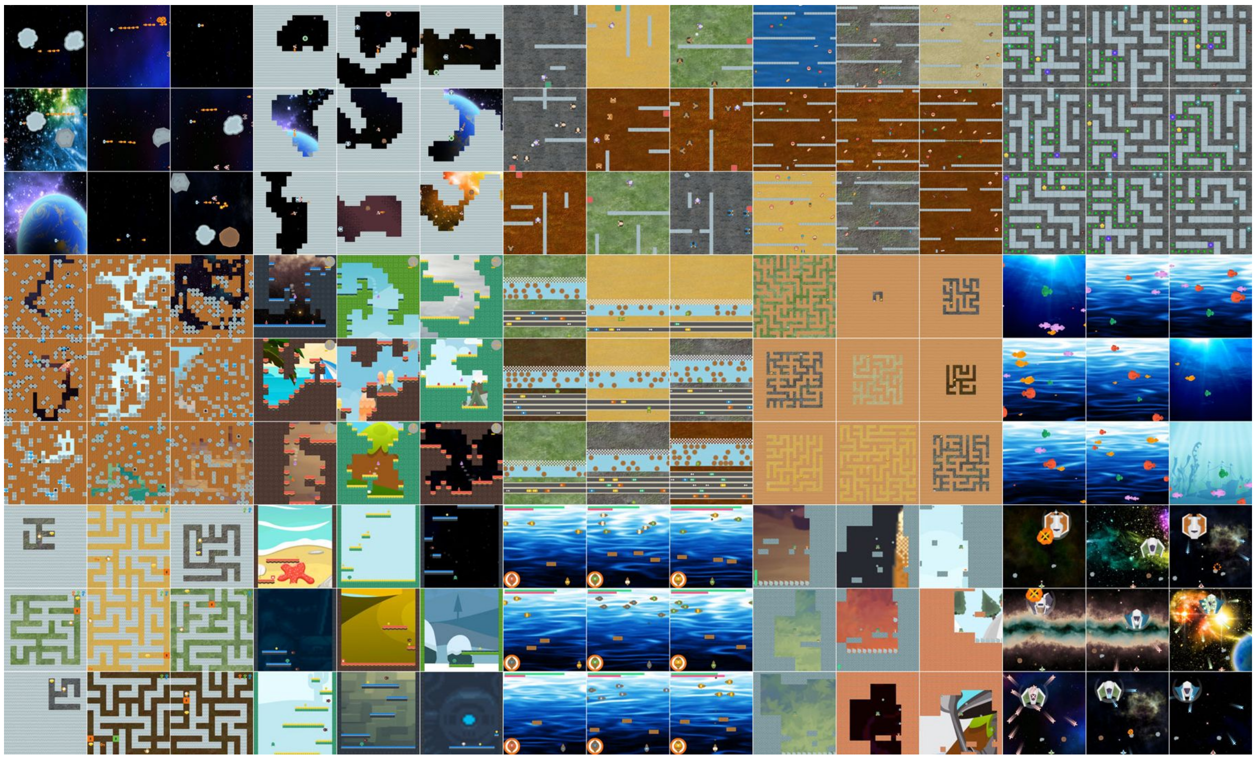


Figure 3.2: Screenshots of multiple procedurally-generated levels from 15 Procgen environments: StarPilot, CaveFlyer, Dodgeball, FruitBot, Chaser, Miner, Jumper, Leaper, Maze, BigFish, Heist, Climber, Plunder, Ninja, BossFight (from left to right, top to bottom).

All environments use a discrete 15 dimensional action space and produce $64 \times 64 \times 3$ RGB observations. We use Procgen's *easy* setup, so for each game, agents are trained on 200 levels and tested on the full distribution of levels. We use PPO as a base for all our methods.

Data Augmentation. In our experiments, we use a set of eight transformations: *crop, grayscale, cutout, cutout-color, flip, rotate, random convolution* and *color-jitter* [Krizhevsky et al. 2012; DeVries and Taylor 2017]. We use **RAD**'s [Laskin et al. 2020] implementation of these transformations, except for *crop*, in which we pad the image with 12 (boundary) pixels on each side and select random crops of 64×64. We found this implementation of *crop* to be significantly better on Procgen, and thus it can be considered an empirical upper bound of RAD in this case. For simplicity, we will refer to our implementation as RAD. **DrAC** uses the same set of transformations as RAD, but is trained with additional regularization losses for the actor and the critic, as described in Section 3.4.1.

Automatic Selection of Data Augmentation. We compare three different approaches for automatically finding an effective transformation: **UCB-DrAC** which uses UCB [Auer 2002] to select an augmentation from a given set, **RL2-DrAC** which uses RL^2 [Wang et al. 2020] to do the same, and **Meta-DrAC** which uses MAML [Finn et al. 2017] to meta-learn the weights of a convolutional network. Meta-DrAC is implemented using the *higher* library [Grefenstette et al. 2019]. Note that we do not expect these approaches to be better than DrAC with the best augmentation. In fact, DrAC with the best augmentation can be considered to be an upper bound for these automatic approaches since it uses the best augmentation during the entire training process.

Ablations. **DrC** and **DrA** are ablations to **DrAC** that use only the value or only the policy regularization terms, respectively. DrC can be thought of an analogue of DrQ [Kostrikov et al. 2020] applied to PPO rather than SAC. **Rand-DrAC** uses a uniform distribution to select an augmentation each time. **Crop-DrAC** uses crop for all games (which is the most effective augmentation on half of the Procgen games). **UCB-RAD** combines UCB with RAD (*i.e.* it does not use the regularization terms).

Baselines. We also compare with **Rand-FM** [Lee et al. 2020], **IBAC-SNI** [Igl et al. 2019], **Mixreg** [Wang et al. 2016b], and **PLR** [Jiang et al. 2020], four methods specifically designed for improving generalization in RL and previously tested on Procgen environments. Rand-FM uses a random convolutional networks to regularize the learned representations, while IBAC-SNI uses an information bottleneck with selective noise injection. Mixreg uses mixtures of observations to impose linearity constraints between the agent's inputs and the outputs, while PLR samples levels according to their learning potential.

Evaluation Metrics. At the end of training, for each method and each game, we compute the average score over 100 episodes and 10 different seeds. The scores are then normalized using the corresponding PPO score on the same game. We aggregate the normalized scores over all 16 Procgen games and report the resulting mean, median, and standard deviation (Table 3.1).

3.5.1 Generalization Performance on Procgen

Table 3.1 shows train and test performance on Procgen. UCB-DrAC significantly outperforms PPO, Rand-FM, IBAC-SNI, PLR, and Mixreg. As shown in Jiang et al. [2020], combining PLR with UCB-DrAC achieves a new state-of-the-art on Procgen leading to a 76% gain over PPO. Regularizing the policy and value function leads to improvements over merely using data augmentation, and thus the performance of DrAC is better than that of RAD (both using the best augmentation for each game). In addition, we demonstrate the importance of regularizing both the policy and the value function rather than either one of them by showing that DrAC is superior to both DrA and DrC. Our experiments show that the most effective way of automatically finding an augmentation is UCB-DrAC. As expected, meta-learning the weights of a CNN using Meta-DrAC performs reasonably well on the games in which the random convolution augmentation helps. But overall, Meta-DrAC and RL2-DrAC are worse than UCB-DrAC. In addition, UCB is generally more stable, easier to implement, and requires less fine-tuning compared to meta-learning algorithms. See Figures 3.3 and 3.4 for a comparison of these methods on each Procgen game. Moreover,

Table 3.1: Train and test performance for the Procgen benchmark (aggregated over all 16 tasks, 10 seeds). (a) compares PPO with four baselines specifically designed to improve generalization in RL and shows that they do not significantly help. (b) compares using the best augmentation from our set with and without regularization, corresponding to DrAC and RAD respectively, and shows that regularization improves performance on both train and test. (c) compares different approaches for automatically finding an augmentation for each task, namely using UCB or RL^2 for selecting the best transformation from a given set, or meta-learning the weights of a convolutional network (Meta-DrAC). (d) shows additional ablations: DrA regularizes only the actor, DrC regularizes only the critic, Rand-DrAC selects an augmentation using a uniform distribution, Crop-DrAC uses image crops for all tasks, and UCB-RAD is an ablation that does not use the regularization losses. UCB-DrAC performs best on both train and test, and achieves a return comparable with or better than DrAC (which uses the best augmentation).

		PPO-Normalized Return (%)					
		Train			**Test**		
	Method	**Median**	**Mean**	**Std**	**Median**	**Mean**	**Std**
(a)	PPO	100.0	100.0	7.2	100.0	100.0	8.5
	Rand-FM	93.4	87.6	8.9	91.6	78.0	9.0
	IBAC-SNI	91.9	103.4	8.5	86.2	102.9	8.6
	Mixreg	95.8	104.2	3.1	105.9	114.6	3.3
	PLR	101.5	106.7	5.6	107.1	128.3	5.8
(b)	**DrAC (Best) (Ours)**	**114.0**	**119.6**	9.4	118.5	138.1	10.5
	RAD (Best)	103.7	109.1	9.6	114.2	131.3	9.4
(c)	**UCB-DrAC (Ours)**	102.3	118.9	8.8	**118.5**	**139.7**	8.4
	RL2-DrAC	96.3	95.0	8.8	99.1	105.3	7.1
	Meta-DrAC	101.3	100.1	8.5	101.7	101.2	7.3
(b)	DrA (Best)	102.6	117.7	11.1	110.8	126.6	9.0
	DrC (Best)	103.3	108.2	10.8	110.6	115.4	8.5
	Rand-DrAC	100.4	99.5	8.4	102.4	103.4	7.0
	Crop-DrAC	97.4	112.8	9.8	114.0	132.7	11.0
	UCB-RAD	100.4	104.8	8.4	103.0	125.9	9.5

automatically selecting the augmentation from a given set using UCB-DrAC performs similarly well or even better than a method that uses the best augmentation for each task throughout the entire training process. UCB-DrAC also achieves higher returns than an ablation that uses a uniform distribution to select an augmentation each time, Rand-DrAC. Nevertheless, UCB-DrAC is better than Crop-DrAC, which uses crop for all the games (which is the best augmentation for 8 out of all 16 Procgen games).

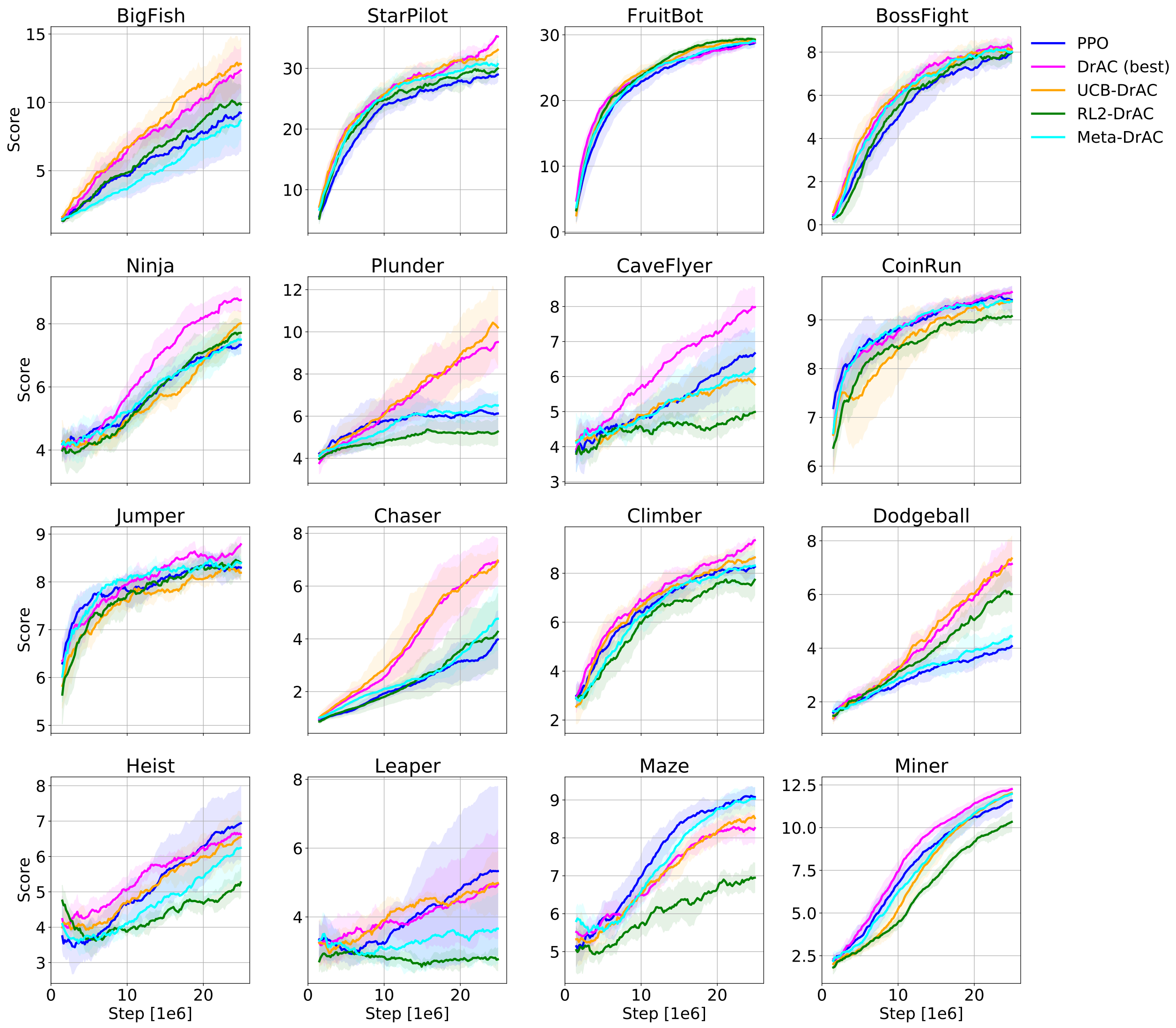


Figure 3.3: Train performance of various approaches that automatically select an augmentation, namely UCB-DrAC, RL2-DrAC, and Meta-DrAC. The mean and standard deviation are computed using 10 runs.

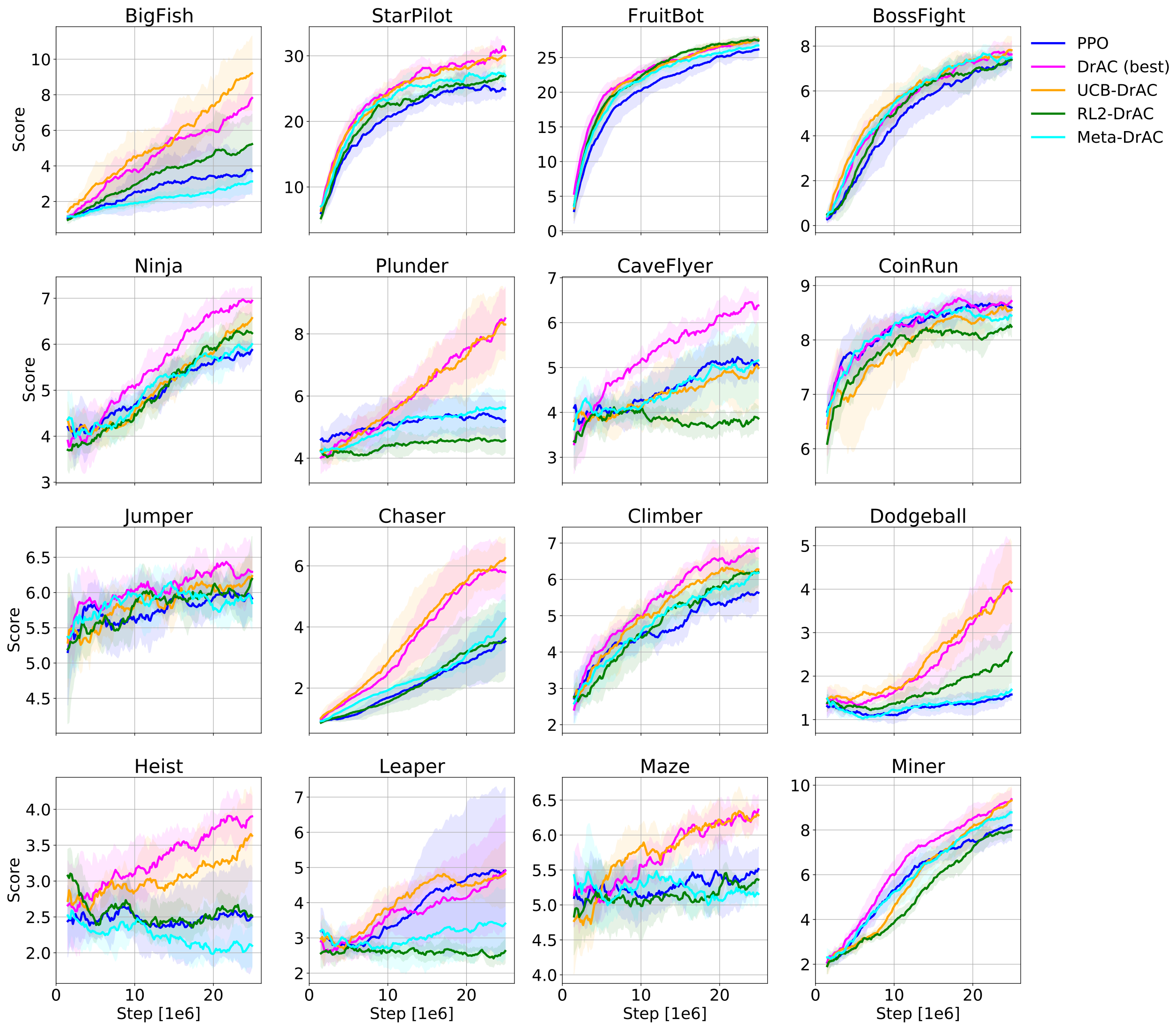


Figure 3.4: Test performance of various approaches that automatically select an augmentation, namely UCB-DrAC, RL2-DrAC, and Meta-DrAC. The mean and standard deviation are computed using 10 runs.

3.5.2 DeepMind Control with Distractors

In this section, we evaluate our approach on the DeepMind Control Suite from pixels (DMC, Tassa et al. [2018a]). We use four tasks, namely Cartpole Balance, Finger Spin, Walker Walk, and Cheetah Run, in two settings with different types of backgrounds, namely the *simple* distractors and the *natural* videos from the Kinetics dataset [Kay et al. 2017], as introduced in Zhang et al. [2020b]. See Figure 3.5 for a few examples of these environments.

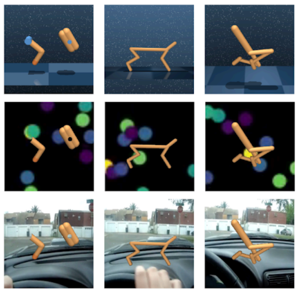


Figure 3.5: DMC environment examples. Top row: default backgrounds without any distractors. Middle row: simple distractor backgrounds with ideal gas videos. Bottom row: natural distractor backgrounds with Kinetics videos. Tasks from left to right: Finger Spin, Cheetah Run, Walker Walk.

Note that in the simple and natural settings, the background is sampled from a list of videos at the beginning of each episode, which creates spurious correlations between the backgrounds and the rewards. As shown in Figures 3.6 and 3.7, in the simple and natural distractor settings, UCB-DrAC outperforms PPO and RAD on all these environments.

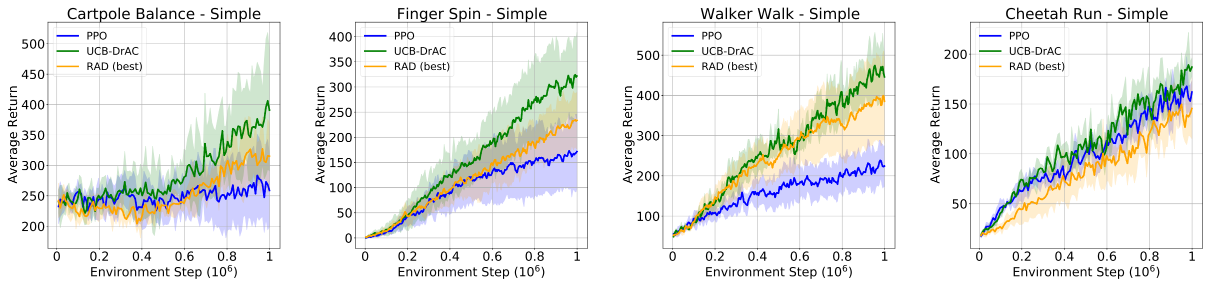


Figure 3.6: Average return on DMC tasks with simple (*i.e.* synthetic) distractor backgrounds with mean and standard deviation computed over 5 seeds. UCB-DrAC outperforms PPO and RAD with the best augmentations.

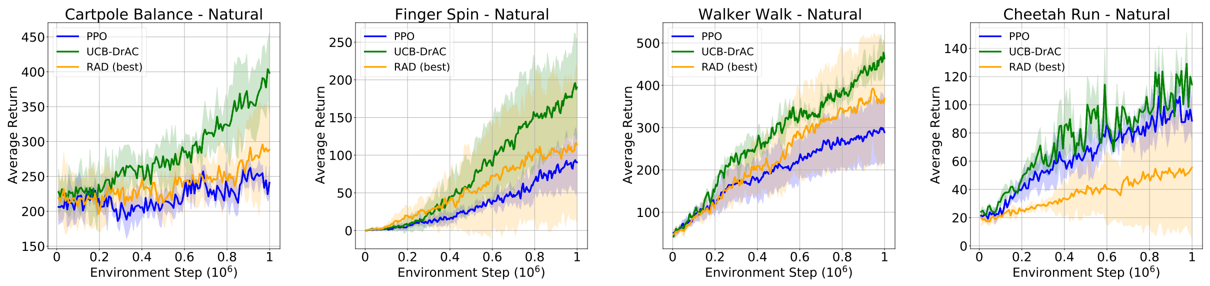


Figure 3.7: Average return on DMC tasks with natural video backgrounds with mean and standard deviation computed over 5 seeds. UCB-DrAC outperforms PPO and RAD with the best augmentations.

3.5.3 Regularization Effect

In Section 3.3.1, we argued that additional regularization terms are needed in order to make the use of data augmentation in RL theoretically sound. However, one might wonder if this problem actually appears in practice. Thus, we empirically investigate the effect of regularizing the policy and value function. For this purpose, we compare the performance of RAD and DrAC with grayscale and random convolution augmentations on Chaser, Miner, and StarPilot.

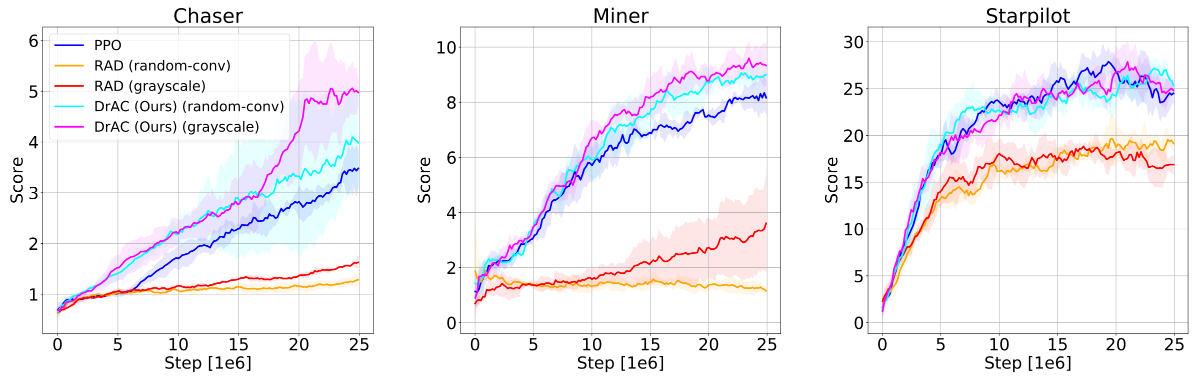


Figure 3.8: Comparison between RAD and DrAC with the same augmentations, grayscale and random convolution, on the test environments of Chaser (left), Miner (center), and StarPilot (right). While DrAC's performance is comparable or better than PPO's, not using the regularization terms, *i.e.* using RAD, significantly hurts performance relative to PPO. This is because, in contrast to DrAC, RAD does not use a principled (importance sampling) estimate of PPO's objective.

Figure 3.8 shows that not regularizing the policy and value function with respect to the transformations used can lead to drastically worse performance than vanilla RL methods, further emphasizing the importance of these loss terms. In contrast, using the regularization terms as part of the RL objective (as DrAC does) results in an agent that is comparable or, in some cases, significantly better than PPO.

3.5.4 Automatic Augmentation

Our experiments indicate there is not a single augmentation that works best across all Procgen games. Moreover, our intuitions regarding the best transformation for each game might be misleading. For example, at a first sight, Ninja appears to be somewhat similar to Jumper, but the augmentation that performs best on Ninja is color-jitter, while for Jumper is random-conv. In contrast, Miner seems like a different type of game than Climber or Ninja, but they all have the same best performing augmentation, namely color-jitter. These observations further underline the need for a method that can automatically find the right augmentation for each task.

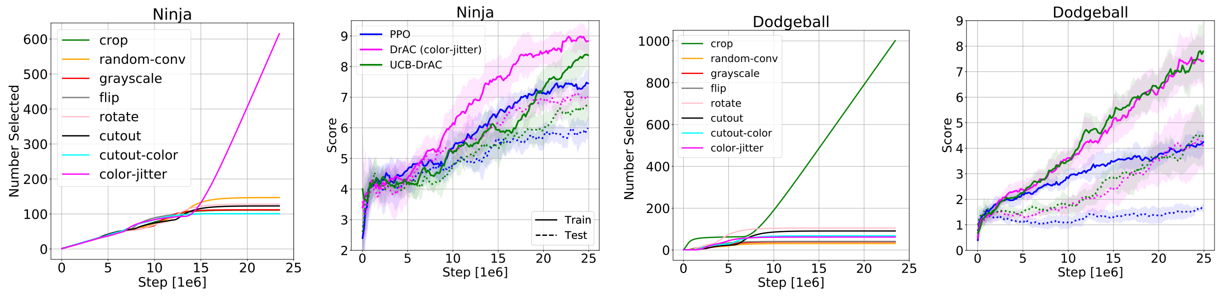


Figure 3.9: Cumulative number of times UCB selects each augmentation over the course of training for Ninja (a) and Dodgeball (c). Train and test performance for PPO, DrAC with the best augmentation for each game (color-jitter and crop, respectively), and UCB-DrAC for Ninja (b) and Dodgeball (d). UCB-DrAC finds the most effective augmentation from the given set and reaches the performance of DrAC. Our methods improve both train and test performance.

3.5.5 Robustness Analysis

To further investigate the generalizing ability of these agents, we analyze whether the learned policies and state representations are invariant to changes in the observations which are irrelevant for solving the task.

Table 3.2: JSD and Cycle-Consistency (%) (aggregated across all Procgen tasks) for PPO, RAD and UCB-DrAC, measured between observations that vary only in their background themes (*i.e.* colors and patterns that do not interact with the agent). UCB-DrAC learns more robust policies and representations that are more invariant to changes in the observation that are irrelevant for the task.

			Cycle-Consistency (%)			
	JSD		**2-way**		**3-way**	
Method	**Mean**	**Median**	**Mean**	**Median**	**Mean**	**Median**
PPO	0.25	0.23	20.50	18.70	12.70	5.60
RAD	0.19	0.18	24.40	22.20	15.90	8.50
UCB-DrAC	**0.16**	**0.15**	**27.70**	**24.80**	**17.30**	**10.30**

Table 3.1 along with Figure 3.4 compare different approaches for automatically finding an augmentation, showing that UCB-DrAC performs best and reaches the asymptotic performance obtained when the most effective transformation for each game is used throughout the entire training process. Figure 3.9 illustrates an example of UCB's policy during training on Ninja and Dodgeball, showing that it converges to always selecting the most effective augmentation, namely color-jitter for Ninja and crop for Dodgeball.

We first measure the Jensen-Shannon divergence (JSD) between the agent's policy for an observation from a training level and a modified version of that observation with a different background theme (*i.e.* color and pattern). Note that the JSD also represents a lower bound for the joint empirical risk across train and test [Ilse et al. 2020]. The background theme is randomly selected from the set of backgrounds available for all other Procgen environments, except for the one of the original training level. Note that the modified observation has the same semantics as the original one (with respect to the reward function), so the agent should have the same policy in both cases. Moreover, many of the backgrounds are not uniform and can contain items such as trees or planets which can be easily misled for objects the agent can interact with. As seen in Table 3.2, UCB-DrAC has a lower JSD than PPO, indicating that it learns a policy that is more robust to changes in the background.

To quantitatively evaluate the quality of the learned representation, we use the cycle-consistency metric proposed by Aytar et al. [2018] and also used by Lee et al. [2020]. Table 3.2 reports the percentage of input observations in the seen environment that are cycle-consistent with trajectories in modified unseen environments, which have a different background but the same layout. UCB-DrAC has higher cycle-consistency than PPO, suggesting that it learns representations that better capture relevant task invariances.

3.6 Conclusion

In this chapter, we introduced UCB-DrAC, a method for automatically finding an effective data augmentation for RL tasks. This approach enables the principled use of data augmentation with actor-critic algorithms by regularizing the policy and value functions with respect to state transformations. As shown here, UCB-DrAC avoids the theoretical and empirical pitfalls typical in naive applications of data augmentation in RL. Our approach improves training performance by 19% and test performance by 40% on the Procgen benchmark, thus setting a new state-of-the-art at the time of release. In addition, the learned policies and representations are more invariant to spurious correlations between observations and rewards. One promising avenue for future research is to use a more expressive function class for meta-learning the transformations in order to capture a wider range of inductive biases.

Our work was the first to show the benefits of automatically selecting a type of data augmentation for improving generalization in reinforcement learning. We were also the first to demonstrate strong results on the entire Procgen benchmark, and thus UCB-DrAC has been widely used as a baseline in future papers [Jiang et al. 2020; Fan and Li 2021; Raileanu and Fergus 2021]. Since the release of our work, other types of data augmentation have demonstrated generalization gains in RL, such as the use of soft augmentations to decouple representation from policy learning [Hansen et al. 2019], the use of convex combinations of the input observations [Wang et al. 2020], or the distillation from an expert with weak augmentations to a student with strong augmentations [Fan et al. 2021]. Data augmentation has also been used in offline reinforcement learning for improving generalization to new environment dynamics [Ball et al. 2021]. To learn more general policies which are less sensitive to task-irrelevant features, Fan and Li [2021] proposed to use multi-view information bottlenecks, while Agarwal et al. [2021] proposed to learn representations which capture the behavioral similarity among states. A number of works have shown that combining data augmentation with other complementary approaches, such as automatic curricula [Jiang et al. 2020] or state abstractions [Agarwal et al. 2021], can lead to additional improvements. In particular, our method UCB-DrAC combined with Prioritized Level Replay [Jiang et al. 2020] improves test performance on the hard setting of Procgen by 87% relative to PPO. Finally, Ko and Ok [2021] have studied the effect of *when* data augmentation is being used during the training process and showed that the optimal timing can depend on the task and augmentation used.

4 | Decoupling Value and Policy for Generalization in Reinforcement Learning

In the previous chapter, we discussed how standard deep reinforcement learning agents can be brittle to changes in their observation which shouldn't affect their behavior, such as different colors or backgrounds. We also showed that constraining the policy and value function to be invariant to certain transformations can result in more robust agents. In this chapter, we study a different problem with standard deep reinforcement learning methods which can lead to overfitting. When training RL agents, particularly from high-dimensional observations such as images, it is common practice to use a shared representation for the policy and value function. However, we argue that more information is needed to accurately estimate the true value function than to learn the optimal policy. Consequently, the use of a shared representation for the policy and value function can lead to overfitting. To alleviate this problem, in this chapter we introduce two approaches which are combined to create IDAAC: Invariant Decoupled Advantage Actor-Critic. First, IDAAC decouples the optimization of the policy and value function, using separate networks to model them. Second, it introduces an auxiliary loss which encourages the representation to be invariant to task-irrelevant properties of the environment. IDAAC shows good generalization to unseen environments, achieving a new state-of-the-art on the Procgen benchmark and outperforming popular methods on DeepMind Control tasks with distractors.

4.1 Introduction

Here we consider the problem of generalizing to unseen instances (or levels) of procedurally generated environments, after training on a relatively small number of such instances. While the high-level goal is the same, the background, dynamics, layouts, as well as the locations, shapes, and colors of various entities, differ across instances.

In this work, we identify a new factor that leads to overfitting in such settings, namely the use of a shared representation for the policy and value function. We point out that accurately estimating the value function requires instance-specific features in addition to the information needed to learn the optimal policy. When training a common network for the policy and value function (as is currently standard practice in pixel-based RL), the need for capturing level-specific features in order to estimate the value function can result in a policy that does not generalize well to new task instances.

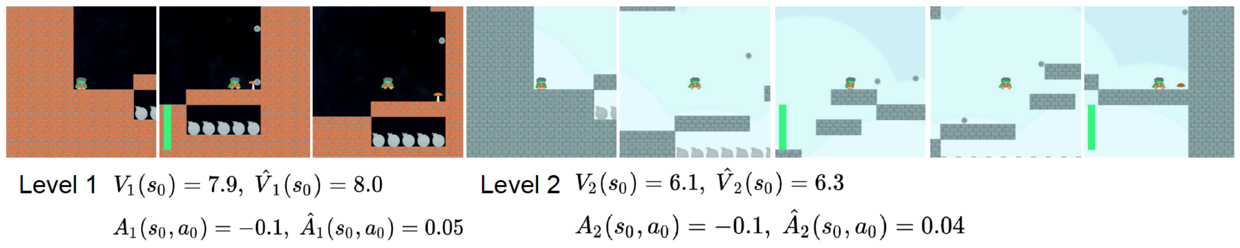

Level 1 $V_1(s_0) = 7.9,\ \hat{V}_1(s_0) = 8.0$
$A_1(s_0, a_0) = -0.1,\ \hat{A}_1(s_0, a_0) = 0.05$
Level 2 $V_2(s_0) = 6.1,\ \hat{V}_2(s_0) = 6.3$
$A_2(s_0, a_0) = -0.1,\ \hat{A}_2(s_0, a_0) = 0.04$

Figure 4.1: Policy-Value Asymmetry. Two Ninja levels with initial observations that are *semantically identical but visually different*. Level 1 (first three frames from the left with black background) is much shorter than Level 2 (last six frames with blue background). Both the true and the estimated values (by a PPO agent trained on 200 levels) of the initial observation are higher for Level 1 than for Level 2 *i.e.* $V_1(s_0) > V_2(s_0)$ and $\hat{V}_1(s_0) > \hat{V}_2(s_0)$. Thus, to accurately predict the value function, the representations must capture level-specific features (such as the backgrounds), which are irrelevant for finding the optimal policy. Consequently, using a common representation for both the policy and value function can lead to overfitting to spurious correlations and poor generalization to unseen levels. In contrast to the value, the true advantage of the initial states and noop action has the same values for the two levels, and the advantages predicted by the agent also have very similar values.

4.1.1 Policy-Value Representation Asymmetry

To illustrate this phenomenon, which we call the *policy-value representation asymmetry*, consider the example in Figure 4.1 which shows two different levels from the Procgen game Ninja [Cobbe et al. 2019b]. The first observations of the two levels are semantically identical and could be represented using the same features (describing the locations of the agent, the bombs, and the platform, while disregarding the background patterns and wall colors). An optimal agent should take the same action in both levels, namely that of moving to the right to further explore the level and move closer to the goal (which is always to its right in this game). However, these two observations have different values. Note that Level 1 is much shorter than Level 2, and the levels look quite different from each other after the initial part. A standard RL agent (*e.g.* PPO [Schulman et al. 2017]) completes the levels in 24 and 50 steps, respectively. The true value of the initial observation is the expected return (*i.e.* sum of discounted rewards) received during the episode. In this game, the agent receives a reward of 10 when it reaches the goal and 0 otherwise. Hence, the true value is higher for Level 1 than for Level 2 since the reward is discounted only for 24 steps rather than 50. In order to accurately estimate the value of an observation (which is part of the objective function for many popular RL methods), the agent must memorize the number of remaining steps in that particular level (which for the initial observation is equivalent to the episode's length). To do this, the agent must use instance-specific features such as the background (which can vary across a level so that each observation within a level has a slightly different background pattern). For the case shown in Figure 4.1, the agent can learn to associate a black background with a higher value than a blue background. But this is only a spurious correlation since the background has no causal effect on the state's value, unlike the agent's position relative to items it can interact with. If an agent uses a shared representation for learning the policy and value function, capturing such spurious correlations can result in policies that do not generalize well to new instances, similar to what happens in supervised learning [Arjovsky et al. 2019].

Furthermore, if the environment is partially observed, an agent should have no way of predicting its expected return in a new level. At the same time, the agent could still select the optimal action if it has learned good state representations (*i.e.* that capture the minimal set of features needed to act in the environment in order to solve the task). Thus, in partially observed procedurally generated environments, accurately predicting the value function can require instance-specific features which are not necessary for learning the optimal policy.

To address the policy-value representation asymmetry, we propose **I**nvariant **D**ecoupled **A**dvantage **A**ctor-**C**ritic or **IDAAC** for short, which makes two algorithmic contributions. First, IDAAC decouples the policy and value optimization by using two separate networks to learn each of them. The policy network has two heads, one for the policy and one for the generalized advantage function. The value network is needed to compute the advantage, which is used both for the policy-gradient objective and as a target for the advantage predictions. Second, IDAAC uses an auxiliary loss which constrains the policy representation to be invariant to the task instance.

To summarize, our work makes the following contributions: (i) identifies that using a shared representation for the policy and value function can lead to overfitting in RL; (ii) proposes a new approach that uses separate networks for the policy and value function while still learning effective behaviors; (iii) introduces an auxiliary loss for encouraging representations to be invariant to the task instance, and (iv) demonstrates state-of-the-art generalization on the Procgen benchmark and outperforms popular RL methods on DeepMind Control tasks with distractors.

4.2 Related Work

Generalization in Deep RL. See Section 3.2 for a discussion of the literature on generalization in deep RL. Since the release of that work, other approaches have been proposed. For example, Chen [2020] use surprise minimization, Bengio et al. [2020] study the link between generalization and interference in TD-learning, while Bertrán et al. [2020] prove that agents trained with off-policy actor-critic methods overfit to their training instances, which is consistent with our empirical results. However, none of these works focus on the asymmetry between the optimal policy and value representation.

Decoupling the Policy and Value Function. While the current standard practice in deep RL from pixels is to share parameters between the policy and value function in order to learn good representations and reduce computational complexity [Mnih et al. 2016a; Silver et al. 2017; Schulman et al. 2017], a few papers have explored the idea of decoupling the two for improving sample efficiency [Barth-Maron et al. 2018; Pinto et al. 2018; Yarats et al. 2019; Andrychowicz et al. 2020; Cobbe et al. 2020]. In contrast with prior work, our work focuses on generalization to unseen environments and is the first one to point out that using shared features for the policy and value functions can lead to overfitting to spurious correlations. Most similar to our work, Cobbe et al. [2020] aim to alleviate the interference between policy and value optimization, but there are some key differences between their approach and ours. In particular, our method does not require an auxiliary learning phase for distilling the value function while constraining the policy, it does not use gradients from the value function to update the policy parameters, and it uses two auxiliary losses for training the policy network, one based on the advantage function and one that enforces invariance with respect to the environment instance. Prior work has also explored the idea of predicting the advantage in the context of Q-learning [Wang et al. 2016c], but this setting does not pose the same challenges since it does not learn policies directly.

4.3 Background

We use the setup from the previous chapter, as described in Section 3.3.

4.4 Approach

We use the setup from the previous chapter, as described in Section 3.3.

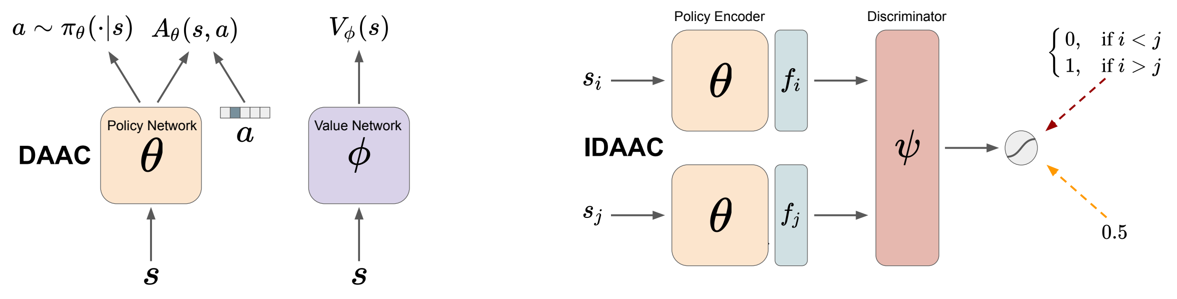


Figure 4.2: Overview of DAAC (left) and IDAAC (right). DAAC uses two separate networks, one for learning the policy and advantage, and one for learning the value. The value estimates are used to compute the advantage targets. IDAAC adds an additional regularizer to the DAAC policy encoder to ensure that it does not contain episode-specific information. The encoder is trained adversarially with a discriminator so that it cannot classify which observation from a given pair (s_i, s_j) was first in a trajectory.

We start by describing our first contribution, namely the **D**ecoupled **A**dvantage **A**ctor-**C**ritic (**DAAC**) algorithm, which uses separate networks for learning the policy and value function (Figure 4.2 (left), Section 4.4.2). Then, we extend this method by adding an auxiliary loss to constrain the policy representation to be invariant to the environment instance, which yields the **I**nvariant **D**ecoupled **A**dvantage **A**ctor-**C**ritic (**IDAAC**) algorithm (Figure 4.2 (right), Section 4.4.3).

4.4.1 Decoupling the Policy and Value Function

To address the problem of overfitting due to the coupling of the policy and value function, we propose a new actor-critic algorithm that uses separate networks for learning the policy and value function, as well as two auxiliary losses. A naive solution to the problem of parameter sharing between the policy and value function would be to simply train two separate networks. However, this approach is insufficient for learning effective behaviors because the policy network relies on gradients from the value function to learn useful features for the training environments. As shown by Cobbe et al. [2020], using separate networks for optimizing the policy and value function leads to drastically worse training performance than using a shared network, with no sign of progress on many of the tasks (see Figure 8 in their paper). Since such approaches cannot even learn useful behaviors for the training environments, they are no better on the test environments. These results indicate that without gradients from the value to update the policy network, the agent struggles to learn good behaviors. This is consistent with the fact that the gradients from the policy objective are notoriously sparse and high-variance, making training difficult, especially for high-dimensional state spaces (*e.g.* when learning from images). In contrast, the value loss can provide denser and less noisy gradients, leading to more efficient training. Given this observation, it is natural to investigate whether other auxiliary losses can provide similarly useful gradients for training the policy network, while also alleviating the problem of overfitting to spurious correlations.

4.4.2 Using the Advantage instead of the Value Function

As an alternative to the value function, we propose to predict the generalized advantage estimate (GAE) or advantage for short. As illustrated in Section 4.5.5, the advantage is less prone to overfitting to certain types of environment idiosyncrasies. Intuitively, the advantage is a measure of the expected additional return which can be obtained by taking a particular action relative to following the current policy. Because the advantage is a *relative* measure of an action's value while the value is an *absolute* measure of a state's value, the advantage can be expected to vary less with the number of remaining steps in the episode. Thus, the advantage is less likely to overfit to such instance-specific features. To learn generalizable representations, we need to fit a metric invariant to cosmetic changes in the observation which do not modify the underlying state. As shown in Figure 4.1, semantically identical yet visually distinct observations can have very different values but the same advantages (for a given action). This indicates that the advantage might be a good candidate to replace the value as an auxiliary loss for training the policy network.

In order to predict the advantage, we need an estimate of the value function which we obtain by simply training a separate network to output the expected return for a given state. Thus, our method consists of two separate networks, the value network parameterized by ϕ which is trained to predict the value function, and the policy network parameterized by θ which is trained to learn a policy that maximizes the expected return and also to predict the advantage function.

The policy network of DAAC is trained to maximize the following objective:

$$J_{\mathrm{DAAC}}(\theta) = J_\pi(\theta) + \alpha_{\mathrm{s}} \mathrm{S}_\pi(\theta) - \alpha_{\mathrm{a}} L_{\mathrm{A}}(\theta), \tag{4.1}$$

where $J_\pi(\theta)$ is the policy gradient objective, $S_\pi(\theta)$ is an entropy bonus to encourage exploration, $L_{\mathrm{A}}(\theta)$ is the advantage loss, while α_{s} and α_{a} are their corresponding weights determining each term's contribution to the total objective.

The policy objective term is the same as the one used by PPO (see Section 2):

$$J_\pi(\theta) = \hat{\mathbb{E}}_t \left[\min \left(r_t(\theta) \hat{A}_t,\ \operatorname{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right) \hat{A}_t \right) \right],$$

where $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$ and $\hat{A}_t$ is the advantage function at time step t.

The advantage function loss term is defined as:

$$L_{\mathrm{A}}(\theta) = \hat{\mathbb{E}}_t \left[\left(A_\theta(s_t, a_t) - \hat{A}_t \right)^2 \right],$$

where $\hat{A}_t$ is the corresponding generalized advantage estimate at time step t, $\hat{A}_t = \sum_{k=t}^{T} (\gamma\lambda)^{k-t} \delta_k$, with $\delta_t = r_t + \gamma V_\phi(s_{t+1}) - V_\phi(s_t)$ which is computed using the estimates from the value network.

The value network of DAAC is trained to minimize the following loss:

$$L_{\mathrm{V}}(\phi) = \hat{\mathbb{E}}_t \left[\left(V_\phi(s_t) - \hat{V}_t \right)^2 \right],$$

where $\hat{V}_t$ is the total discounted reward obtained during the corresponding episode after step t, $\hat{V}_t = \sum_{k=t}^{T} \gamma^{k-t} r_k$.

During training, we alternate between E_π epochs for training the policy network and E_V epochs for training the value network every N_π policy updates. See Algorithm 3 for a more detailed description of DAAC.

Algorithm 3 DAAC: Decoupled Advantage Actor-Critic

```
 1: Hyperparameters: Total number of updates N, replay buffer size T, number of epochs per
    policy update $E_\pi$, number of epochs per value update $E_V$, frequency of value updates $N_\pi$,
    weight for the advantage loss $\alpha_a$, initial policy parameters $\theta$, initial value parameters $\phi$.
 2: for $n = 1, \ldots, N$ do
 3:     Collect $\mathcal{D} = \{(s_t, a_t, r_t, s_{t+1})\}_{t=1}^{T}$ using $\pi(\theta)$.
 4:     Compute the value and advantage targets $\hat{V}_t$ and $\hat{A}_t$ for all states $s_t$
 5:     for $i = 1, \ldots, E_\pi$ do
 6:         $L_{\mathrm{A}}(\theta) = \hat{\mathbb{E}}_t \left[ \left( A_\theta(s_t, a_t) - \hat{A}_t \right)^2 \right]$      ▷ Compute the Advantage Loss
 7:         $J_{\mathrm{DAAC}}(\theta) = J_\pi(\theta) + \alpha_{\mathrm{s}} \mathrm{S}_\pi(\theta) - \alpha_{\mathrm{a}} L_{\mathrm{A}}(\theta)$      ▷ Compute the Policy Loss
 8:         $\theta \leftarrow_\theta J_{\mathrm{DAAC}}$      ▷ Update the Policy Network
 9:     end for
10:     if $n \% N_\pi = 0$ then
11:         for $j = 1, \ldots, E_V$ do
12:             $L_{\mathrm{V}}(\phi) = \hat{\mathbb{E}}_t \left[ \left( V_\phi(s_t) - \hat{V}_t \right)^2 \right]$      ▷ Compute the Value Loss
13:             $\phi \leftarrow_\phi L_{\mathrm{V}}$      ▷ Update the Value Network
14:         end for
15:     end if
16: end for
```

As our experiments show, predicting the advantage rather than the value provides useful gradients for the policy network so it can learn effective behaviors on the training environments, thus overcoming the challenges encountered by prior attempts at learning separate policy and value networks [Cobbe et al. 2020]. In addition, it mitigates the problem of overfitting caused by the use of value gradients to update the policy network, thus also achieving better performance on test environments.

4.4.3 Learning Instance-Invariant Features

From a generalization perspective, a good state representation is characterized by one that captures the minimum set of features necessary to learn the optimal policy and ignores instance-specific features which might lead to overfitting. As emphasized in Figure 4.1, due to the diversity of procedurally generated environments, the observations may contain information indicative of the number of remaining steps in the corresponding level. Since different levels have different lengths, capturing such information given only a partial view of the environment translates into capturing information specific to that level. Because such features overfit to the idiosyncrasies of the training environments, they can results in suboptimal policies on unseen instances of the same task.

Hence, one way of constraining the learned representations to be agnostic to the environment instance is to discourage them from carrying information about the number of remaining steps in the level. This can be formalized using an adversarial framework so that a discriminator cannot tell which observation from a given pair came first within an episode, based solely on their learned features. Similar ideas have been proposed for learning disentangled representations of videos [Denton and Birodkar 2017].

Let E_θ be an encoder that takes as input an observation s and outputs a feature vector f. This encoder is the same as the one used by the policy network so it is also parameterized by θ. Let D be a discriminator parameterized by ψ that takes as input two features f_i and f_j (in this order), corresponding to two observations from the same trajectory s_i and s_j, and outputs a number between 0 and 1 which represents the probability that observation s_i came before observation s_j. The discriminator is trained using a cross-entropy loss that aims to predict which observation was first in the trajectory:

$$\begin{aligned} L_{\mathrm{D}}(\psi) = &- \log \left[\mathrm{D}_{\psi}\left(\mathrm{E}_{\theta}(s_i), \mathrm{E}_{\theta}(s_j)\right)\right] \\ &- \log \left[1 - \mathrm{D}_{\psi}\left(\mathrm{E}_{\theta}(s_i), \mathrm{E}_{\theta}(s_j)\right)\right]. \end{aligned} \tag{4.2}$$

Note that only the discriminator's parameters are updated by minimizing the loss in eq. 4.2, while the encoder's parameters remain fixed during this optimization.

The other half of the adversarial framework imposes a loss function on the encoder that tries to maximize the uncertainty (*i.e.* entropy) of the discriminator regarding which observation was first in the episode:

$$\begin{aligned} L_{\mathrm{E}}(\theta) = &- \frac{1}{2}\log \left[\mathrm{D}_{\psi}\left(\mathrm{E}_{\theta}(s_i), \mathrm{E}_{\theta}(s_j)\right)\right] \\ &- \frac{1}{2}\log \left[1 - \mathrm{D}_{\psi}\left(\mathrm{E}_{\theta}(s_i), \mathrm{E}_{\theta}(s_j)\right)\right]. \end{aligned} \tag{4.3}$$

Similar to the above, only the encoder's parameters are updated by minimizing the loss in eq. 4.3, while the discriminator's parameters remain fixed during this optimization.

Thus, the policy network is encouraged to learn state representations so that the discriminator cannot identify whether a state came before or after another state. In so doing, the learned representations cannot carry information about the number of remaining steps in the environment, yielding features which are less instance-dependent and thus more likely to generalize outside the training distribution. Note that this adversarial loss is only used for training the policy network and not the value network.

To train the policy network, we maximize the following objective which combines the DAAC objective from eq. 4.1 with the above adversarial loss, resulting in IDAAC's objective:

$$J_{\mathrm{IDAAC}}(\theta) = J_{\mathrm{DAAC}}(\theta) - \alpha_i L_{\mathrm{E}}(\theta), \tag{4.4}$$

where α_i is the weight of the adversarial loss relative to the policy objective. Similar to DAAC, a separate value network is trained. See Algorithm 4 for a more detailed description of IDAAC.

Algorithm 4 IDAAC: Invariant Decoupled Advantage Actor-Critic

```
1: Hyperparameters: Total number of updates N, replay buffer size T, number of epochs per
   policy update $E_\pi$, number of epochs per value update $E_V$, frequency of value updates $N_\pi$,
   weight for the invariance loss $\alpha_i$, initial policy parameters $\theta$, initial value parameters $\phi$.
2: for $n = 1, \dots, N$ do
3:     Collect $\mathcal{D} = \{(s_t, a_t, r_t, s_{t+1})\}_{t=1}^{T}$ using $\pi(\theta)$.
4:     Compute the value and advantage targets $\hat{V}_t$ and $\hat{A}_t$ for all states $s_t$
5:     for $i = 1, \dots, E_\pi$ do
6:         $L_{\mathrm{A}}(\theta) = \hat{\mathbb{E}}_t\left[\left(A_\theta(s_t, a_t) - \hat{A}_t\right)^2\right]$    ▷ Compute the Advantage Loss
7:         $L_{\mathrm{E}}(\theta) = -\frac{1}{2}\log\left[\mathrm{D}_\psi\left(\mathrm{E}_\theta(s_i), \mathrm{E}_\theta(s_j)\right)\right] - \frac{1}{2}\log\left[1 - \mathrm{D}_\psi\left(\mathrm{E}_\theta(s_i), \mathrm{E}_\theta(s_j)\right)\right]$    ▷ Compute the Encoder Loss
8:         $J_{\mathrm{IDAAC}}(\theta, \phi, \psi) = J_\pi(\theta) + \alpha_{\mathrm{s}}\mathrm{S}_\pi(\theta) - \alpha_{\mathrm{a}}L_{\mathrm{A}}(\theta) - \alpha_i L_{\mathrm{E}}(\theta)$    ▷ Compute the Policy Loss
9:         $L_{\mathrm{D}}(\psi) = -\log\left[\mathrm{D}_\psi\left(\mathrm{E}_\theta(s_i), \mathrm{E}_\theta(s_j)\right)\right] - \log\left[1 - \mathrm{D}_\psi\left(\mathrm{E}_\theta(s_i), \mathrm{E}_\theta(s_j)\right)\right]$    ▷ Compute the Discriminator Loss
10:        $\theta \leftarrow_\theta J_{\mathrm{IDAAC}}$    ▷ Update the Policy Network
11:        $\psi \leftarrow_\psi L_{\mathrm{D}}$    ▷ Update the Discriminator
12:    end for
13:    if $n \ \% \ N_\pi = 0$ then
14:        for $j = 1, \dots, E_V$ do
15:            $L_{\mathrm{V}}(\phi) = \hat{\mathbb{E}}_t\left[\left(V_\phi(s_t) - \hat{V}_t\right)^2\right]$    ▷ Compute the Value Loss
16:            $\phi \leftarrow_\phi L_{\mathrm{V}}$    ▷ Update the Value Network
17:        end for
18:    end if
19: end for
```

We expect this inductive bias to be mostly useful in Markovian environments which do not require memory to solve the task and the optimal action can be determined given only the current observation (even if these are only partial views of the environment). In environments which require memory to be solved, we expect that it might be desirable for the representations to contain information regarding the order of the observations.

4.5 Experiments

In this section, we evaluate our methods on two distinct environments: (i) three DeepMind Control suite tasks with synthetic and natural background distrators [Zhang et al. 2020b] and (ii) the full Procgen benchmark [Cobbe et al. 2019b] which consists of 16 procedurally generated games. Procgen in particular has a number of attributes that make it a good testbed for generalization in RL: (i) it has a diverse set of games in a similar spirit with the ALE benchmark [Bellemare et al. 2013]; (ii) each of these games has procedurally generated levels which present agents with meaningful generalization challenges; (iii) agents have to learn motor control directly from images, and (iv) it has a clear protocol for testing generalization, the focus of our investigation.

All Procgen environments use a discrete 15 dimensional action space and produce $64 \times 64 \times 3$ RGB observations. We use Procgen's *easy* setup, so for each game, agents are trained on 200 levels and tested on the full distribution of levels.

4.5.1 Generalization Performance on Procgen

We compare DAAC and IDAAC with seven other RL algorithms: **PPO** [Schulman et al. 2017], **UCB-DrAC** [Raileanu et al. 2020], **PLR** [Jiang et al. 2020], **Mixreg** [Wang et al. 2020], **IBAC-SNI** [Igl et al. 2019], **Rand-FM** [Lee et al. 2020], and **PPG** [Cobbe et al. 2020]. UCB-DrAC is the previous state-of-the-art on Procgen and uses data augmentation to learn policy and value functions invariant to various input transformations PLR is a newer approach that uses an automatic curriculum based on the learning potential of each level and achieves strong results on Procgen. Rand-FM uses a random convolutional network to regularize the learned representations, IBAC-SNI uses an information bottleneck with selective noise injection, while Mixreg uses mixtures of observations to impose linearity constraints between the agent's inputs and outputs. All three were designed to improve generalization in RL and evaluated on Procgen games. Finally, PPG is the only method we are aware of that learns good policies while decoupling the optimization of the policy and value function. However, PPG was designed to improve sample efficiency rather than generalization and the method was evaluated only on Procgen's training distribution of environments.

Table 4.1: PPO-Normalized Procgen scores on train and test levels after training on 25M environment steps. Our approaches, DAAC and IDAAC, establish a new state-of-the-art on the test distribution of environments from the Procgen benchmark, while also showing strong training performance. The mean and standard deviation are computed using 10 runs with different seeds.

Score	RAND-FM	IBAC-SNI	Mixreg	PLR	UCB-DrAC	PPG	DAAC (Ours)	IDAAC (Ours)
Train	87.6 ± 8.9	103.4 ± 8.5	104.2 ± 3.1	106.7 ± 5.6	118.9 ± 8.0	**144.5 ± 5.7**	131.0 ± 6.1	132.2 ± 5.9
Test	78.0 ± 9.0	102.9 ± 8.6	114.6 ± 3.3	128.3 ± 5.8	139.7 ± 8.3	152.2 ± 5.8	162.3 ± 6.2	**163.7 ± 6.1**

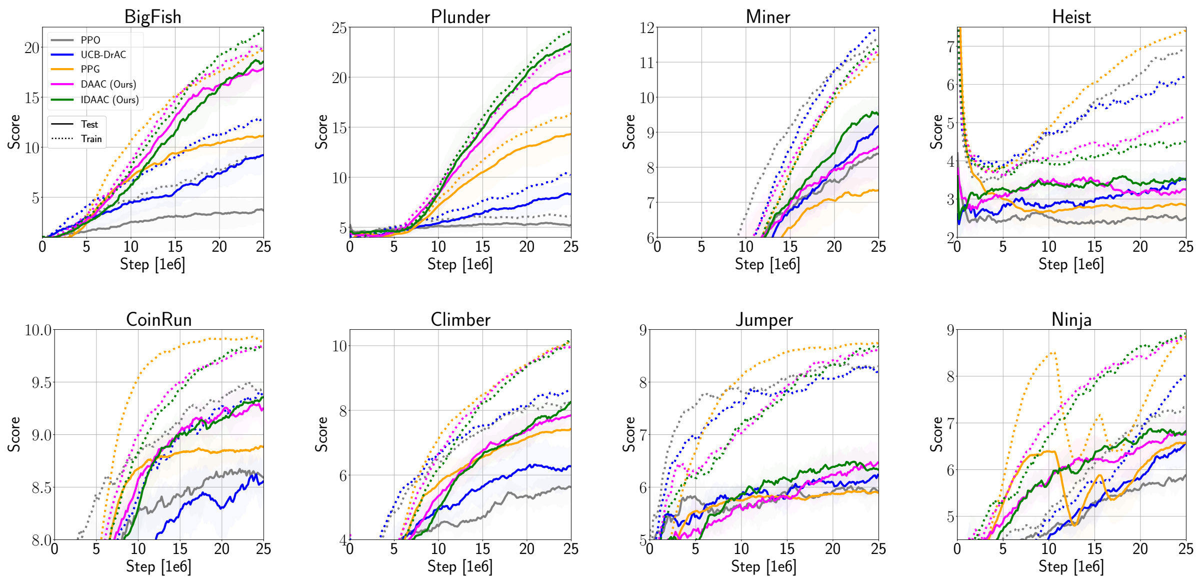


Figure 4.3: Train and Test Performance for IDAAC, DAAC, PPG, UCB-DrAC, and PPO, on eight diverse Procgen games. IDAAC and DAAC display state-of-the-art performance on the test levels, beating leading approaches (PPG and UCB-DrAC), as well a PPO baseline. Furthermore, IDAAC and DAAC exhibit a smaller generalization gap than other methods. The mean and standard deviation are computed over 10 runs with different seeds.

Table 4.1 shows the train and test performance of all methods, aggregated across all Procgen games. DAAC and IDAAC outperform all the baselines on both train and test. Figure 4.3 shows the train and test performance on eight of the Procgen games. We show comparisons with a vanilla RL algorithm PPO, the previous state-of-the-art UCB-DrAC and our strongest baseline

PPG. Both of our approaches, DAAC and IDAAC, show superior results on the test levels, relative to the other methods. In addition, IDAAC and DAAC achieve better or comparable performance on the training levels for most of these games. While DAAC already shows notable gains over the baselines, IDAAC further improves upon it, thus emphasizing the benefits of combining our two contributions. Figures 4.4 and 4.5 show the train and test performance, respectively, for IDAAC, DAAC, PPG, UCB-DrAC, and PPO on all Procgen games.

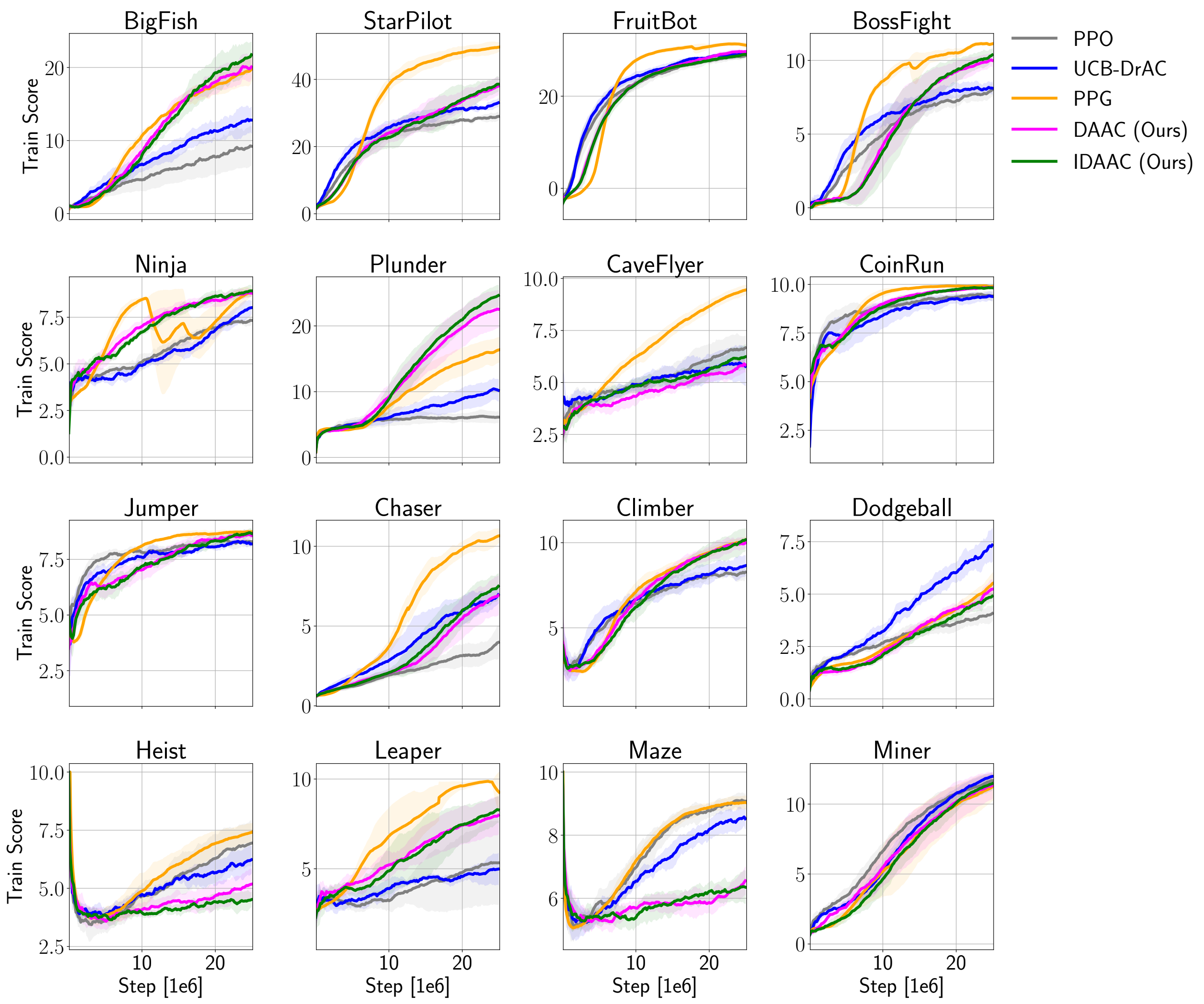


Figure 4.4: Train Performance of IDAAC, DAAC, PPG, UCB-DrAC, and PPO on all Procgen games. IDAAC outperforms the other methods on most games and is significantly better than PPO. The mean and standard deviation are computed over 10 runs with different seeds.

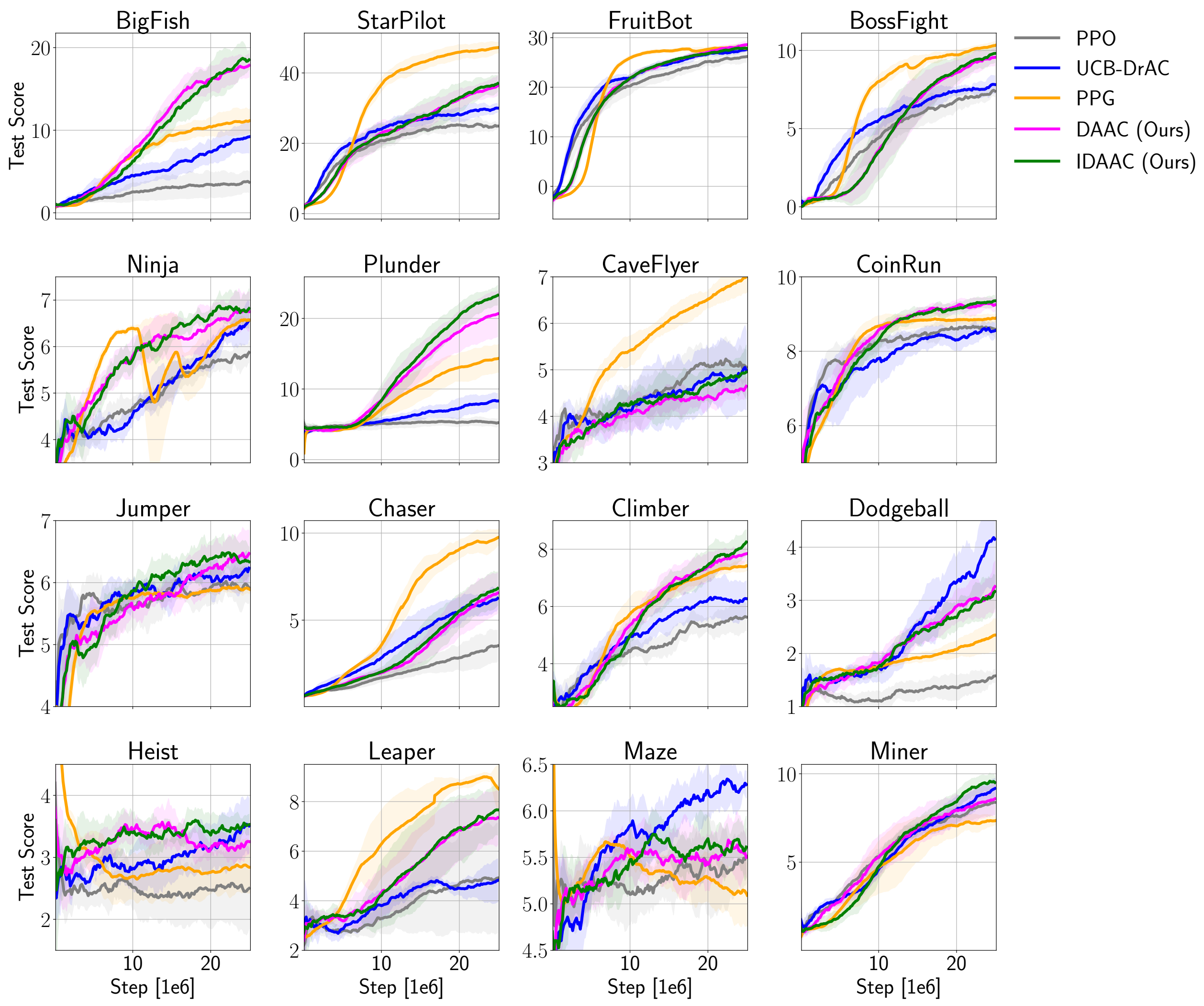


Figure 4.5: Test Performance of IDAAC, DAAC, PPG, UCB-DrAC, and PPO on all Procgen games. IDAAC outperforms the other methods on most games and is significantly better than PPO. The mean and standard deviation are computed over 10 runs with different seeds.

4.5.2 Ablations

We also performed a number of ablations to emphasize the importance of each component used by our method. First, **D**ecoupled **V**alue **A**ctor-**C**ritic or **DVAC** is an ablation to **DAAC** that learns to predict the value rather than the advantage for training the policy network. This ablation helps disentangle the effect of predicting the advantage function from the effect of using a separate value network and performing multiple updates for the value than for the policy. In principle, this decoupling could result in a more accurate estimate of the value function, which can in turn lead to more effective policy optimization. Second, **A**dvantage **A**ctor-**C**ritic or **AAC** is a modification to PPO that includes an extra head for predicting the advantage function (in a single network). The role of this ablation is to understand the importance of *not* backpropagating gradients from the value into the policy network, even while using gradients from the advantage.

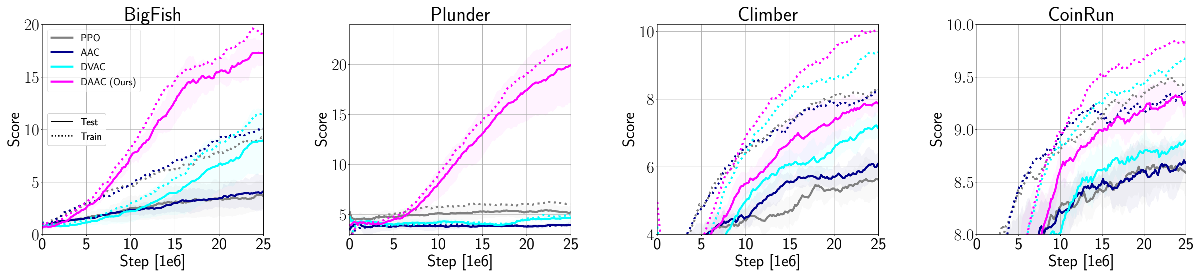


Figure 4.6: Train and Test Performance for PPO, DAAC, and two of its ablations, DVAC and AAC, on four Procgen games. DAAC outperforms all the ablations on both train and test, emphasizing the importance of each component. The mean and standard deviation are computed over 5 runs with different seeds.

Figure 4.6 shows the train and test performance of DAAC, DVAC, AAC, and PPO on four Procgen games. DAAC outperforms all the ablations on both train and test environments, emphasizing the importance of each component. In particular, the fact that AAC's generalization ability is worse than that of DAAC suggests that predicting the advantage function in addition to also predicting the value function (as part of training the policy network) does not solve the problem of overfitting. Hence, using the value loss to update the policy parameters still hurts generalization even when the advantage is also used to train the network. In addition, the fact that DVAC has worse test performance than DAAC indicates that DAAC's gains are not merely due to having access to a more accurate value function or to the reduced interference between optimizing the policy and value.

These results are consistent with our claim that using gradients from the value to update the policy can lead to representations that overfit to spurious correlations in the training environments. In contrast, using gradients from the advantage to train the policy network leads to agents that generalize better to new environments.

4.5.3 DeepMind Control with Distractors

In this section, we evaluate our methods on the DeepMind Control Suite from pixels (DMC, Tassa et al. [2018a]). We use three tasks, namely Cartpole Balance, Cartpole Swingup, and Ball In Cup. For each task, we study two settings with different types of backgrounds, namely *synthetic* distractors and *natural* videos from the Kinetics dataset [Kay et al. 2017], as introduced in Zhang et al. [2020b]. Note that in the synthetic and natural settings, the background is sampled from a list of videos at the beginning of each episode, which creates spurious correlations between backgrounds and rewards. As shown in Figure 4.7, DAAC and IDAAC significantly outperform PPO, UCB-DrAC, and PPG on all these environments.

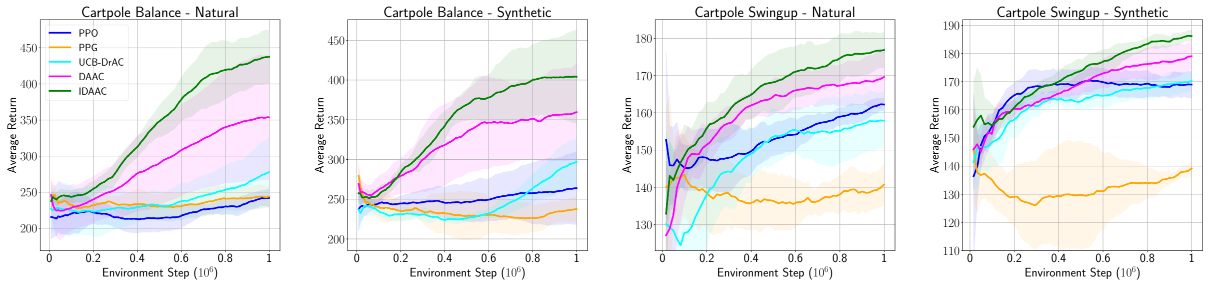


Figure 4.7: Average return on two DMC tasks, Cartpole Balance and Cartpole Swingup with natural and synthetic video backgrounds. Our DAAC and IDAAC approaches outperform PPO, PPG, and UCB-DrAC. The mean and standard deviation are computed over 10 runs with different seeds.

For our DMC experiments, we followed the protocol proposed in Zhang et al. [2020b] to modify the tasks so that they contain natural and synthetic distractors in the background. For each DMC task, we split the generated environments (each with a different background video) into training (80%) and testing (20%). The results shown here correspond to the average return on the test environments, over the course of training. Note that this setting is slightly different from the one used in Raileanu et al. [2020] which shows results on all the generated environments, just like Zhang et al. [2020b].

4.5.4 Value Loss and Generalization

When using actor-critic policy-gradient algorithms, a more accurate estimate of the value function leads to lower variance gradients and thus better policy optimization on a given environment [Sutton et al. 1999]. While an accurate value function improves sample efficiency and training performance, it can also lead to overfitting when the policy and value share the same representation. To validate this claim, we looked at the correlation between value loss and test performance. More specifically, we trained 6 PPO agents on varying numbers of Procgen levels, namely 200, 500, 1000, 2000, 5000, and 10000. As expected, models trained on a larger number of levels generalize better to unseen levels as illustrated in Figure 4.8. However, *agents trained on more levels have a higher value loss at the end of training than agents trained on fewer levels*, so the *value loss is positively correlated with generalization ability*. This observation is consistent with our claim that using a shared network for the policy and value function can lead to overfitting. Our hypothesis was that, when using a common network for the policy and value function, accurately predicting the values implies that the learned representation relies on spurious correlation, which would likely lead to poor generalization at test time. Similarly, an agent with good generalization suggests that its representation relies on the features needed to learn an optimal policy for the entire family of environments, which are insufficient for accurately predicting the value function (as explained in Section 4.1.1). See Figures 4.9, 4.10, and 4.11 for the relationship between value loss, test score, and the number of training levels for all Procgen games.

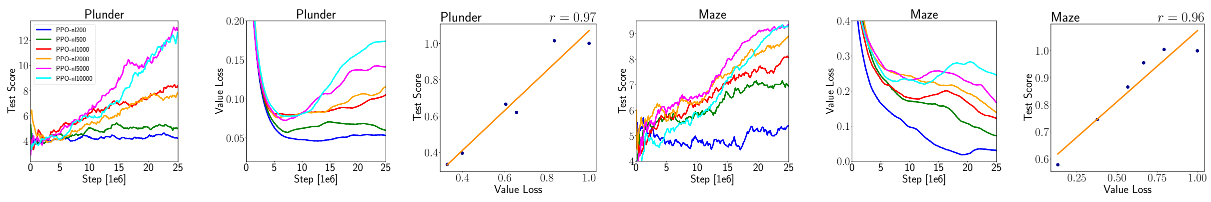


Figure 4.8: PPO agents trained on varying numbers of levels: 200, 500, 1000, 2000, 5000, and 10000 for Plunder (left) and Maze (right). For each game, from left to right we show the test score, value loss, and correlation between value loss and test score after 25M training steps. Both the test score and the value loss increase with the number of training levels used. These results indicate there is a *positive correlation between value loss and generalization* when using a shared network for the policy and value function.

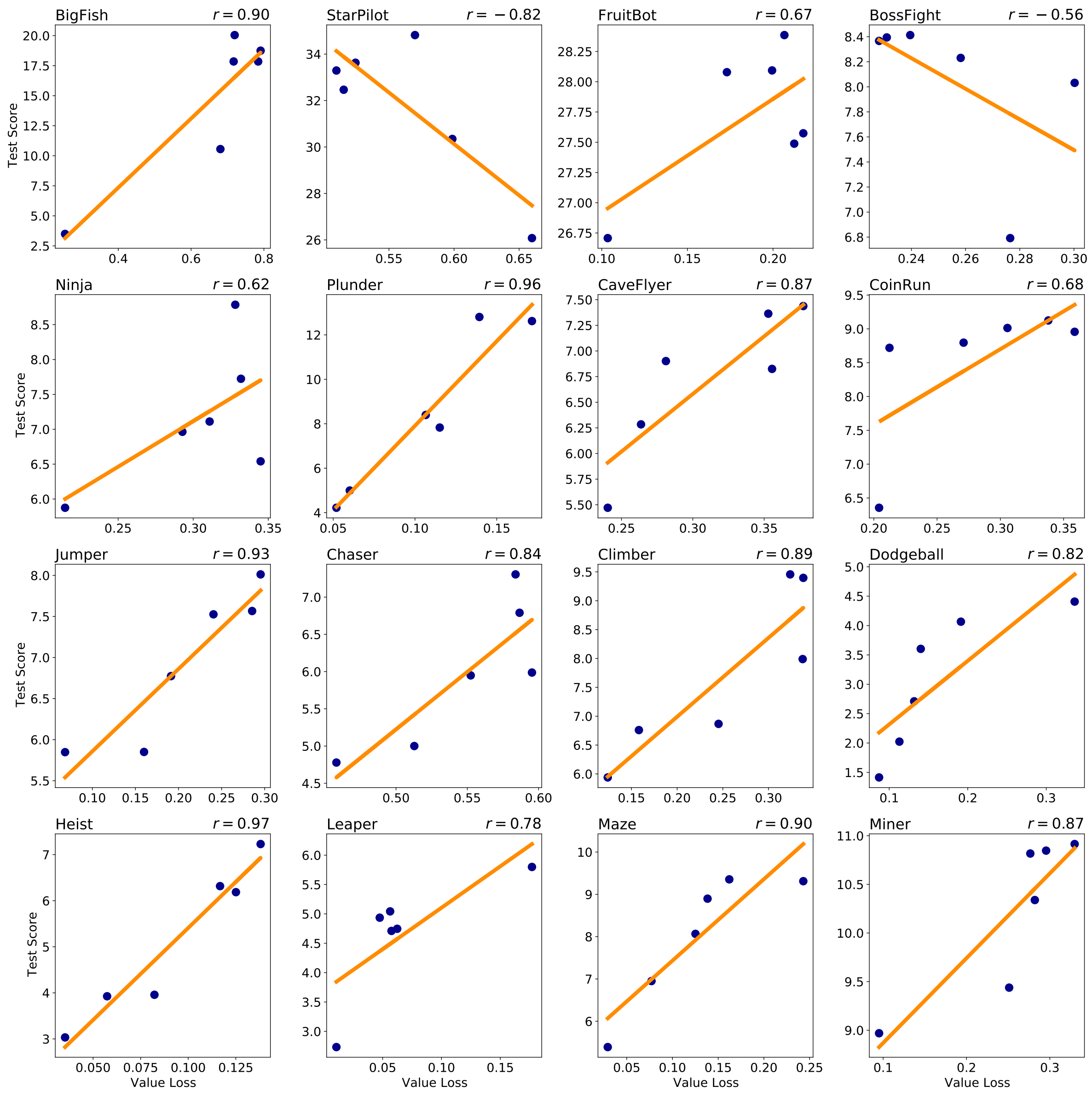


Figure 4.9: Correlation of the value loss and test score for PPO agents trained on varying numbers of levels: 200, 500, 1000, 2000, 5000, and 10000. Surprisingly, the value loss is positively correlated with the test score for most games, suggesting that *models with larger value loss generalize better.*

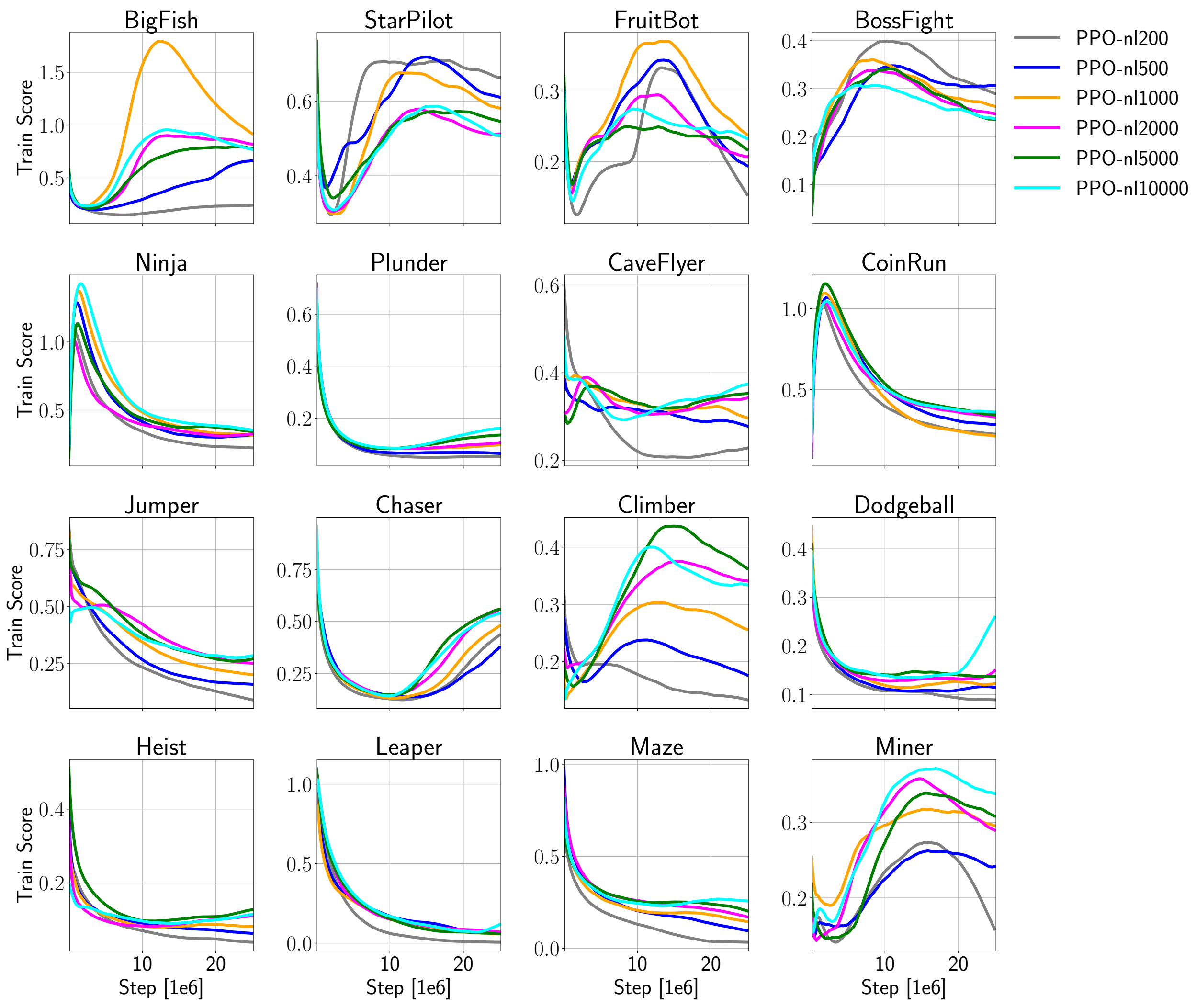


Figure 4.10: Test score for PPO agents trained on varying numbers of levels: 200, 500, 1000, 2000, 5000, and 10000. For most games, the test score increases with the number of training levels, suggesting that models trained on more levels generalize better to unseen levels, as expected.

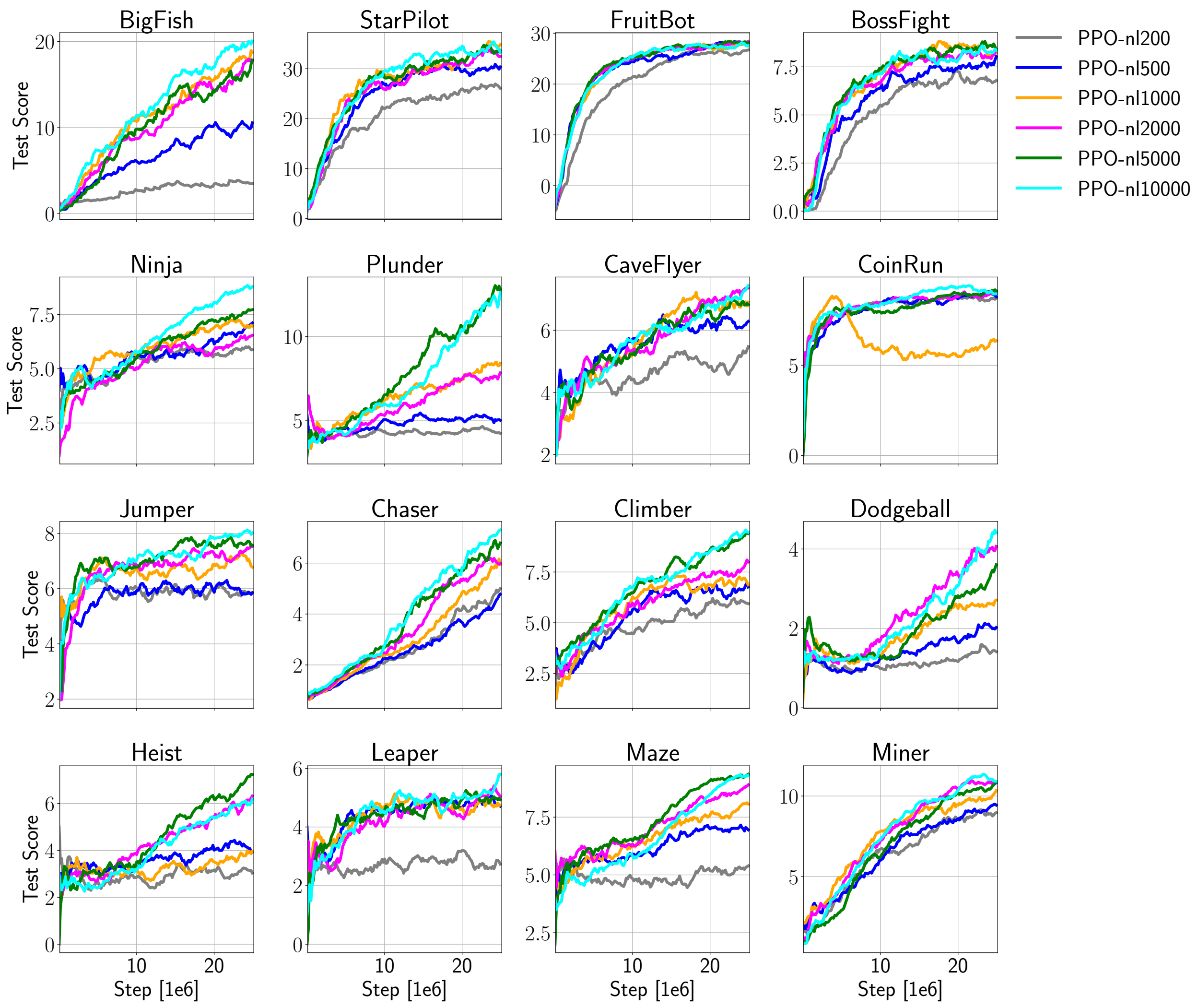
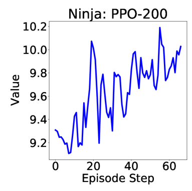
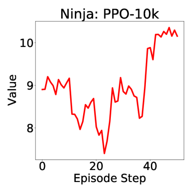
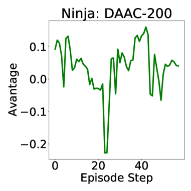
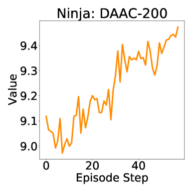


Figure 4.11: Value loss for PPO agents trained on varying numbers of levels: 200, 500, 1000, 2000, 5000, and 10000. For most games, the value loss increases with the number of training levels used, suggesting that models trained on more levels (and thus with better generalization) have higher value loss.

4.5.5 Advantage vs. Value During an Episode

In Section 4.1, we claim that in procedurally generated environments with partial observability, in order to accurately estimate the value function, the agent needs to memorize the number of remaining steps in the level. As Figure 4.1 shows, a standard RL agent predicts very different values for the initial states of two levels even if the observations are semantically identical. This suggests that the agent must have memorized the length of each level since the partial observation at the beginning of a level does not contain enough information to accurately predict the expected return.

Table 4.2: The trade-off between generalization and value accuracy illustrated on a single Ninja level. A PPO agent trained on 200 levels (blue) has high value accuracy but low generalization performance, and its value predictions have a near linear dependency on the episode step. This linear relationship further supports the claim that the agent memorizes level-specific features which are needed to predict the value function given only partial observations. In contrast, a PPO agent trained on 10k levels (red) with good generalization but low value accuracy does not display this linear trend. When sharing parameters for the policy and value function, there is a trade-off between fitting the value and learning general policies. By decoupling the policy and value, our model DAAC can achieve both high value accuracy and good generalization performance. To train the policy, DAAC uses gradients from predicting the advantage (green), which does not display the linear trend, thus is less prone to overfitting. DAAC's value estimate (orange) still shows a linear trend but, in contrast to PPO, this does not negatively affect the policy since DAAC uses separate networks to learn the policy and value.

PPO-200	PPO-10k	DAAC-200
Test Score: Value Loss: 	Test Score: Value Loss: 	Test Score: 7 Value Loss: ( 

We further investigate this issue by exploring the predicted value's dependency on the episode step. Instead of just comparing the initial states as in Figure 4.1, we plot the value predicted by the agent over the course of an entire trajectory in one of the Ninja levels (see Table 4.2). Since the agent is rewarded only if it reaches the goal at the end of the episode, the true value increases linearly with the episode step over the course of the agent's trajectory. As seen in Table 4.2, the estimated value of a PPO agent trained on 200 levels also has a quasi-linear dependence on the episode step, suggesting that the agent must know how many remaining steps are in the game at any point during this episode. However, the episode step cannot be inferred solely from partial observations since the training levels contain observations with the same semantics but different values, as illustrated in Figure 4.1. In order to accurately predict the values of such observations, the agent must learn representations that capture level-specific features (such as the backgrounds) which would allow it to differentiate between semantically similar observations with different values. Since PPO uses a common representation for the policy and value function, this can lead to policies that overfit to the particularities of the training environments. Note that a PPO agent trained on 10k levels does not show the same linear trend between the value and episode step. This implies that there is a trade-off between generalization and value accuracy for models that use a shared network to learn the policy and value.

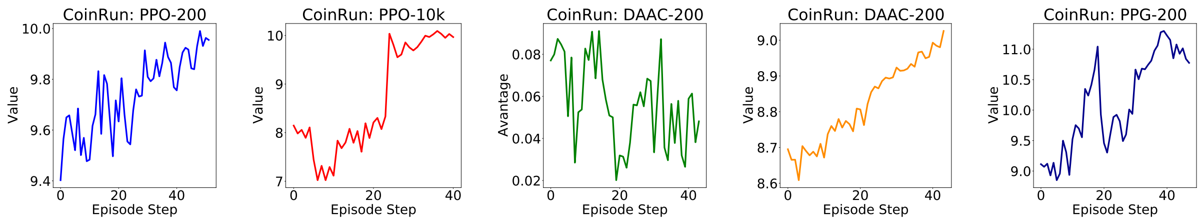


Figure 4.12: Examples illustrating the timestep-dependence of the value function for a single CoinRun level. The dark and light blue curves show the value as a function of episode step for PPG and PPO, respectively, each trained on 200 levels. Note the near linear relationship, indicating overfitting to the training levels. By contrast, a PPO model (red) trained on 10k levels (thus exhibiting far less overfitting) does not show this relationship. Our DAAC model trained on 200 levels (green) also lacks this adverse dependence in the advantage prediction which is used for training the policy network, thus is able to generalize better than the PPO model trained on the same amount of data (see Fig. 4.3). Nevertheless, DAAC's value estimate still have a linear trend (orange) but, in contrast to PPO and PPG, this does not negatively affect the policy since we use separate networks for learning the policy and value.

By decoupling the policy and value, our model DAAC can achieve both high value accuracy and good generalization performance. Note that the advantage estimated by DAAC (trained on 200 levels) shows no clear dependence on the environment step, thus being less prone to overfitting.

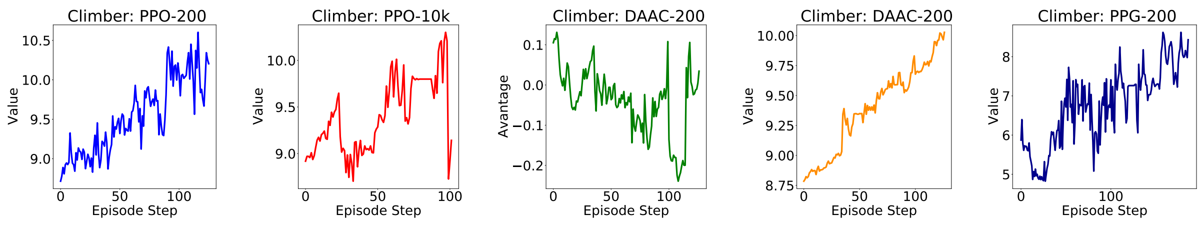


Figure 4.13: Examples illustrating the timestep-dependence of the value function for a single Climber level. The dark and light blue curves show the value as a function of episode step for PPG and PPO, respectively, each trained on 200 levels. Note the near linear relationship, indicating overfitting to the training levels. By contrast, a PPO model (red) trained on 10k levels (thus exhibiting far less overfitting) does not show this relationship. Our DAAC model trained on 200 levels (green) also lacks this adverse dependence in the advantage prediction which is used for training the policy network, thus is able to generalize better than the PPO model trained on the same amount of data (see Fig. 4.3). Nevertheless, DAAC's value estimate still have a linear trend (orange) but, in contrast to PPO and PPG, this does not negatively affect the policy since we use separate networks for learning the policy and value.

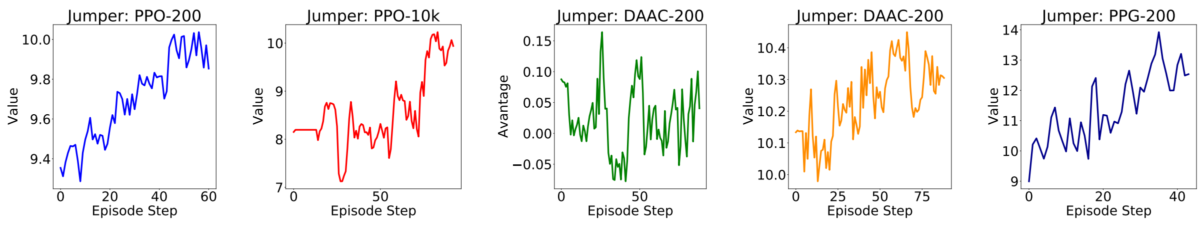


Figure 4.14: Examples illustrating the timestep-dependence of the value function for a single Jumper level. The dark and light blue curves show the value as a function of episode step for PPG and PPO, respectively, each trained on 200 levels. Note the near linear relationship, indicating overfitting to the training levels. By contrast, a PPO model (red) trained on 10k levels (thus exhibiting far less overfitting) does not show this relationship. Our DAAC model trained on 200 levels (green) also lacks this adverse dependence in the advantage prediction which is used for training the policy network, thus is able to generalize better than the PPO model trained on the same amount of data (see Fig. 4.3). Nevertheless, DAAC's value estimate still have a linear trend (orange) but, in contrast to PPO and PPG, this does not negatively affect the policy since we use separate networks for learning the policy and value.

Nevertheless, DAAC's value estimate still shows a linear trend but, in contrast to PPO, this does not negatively affect the policy since DAAC uses separate networks for the policy and value. This analysis indicates that using advantages rather than values to update the policy network leads to better generalization performance while also being able to accurately predict the value function. See Figures 4.12, 4.13, and 4.14 for similar results on CoinRun, Climber, and Jumper, as well as comparisons with PPG which displays a similar trend as PPO trained on 200 levels.

4.5.6 Value Variance

In this section, we look at the variance in the predicted values for the initial observation, across all training levels. In partially-observed procedurally generated environments, there should be no way of telling how difficult or long a level is (and thus how much reward is expected) from the initial observation alone since the end of the level cannot be seen. Thus, we would expect a model with strong generalization to predict similar values for the initial observation irrespective of the environment instance. If this is not the case and the model uses a common representation for the policy and value function, the policy is likely to overfit to the training environments. As Figure 4.15 shows, the variance decreases with the number of levels used for training. This is consistent with our observation that models with better generalization predict more similar values for observations that are semantically different such as the initial observation. In contrast, models trained on a low number of levels, memorize the value of the initial observation for each level, leading to poor generalization in new environments. Note that we chose to illustrate this phenomenon on three of the Procgen games (*i.e.* Climber, Jumper, and Ninja) where it is more apparent due to their partial-observability and substantial level diversity (in terms of length).

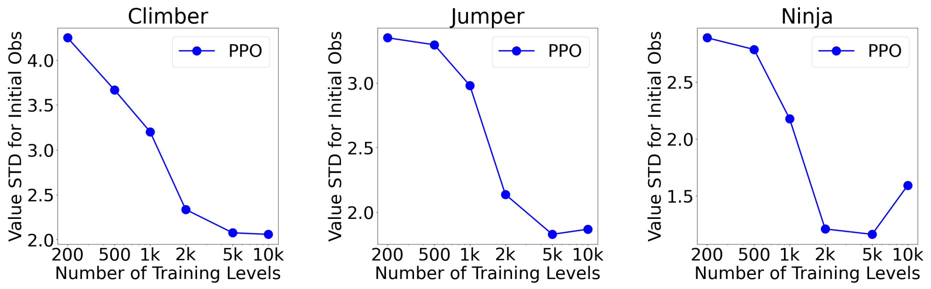


Figure 4.15: Standard deviation of the predicted values for the initial observations from 200 levels, as a function of the number of training levels. From left to right: Climber, Jumper, and Ninja. The 200 levels used to compute the standard deviation were part of the training set for all the agents. Note that, in general, the variance of the value decreases with the number of training levels. This is consistent with our claim that when sharing parameters for the policy and value, models which generalize better predict close values for observations that are semantically similar (*e.g.* the initial observation) since they learn representations which are less prone to overfitting to level-specific features.

4.5.7 Robustness to Spurious Correlations

In this section, we investigate how robust the learned features, policies, and predicted values or advantages, are to spurious correlations. To answer this question, we measure how much the features, policies, and predictions vary when the background of an observation changes. Note that the change in background does not change the underlying state of the environment but only its visual aspect. Hence, a change in background should not modify the agent's policy or learned representation. For this experiment, we collect a buffer of 1000 observations from 20 training levels, using a PPO agent trained on 200 levels. Then, we create 10 extra versions of each observation by changing the background. We then measure the L1-norm and L2-norm between the learned representation (*i.e.* final vector before the policy's softmax layer) of the original observation and each of its other versions. We also compute the difference in predicted outputs (*i.e.* values for PPO and PPG or advantages for DAAC and IDAAC) and the Jensen-Shannon Divergence (JSD) between the policies. For all these metrics, we first take the mean for all 10 backgrounds to obtain a single point for each observation, and then we report the mean and standard deviation across all 1000 observations, resulting in an average statistic of how much these metrics change as a result of varying the background.

Figure 4.16 shows the results for Ninja, comparing IDAAC, DAAC, PPG, as well as PPO trained on 200 and 10k levels. In particular, the results show that both our methods learn representations which are more robust to changes in the background than PPO and PPG (assuming all methods are trained on the same number of levels *i.e.* 200). Overall, the differences due to background changes in the auxiliary outputs of the policy networks for DAAC and IDAAC (*i.e.* the predicted advantages) are smaller than those of PPO and PPG (*i.e.* the predicted values). These results indicate that DAAC and IDAAC are more robust than PPO and PPG to visual features which are irrelevant for control.

We do not observe a significant difference across the JSDs of the different methods. However, in the case of Procgen, the JSD isn't a perfect measure of the semantic difference between two policies because some of the actions have the same effect on the environment (*e.g.* in Ninja, there are two actions that move the agent to the right) and thus are interchangeable. Two policies could have a large JSD while being semantically similar, thus rendering the policy robustness analysis inconclusive.

In some cases, PPO-10k exhibits better robustness to different backgrounds than DAAC and IDAAC by exhibiting a lower feature norm and value or advantage difference. However, PPO-10k is a PPO model trained on 10000 levels of a game, while DAAC and IDAAC are trained only on 200 levels. For most Procgen games, training on 10k levels is enough to generalize to the test distribution so PPO-10k is expected to generalize better than methods trained on 200 levels. In this work, we are interested in generalizing to unseen levels from a small number of training levels, so PPO-10k is used as an upper-bound rather than a baseline since a direct comparison wouldn't be fair. Hence, it is not surprising that some of these robustness metrics are better for PPO-10k than DAAC and IDAAC.

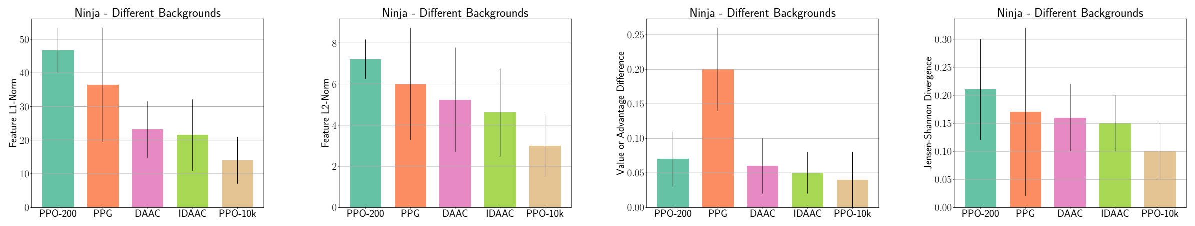


Figure 4.16: Variations in the learned features, policies, and values or advantages when changing the background in Ninja. From left to right we report the L1 and L2-norm for the features, the value or advantage difference, and the Jensen-Shannon Divergence for the policy. We compare PPO trained on 200 and 10k levels with PPG, DAAC, and IDAAC. Our models are more robust to changes in the background (which does not affect the state). The means and standard deviations were computed over 10 different backgrounds.

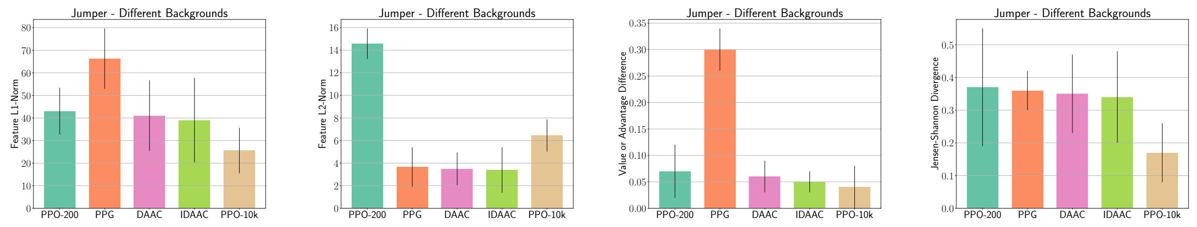


Figure 4.17: Variations in the learned features, policies, and values or advantages when changing the background in Jumper. From left to right we report the L1 and L2-norm for the features, the value or advantage difference, and the Jensen-Shannon Divergence for the policy. We compare PPO trained on 200 and 10k levels with PPG, DAAC, and IDAAC. Our models are more robust to changes in the background (which does not affect the state). The means and standard deviations were computed over 10 different backgrounds.

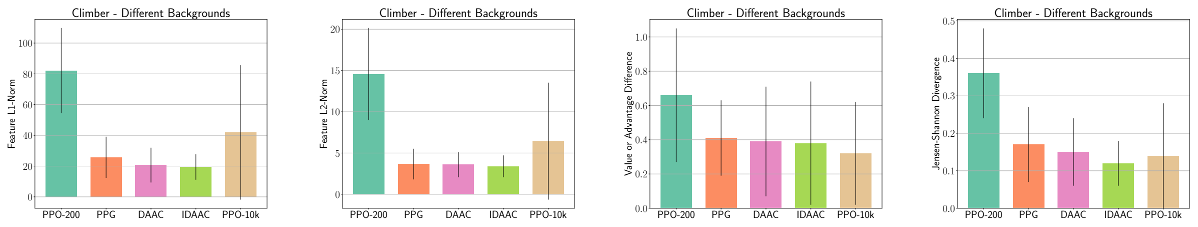


Figure 4.18: Variations in the learned features, policies, and values or advantages when changing the background in Climber. From left to right we report the L1 and L2-norm for the features, the value or advantage difference, and the Jensen-Shannon Divergence for the policy. We compare PPO trained on 200 and 10k levels with PPG, DAAC, and IDAAC. Our models are more robust to changes in the background (which does not affect the state). The means and standard deviations were computed over 10 different backgrounds.

4.6 Conclusion

In this chapter, we identified a new problem with standard deep reinforcement learning algorithms which causes overfitting, namely the asymmetry between the policy and value representation. To alleviate this problem, we proposed IDAAC, which decouples the optimization of the policy and value function while still learning effective behaviors. IDAAC also introduces an auxiliary loss which constrains the policy representation to be invariant with respect to the environment instance. IDAAC achieves a new state-of-the-art on the Procgen benchmark and outperforms strong RL algorithms on DeepMind Control tasks with distractors. In contrast to other popular methods, our approach can both achieve good generalization while also learning accurate value estimates. Moreover, IDAAC learns representations and predictions which are more robust to cosmetic changes in the observations that do not change the underlying state of the environment. Since this work was published, other papers have cited and used our approach as a baseline [Miao et al. 2021; Mazoure et al. 2021].

One limitation of our work is the focus on learning representations which are invariant to the number of remaining steps in the episode. While this inductive bias will not be helpful for all problems, the settings where we can expect most gains are Markovian environments with partial observability, a set of goal states, and episode length variations (*e.g.* navigation of different layouts). A promising avenue for future work is to investigate other auxiliary losses in order to efficiently learn more general behaviors. One desirable property of such auxiliary losses is to capture the minimal set of features needed to act in the environment. While our experiments show that predicting the advantage function improves generalization, we currently lack a firm theoretical argument for this. The advantage could act as a regularizer, being less prone to memorizing the remaining episode length, or it could be better correlated with the underlying state of the environment rather than its visual appearance. Investigating these hypotheses could further improve our understanding of what leads to better representations and what we are still missing. Finally, the solution we propose here is only a first step towards solving the policy-value representation asymmetry and we hope many other ideas will be explored in future work.

5 | Impact-Driven Exploration in Procedurally Generated Environments

In the previous two chapters, we studied the problem of zero-shot generalization to new instances of a task, in the context of image-based deep reinforcement learning. In this chapter, we will investigate the problem of learning general exploration strategies which work across a wide range of task instances. Exploration in sparse reward environments remains one of the key challenges of model-free reinforcement learning. Instead of solely relying on extrinsic rewards provided by the environment, many state-of-the-art methods use intrinsic rewards to encourage exploration. In this chapter, we show that existing methods fall short in procedurally generated environments where an agent needs to solve many different instances of the same task, so it is unlikely to visit a state more than once. We propose a new type of intrinsic reward which encourages the agent to take actions that lead to significant changes in its learned state representation. We evaluate our method on multiple challenging procedurally generated tasks in MiniGrid, as well as on tasks with high-dimensional observations used in prior work. Our experiments demonstrate that this approach is more sample efficient than existing exploration methods, particularly for procedurally generated MiniGrid environments. Furthermore, we analyze the learned behavior as well as the intrinsic reward received by our agent. In contrast to previous methods, our intrinsic reward does not diminish during the course of training and it rewards the agent substantially more for interacting with objects that it can control.

5.1 Introduction

RL agents learn to act in new environments through trial and error, in an attempt to maximize their cumulative reward. However, many environments of interest, particularly those closer to real-world problems, do not provide a steady stream of rewards for agents to learn from. In such settings, agents require many episodes to come across any reward, often rendering standard RL methods inapplicable.

Inspired by human learning, the use of intrinsic motivation has been proposed to encourage agents to learn about their environments even when extrinsic feedback is rarely provided [Schmidhuber 1991b, 2010; Oudeyer et al. 2007; Oudeyer and Kaplan 2009]. This type of exploration bonus emboldens the agent to visit new states [Bellemare et al. 2016; Burda et al. 2019b; Ecoffet et al. 2019] or to improve its knowledge and forward prediction of the world dynamics [Pathak et al. 2017; Burda et al. 2019a], and can be highly effective for learning in hard exploration games such as Montezuma's Revenge [Mnih et al. 2016b]. However, most hard exploration environments used in previous work have either a limited state space or an easy way to measure similarity between states [Ecoffet et al. 2019] and generally use the same "*singleton*" environment for training and evaluation [Mnih et al. 2016b; Burda et al. 2019a]. Deep RL agents trained in this way are prone to overfitting to a specific environment and often struggle to generalize to even slightly different settings [Rajeswaran et al. 2017b; Zhang et al. 2018a,d]. As a first step towards addressing this problem, a number of procedurally generated environments have been recently released, for example DeepMind Lab [Beattie et al. 2016], Sokoban [Racanière et al. 2017], Malmö [Johnson et al. 2016], CraftAssist [Jernite et al. 2019], Sonic [Nichol et al. 2018a], CoinRun [Cobbe et al. 2019c], Obstacle Tower [Juliani et al. 2019a], or Capture the Flag [Jaderberg et al. 2019].

In this chapter, we investigate exploration in procedurally generated sparse reward environments. We demonstrate that many popular exploration methods, which are state-of-the-art in challenging singleton environments, fall short in procedurally generated ones as they (i) make strong assumptions about the environment (deterministic or resettable to previous states) [Ecoffet et al. 2019; Aytar et al. 2018], (ii) make strong assumptions about the state space (small number of different states or easy to determine if two states are similar) [Ecoffet et al. 2019; Burda et al. 2019b; Bellemare et al. 2016; Ostrovski et al. 2017; Machado et al. 2018a], or (iii) provide intrinsic rewards that can diminish quickly during training [Pathak et al. 2017; Burda et al. 2019a].

To overcome these limitations, we propose **Rewarding Impact-Driven Exploration** (RIDE), a novel intrinsic reward for exploration in RL that encourages the agent to take actions which result in impactful changes to its representation of the environment state (see Figure 5.1 for an illustration). We compare against state-of-the-art intrinsic reward methods on singleton environments with high-dimensional observations (*i.e.* visual inputs), as well as on hard-exploration tasks in procedurally generated gridworld environments. Our experiments show that RIDE outperforms state-of-the-art exploration methods, particularly in procedurally generated environments. Furthermore, we present a qualitative analysis demonstrating that RIDE, in contrast to prior work, does not suffer from diminishing intrinsic rewards during training and encourages agents substantially more to interact with objects that they can control (relative to other state-action pairs).

5.2 Related Work

The problem of exploration in reinforcement learning has been extensively studied. Exploration methods encourage RL agents to visit novel states in various ways, for example by rewarding surprise [Schmidhuber 1991b,a, 2010, 2006; Achiam and Sastry 2017], information gain [Little and Sommer 2013; Still and Precup 2012; Houthooft et al. 2016], curiosity [Pathak et al. 2017; Burda

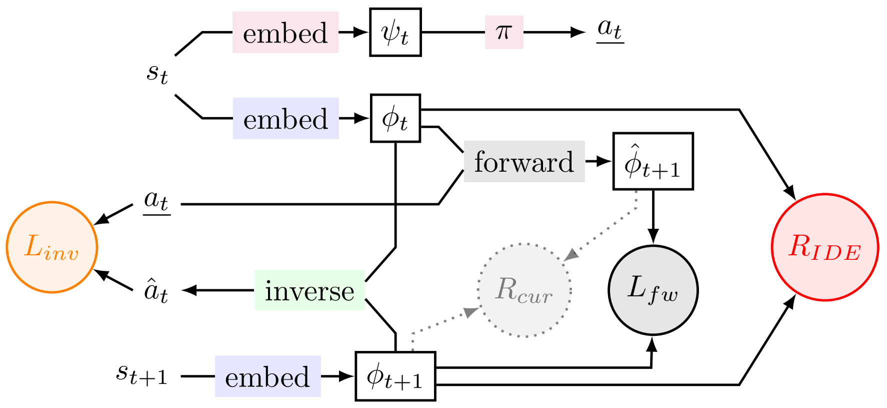


Figure 5.1: RIDE rewards the agent for actions that have an impact on the state representation (R_{IDE}), which is learned using both a forward (L_{fw}) and an inverse dynamics (L_{inv}) model.

et al. 2019b], empowerment [Klyubin et al. 2005; Rezende and Mohamed 2015; Gregor et al. 2017], diversity [Eysenbach et al. 2019], feature control [Jaderberg et al. 2017; Dilokthanakul et al. 2019], or decision states [Goyal et al. 2019; Modhe et al. 2019]. Another class of exploration methods apply the Thompson sampling heurisitc [Osband et al. 2016; Ostrovski et al. 2017; O'Donoghue et al. 2018; Tang et al. 2017]. Osband et al. [2016] use a family of randomized Q-functions trained on bootstrapped data to select actions, while Fortunato et al. [2018] add noise in parameter space to encourage exploration. Here, we focus on intrinsic motivation methods, which are widely-used and have proven effective for various hard-exploration tasks [Mnih et al. 2016b; Pathak et al. 2017; Bellemare et al. 2016; Burda et al. 2019b].

Intrinsic motivation can be useful in guiding the exploration of RL agents, particularly in environments where the extrinsic feedback is sparse or missing altogether [Oudeyer et al. 2007, 2008; Oudeyer and Kaplan 2009; Schmidhuber 1991b, 2010]. The most popular and effective kinds of intrinsic motivation can be split into two broad classes: count-based methods that encourage the agent to visit novel states and curiosity-based methods that encourage the agent to learn about the environment dynamics.

Count-Based Exploration. Strehl and Littman [2008] proposed the use of state visitation counts as an exploration bonus in tabular settings. More recently, such methods were extended to high-dimensional state spaces [Bellemare et al. 2016; Ostrovski et al. 2017; Martin et al. 2017; Tang et al. 2017; Machado et al. 2018a]. Bellemare et al. [2016] use a Context-Tree Switching (CTS) density model to derive a state pseudo-count, while Ostrovski et al. [2017] use PixelCNN as a state density estimator. Burda et al. [2019b] employ the prediction error of a random network as exploration bonus with the aim of rewarding novel states more than previously seen ones. However, one can expect count-based exploration methods to be less effective in procedurally generated environments with sparse reward. In these settings, the agent is likely to characterize two states as being different even when they only differ by features that are irrelevant for the task (*e.g.* the texture of the walls). If the agent considers most states to be "novel", the feedback signal will not be distinctive or varied enough to guide the agent.

Curiosity-Driven Exploration. Curiosity-based bonuses encourage the agent to explore the environment to learn about its dynamics. Curiosity can be formulated as the error or uncertainty in predicting the consequences of the agent's actions [Stadie et al. 2015; Pathak et al. 2017; Burda et al. 2019b]. For example, Pathak et al. [2017] learn a latent representation of the state and design an intrinsic reward based on the error of predicting the next state in the learned latent space. While we use a similar mechanism for learning state embeddings, our exploration bonus is very different and builds upon the difference between the latent representations of two consecutive states. As we will see in the following sections, one problem with their approach is that the intrinsic reward can vanish during training, leaving the agent with no incentive to further explore the environment and reducing its feedback to extrinsic reward only.

Generalization in Deep RL. Most of the existing exploration methods that have achieved impressive results on difficult tasks [Ecoffet et al. 2019; Pathak et al. 2017; Burda et al. 2019b; Bellemare et al. 2016; Choi et al. 2019; Aytar et al. 2018], have been trained and tested on the same environment and thus do not generalize to new instances. Several recent papers [Rajeswaran et al. 2017b; Zhang et al. 2018a,d; Machado et al. 2018b; Foley et al. 2018] demonstrate that deep RL is susceptible to severe overfitting. As a result, a number of benchmarks have been recently released for testing generalization in RL [Beattie et al. 2016; Cobbe et al. 2019c; Packer et al. 2018a; Justesen et al. 2018b; Leike et al. 2017; Nichol et al. 2018a; Juliani et al. 2019a]. Here, we make another step towards developing exploration methods that can generalize to unseen scenarios by evaluating them on procedurally generated environments. We opted for MiniGrid [Chevalier-Boisvert et al. 2018] because it is fast to run, provides a standard set of tasks with varied difficulty levels, focuses on single-agent, and does not use visual inputs, thereby allowing us to better isolate the exploration problem.

More closely related to our work are the papers of Marino et al. [2019] and Zhang et al. [2019]. Marino et al. [2019] use a reward that encourages changing the values of the non-proprioceptive features for training low-level policies on locomotion tasks. Their work assumes that the agent has access to a decomposition of the observation state into internal and external parts, an assumption which may not hold in many cases and may not be trivial to obtain even if it exists. Zhang et al. [2019] use the difference between the successor features of consecutive states as intrinsic reward. In this framework, a state is characterized through the features of all its successor states. While both of these papers use fixed (*i.e.* not learned) state representations to define the intrinsic reward, we use forward and inverse dynamics models to learn a state representation constrained to only capture elements in the environment that can be influenced by the agent. Lesort et al. [2018] emphasize the benefits of using a learned state representation for control as opposed to a fixed one (which may not contain information relevant for acting in the environment). In the case of Zhang et al. [2019], constructing a temporally extended state representation for aiding exploration is not trivial. Such a feature space may add extra noise to the intrinsic reward due to the uncertainty of future states. This is particularly problematic when the environment is highly stochastic or the agent often encounters novel states (as it is the case in procedurally generated environments).

5.3 Background

We use the standard formalism of a single agent in a POMDP as defined in Section 3.3 and used in the previous two chapters. Along with the extrinsic reward r_t^e, the agent also receives some intrinsic reward r_t^i, which can be computed for any (s_t, a_t, s_{t+1}) tuple. Consequently, the agent tries to maximize the weighted sum of the intrinsic and extrinsic reward: $r_t = r_t^e + \omega_{ir} r_t^i$ where ω_{ir} is a hyperparameter to weight the importance of both rewards.

We built upon the work of Pathak et al. [2017] who note that some parts of the observation may have no influence on the agent's state. Thus, Pathak et al. propose learning a state representation that disregards those parts of the observation and instead only models (i) the elements that the agent can control, as well as (ii) those that can affect the agent, even if the agent cannot have an effect on them. Concretely, Pathak et al. learn a state representations $\phi(s) = f_{emb}(s;\ \theta_{emb})$ of a state s using an inverse and a forward dynamics model (see Figure 5.1). The forward dynamics model is a neural network parametrized by θ_{fw} that takes as inputs $\phi(s_t)$ and a_t, predicts the next state representation: $\hat{\phi}(s_{t+1}) = f_{fw}(\phi_t,\ a_t;\ \theta_{fw})$, and it is trained to minimize $L_{fw}(\theta_{fw}, \theta_{emb}) = \|\hat{\phi}(s_{t+1}) - \phi(s_{t+1})\|_2^2$. The inverse dynamics model is also a neural network parameterized by θ_{inv} that takes as inputs $\phi(s_t)$ and $\phi(s_{t+1})$, predicts the agent's action: $\hat{a}_t = f_{inv}(\phi_t, \phi_{t+1};\ \theta_{inv})$, and it is trained to minimize $L_{inv}(\theta_{inv}, \theta_{emb}) = CrossEntropy(\hat{a}_t, a_t)$ when the action space is discrete. Pathak et al.'s curiosity-based intrinsic reward is proportional to the squared Euclidean distance between the actual embedding of the next state $\phi(s_{t+1})$ and the one predicted by the forward model $\hat{\phi}(s_{t+1})$.

5.4 Approach

Our main contribution is a novel intrinsic reward based on the change in the state representation produced by the agent's action. The proposed method encourages the agent to try out actions that have a significant impact on the environment. We demonstrate that this approach can promote effective exploration strategies when the feedback from the environment is sparse.

We train a forward and an inverse dynamics model to learn a latent state representation $\phi(s)$ as proposed by Pathak et al. [2017]. However, instead of using the Euclidean distance between the predicted next state representation and the actual next state representation as intrinsic reward (R_{cur} in Figure 5.1), we define impact-driven reward as the Euclidean distance between consecutive state representations (R_{IDE} in Figure 5.1). Compared to curiosity-driven exploration, impact-driven exploration rewards the agent for very different state-actions, leading to distinct agent behaviors which we analyze in Section 5.5.5.

Stanton and Clune [2018] categorize exploration into: *across-training* and *intra-life* and argue they are complementary. Popular methods such as count-based exploration [Bellemare et al. 2016] encourage agents to visit novel states in relation to all prior training episodes (*i.e.* across-training novelty), but they do not consider whether an agent visits novel states within some episode (*i.e.* intra-life novelty). As we will see, RIDE combines both types of exploration.

Formally, RIDE is computed as the L_2-norm $\|\phi(s_{t+1}) - \phi(s_t)\|_2$ of the difference in the learned state representation between consecutive states. However, to ensure that the agent does not go back and forth between a sequence of states (with a large difference in their embeddings) in order to gain intrinsic reward, we discount RIDE by episodic state visitation counts. Concretely, we divide the impact-driven reward by $\sqrt{N_{ep}(s_{t+1})}$, where $N_{ep}(s_{t+1})$ is the number of times that state has been visited during the current episode, which is initialized to 1 in the beginning of the episode. In high-dimensional regimes, one can use episodic pseudo-counts instead [Bellemare et al. 2016; Ostrovski et al. 2017]. Thus, the overall intrinsic reward provided by RIDE is calculated as:

$$R_{IDE}(s_t, a_t) \equiv r_t^i(s_t, a_t) = \frac{\|\phi(s_{t+1}) - \phi(s_t)\|_2}{\sqrt{N_{ep}(s_{t+1})}}$$

where $\phi(s_{t+1})$ and $\phi(s_t)$ are the learned representations of consecutive states, resulting from the agent transitioning to state s_{t+1} after taking action a_t in state s_t. The state is projected into a latent space using a neural network with parameters θ_{emb}.

The overall optimization problem that is solved for training the agent is

$$\min_{\theta_\pi, \theta_{inv}, \theta_{fw}, \theta_{emb}} \left[\omega_\pi L_{RL}(\theta_\pi) + \omega_{fw} L_{fw}(\theta_{fw}, \theta_{emb}) + \omega_{inv} L_{inv}(\theta_{inv}, \theta_{emb})\right]$$

where θ_π are the parameters of the policy and value network ($a_t \sim \pi(s_t; \theta_\pi)$), and ω_π, ω_{inv} and ω_{fw} are scalars that weigh the relative importance of the reinforcement learning (RL) loss to that of the inverse and forward dynamics losses which are used for learning the intrinsic reward signal. Note that we never update the parameters of the inverse (θ_{inv}), forward (θ_{fw}), or embedding networks (θ_{emb}) using the signal from the intrinsic or extrinsic reward (*i.e.* the RL loss); we only use these learned state embeddings for constructing the exploration bonus and never as part of the agent's policy (Figure 5.1 highlights that the policy learns its own internal representation of the state ψ_t, which is only used for control and never for computing the intrinsic reward). Otherwise, the agent can artificially maximize its intrinsic reward by constructing state representations with large distances among themselves, without grounding them in environment observations.

Note that there is no incentive for the learned state representations to encode features of the environment that cannot be influenced by the agent's actions. Thus, our agent will not receive rewards for reaching states that are inherently unpredictable, making exploration robust with respect to distractor objects or other inconsequential sources of variation in the environment. As we will later show, RIDE is robust to the well-known noisy-TV problem in which an agent, that is rewarded for errors in the prediction of its forward model (such as the one proposed in Pathak et al. [2017]), gets attracted to local sources of entropy in the environment. Furthermore, the difference of consecutive state representations is unlikely to go to zero during learning as they are representations of actual states visited by the agent and constrained by the forward and inverse model. This is in contrast to Pathak et al. [2017] and Burda et al. [2019b] where the intrinsic reward goes to zero as soon as the forward model becomes sufficiently accurate or the agent's policy only explores well known parts of the state space.

5.5 Experiments

We evaluate RIDE on procedurally generated environments from MiniGrid, as well as on two existing singleton environments with high-dimensional observations used in prior work, and compare it against both standard RL and three commonly used intrinsic reward methods for exploration. For all our experiments, we show the mean and standard deviation of the average return across 5 different seeds for each model. The average return is computed as the rolling mean over the past 100 episodes.

At the time of writing this chapter, MiniGrid was one of the only platforms with procedurally generated environments which were fast enough to enable research progress in a timely fashion. The Procgen benchmark was released after the publishing of this paper. In addition, MiniGrid is better suited for understanding the behavior of various algorithms in sparse environments since it allows one to systematically increase the exploration difficulty. In contrast, changing the reward sparsity cannot be easily done in Procgen.

5.5.1 Environments

The first set of environments are procedurally generated grid-worlds in MiniGrid [Chevalier-Boisvert et al. 2018]. We consider three types of hard exploration tasks: *MultiRoomNXSY*, *KeyCorridorS3R3*, and *ObstructedMaze2Dlh*.

In MiniGrid, the world is a partially observable grid of size $N \times N$. Each tile in the grid contains at most one of the following objects: wall, door, key, ball, box and goal. The agent can take one of seven actions: turn left or right, move forward, pick up or drop an object, toggle or done.

For the sole purpose of comparing in a fair way to the curiosity-driven exploration work by

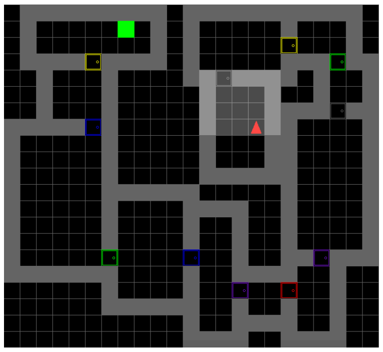


Figure 5.2: Rendering of a procedurally generated environment from MiniGrid's MultiRoomN12S10 task.

Pathak et al. [2017], we ran a one-off experiment on their Mario (singleton) environment [Kauten 2018]. We train our model with and without extrinsic reward on the first level of the game. The last (singleton) environment we evaluate on is *VizDoom* [Kempka et al. 2016].

5.5.2 Baselines

For all our experiments, we use IMPALA [Espeholt et al. 2018b] following the implementation of Küttler et al. [2019] as the base RL algorithm, and RMSProp [Tieleman and Hinton 2012] for optimization. All models use the same basic RL algorithm and network architecture for the policy and value functions, differing only in how intrinsic rewards are defined. In our experiments we compare with the following baselines: **Count**: Count-Based Exploration by Bellemare et al. [2016] which uses state visitation counts to give higher rewards for new or rarely seen states. **RND**: Random Network Distillation Exploration by Burda et al. [2019b] which uses the prediction error of a random network as exploration bonus with the aim of rewarding novel states more than previously encountered ones. **ICM**: Intrinsic Curiosity Module by Pathak et al. [2017] (see Section 5.3). **IMPALA**: Standard RL approach by Espeholt et al. [2018b] that uses only extrinsic reward and encourages random exploration by entropy regularization of the policy.

We present the results of RIDE in comparison to popular exploration methods, as well as an analysis of the learned policies and properties of the intrinsic reward generated by different methods.

5.5.3 MiniGrid Results

Figure 5.3 summarizes our results on various hard MiniGrid tasks. Note that the standard RL approach IMPALA (purple) is not able to learn in any of the environments since the extrinsic reward is too sparse. Furthermore, our results reveal that RIDE is more sample efficient compared to all the other exploration methods across all MiniGrid tasks considered here. While other exploration bonuses seem effective on easier tasks and are able to learn optimal policies where IMPALA fails, the gap between our approach and the others is increasing with the difficulty of the task. Furthermore, RIDE manages to solve some very challenging tasks on which the other methods fail to get any reward even after training on over 100M frames (Figure 5.3).

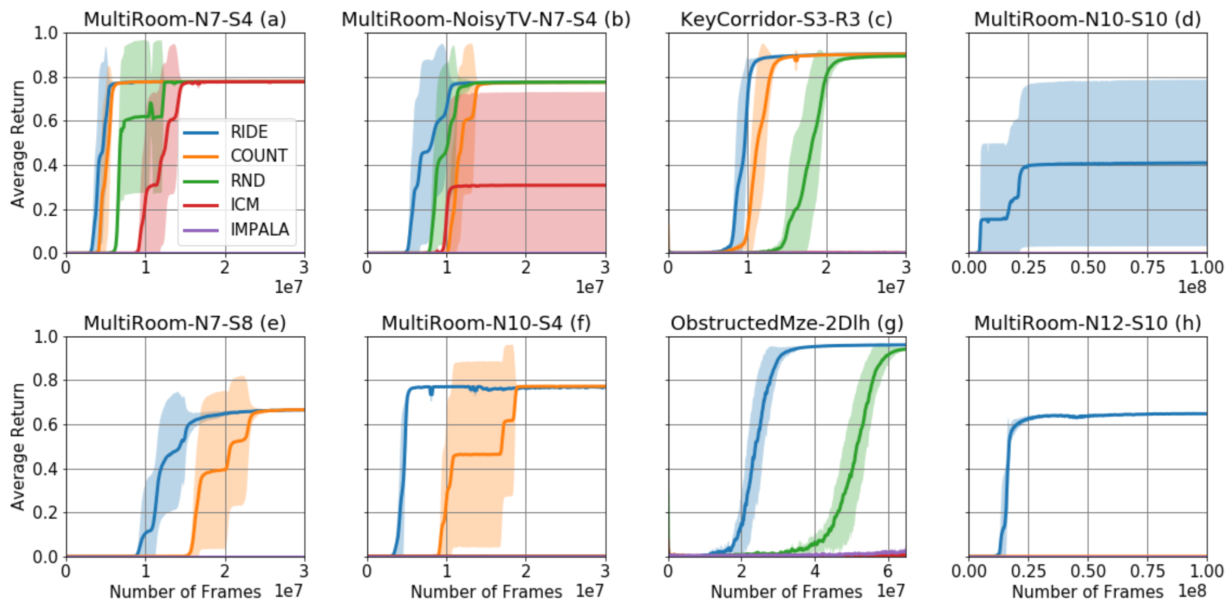


Figure 5.3: Performance of RIDE, Count, RND, ICM and IMPALA on a variety of hard exploration problems in MiniGrid. Note RIDE is the only one that can solve the hardest tasks.

In addition to existing MiniGrid tasks, we also tested the model's ability to deal with stochasticity in the environment by adding a "noisy TV" in the MiniGridN7S4 task, resulting in the new MiniGirdN7S4NoisyTV task (left-center plot in the top row of Figure 5.3). The noisy TV is implemented as a ball that changes its color to a randomly picked one whenever the agent takes a particular action. As expected, the performance of ICM drops as the agent becomes attracted to the ball while obtaining intrinsic rewarded for not being able to predict the next color. The Count model also needs more time to train, likely caused by the increasing number of rare and novel states (due to the changing color of the ball).

5.5.4 Ablations

In this section, we aim to better understand the effect of using episodic discounting as part of the intrinsic reward, as well as that of using entropy regularization as part of the IMPALA loss.

Figure 5.4 compares the performance of our model on different MiniGrid tasks with that of three ablations. The first one only uses episodic state counts as exploration bonus without multiplying it by the impact-driven intrinsic reward (*OnlyEpisodicCounts*), the second one only uses the impact-driven exploration bonus without multiplying it by the episodic state count term (*NoEpisodicCounts*), while the third one is the *NoEpisodicCounts* model without the entropy regularization term in the IMPALA loss (*NoEntropyNoEpisodicCounts*).

OnlyEpisodicCounts does not solve any of the tasks. *NoEntropyNoEpisodicCounts* either converges to a suboptimal policy or completely fails. In contrast, *NoEpisodicCounts* can solve the easier tasks but it requires more interactions than RIDE and fails to learn on the hardest domain. During training, *NoEpisodicCounts* can get stuck cycling between two states (with a large distance in the embedding states) but due to entropy regularization, it can sometimes escape such local optima (unlike *NoEntropyNoEpisodicCounts*) if it finds extrinsic reward. However, when the reward is too sparse, *NoEpisodicCounts* is insufficient while RIDE still succeeds, indicating the effectiveness of augmenting the impact-driven intrinsic reward with the episodic count term.

Figure 5.5 shows the average number of states visited during an episode of MultiRoomN12S10, measured at different training stages for our full RIDE model and the *NoEpisodicCounts* ablation.

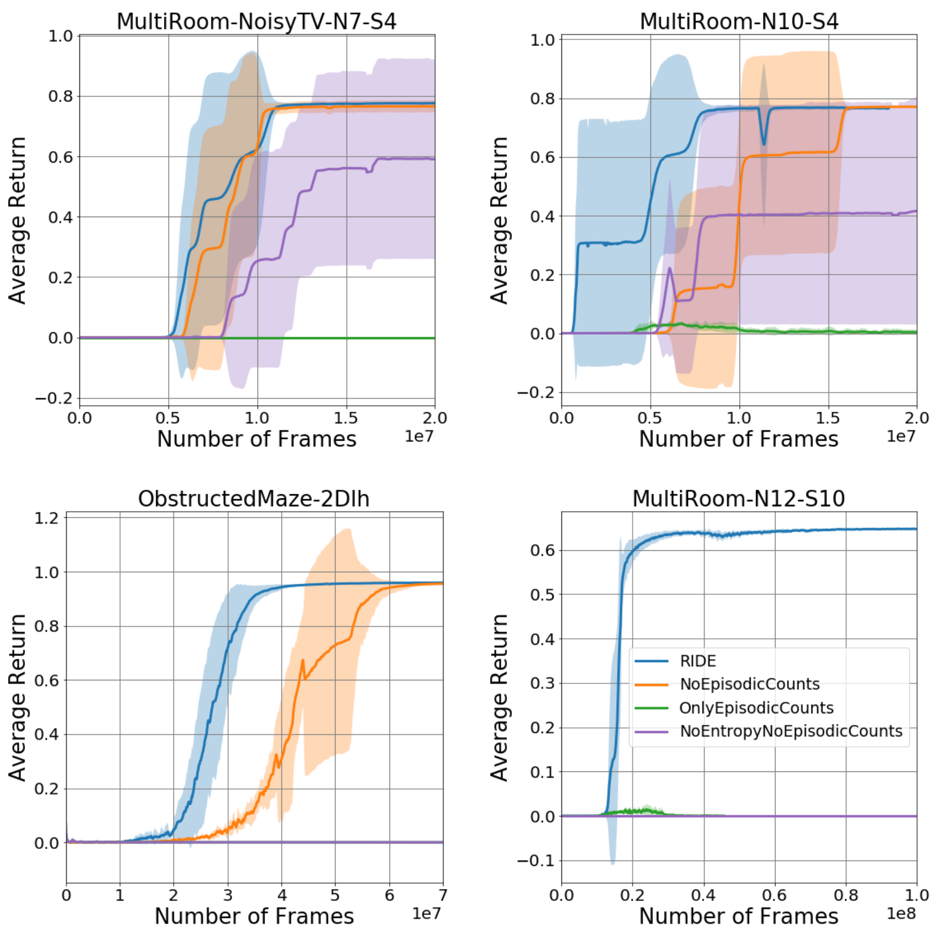


Figure 5.4: Comparison between the performance of RIDE and three ablations: *OnlyEpisodicCounts*, *NoEpisodicCounts*, and *NoEntropyNoEpisodicCounts*.

While the *NoEpisodicCounts* ablation always visits a low number of different states each episode (≤ 10), RIDE visits an increasing number of states throughout training (converging to ~ 100 for an optimal policy). Hence, it can be inferred that *NoEpisodicCounts* revisits some of the states. This claim can be further verified by visualizing the agents' behaviors. After training, *NoEpisodicCounts* goes back and forth between two states, while RIDE visits each state once on its path to the goal. Consistent with our intuition, discounting the intrinsic reward by the episodic state-count term does help to avoid this failure mode.

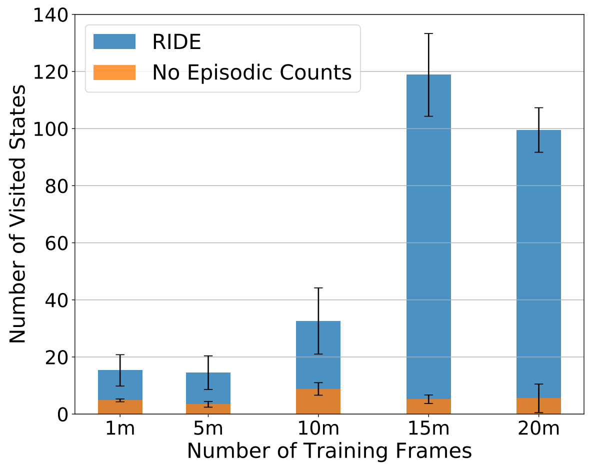


Figure 5.5: Average number of states visited during an episode of MultiRoomN12S10, measured at different training stages for our full RIDE model (blue) and the *NoEpisodicCounts* ablation (orange).

5.5.5 Analysis of the Intrinsic Reward

To better understand the effectiveness of different exploration methods, we investigate the intrinsic reward an agent receives for certain trajectories in the environment.

Figure 5.6 shows a heatmap of the intrinsic reward received by RND, ICM, and RIDE on a sampled environment after having been trained on procedurally generated environments from the MultiRoomN7S4 task. While all three methods can solve this task, the intrinsic rewards received are different. Specifically, the RIDE agent is rewarded in a much more structured manner for opening doors, entering new rooms and turning at decision points. Table 5.1 provides quantitative numbers for this phenomenon. We record the intrinsic rewards received for each type of action, averaged over 100 episodes. We found that RIDE is putting more emphasis on actions interacting with the door than for moving forward or turning left or right, while the other methods reward actions more uniformly.

In order to understand how various interactions with objects are rewarded by the different exploration methods, we also looked at the intrinsic reward in the ObstructedMaze2Dlh environment which contains multiple objects . However, the rooms are connected by locked doors and

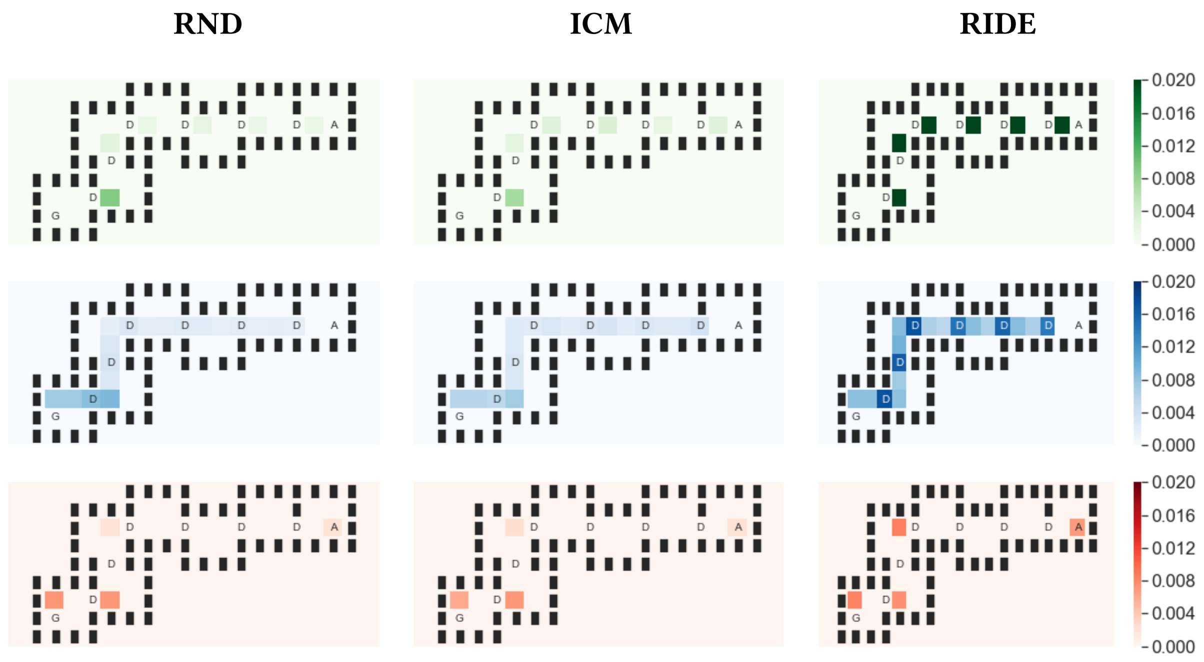


Figure 5.6: Intrinsic reward heatmaps for RND, ICM, and RIDE (from left to right) for opening doors (green), moving forward (blue), or turning left or right (red) on a random environment from the MultiRoomN7S4 task. A is the agent's starting position, G is the goal position and D are doors that have to be opened on the way.

	Open Door		Turn Left / Right		Move Forward	
Model	**Mean**	**Std**	**Mean**	**Std**	**Mean**	**Std**
RIDE	0.0490	0.0019	0.0071	0.0034	0.0181	0.0116
RND	0.0032	0.0018	0.0031	0.0028	0.0026	0.0017
ICM	0.0055	0.0003	0.0052	0.0003	0.0056	0.0003

Table 5.1: Mean intrinsic reward per action over 100 episodes on a random maze in MultiRoomN7S4.

the keys for unlocking the doors are hidden inside boxes. The agent does not know in which room the ball is located and it needs the color of the key to match that of the door in order to open it. Moreover, the agent cannot hold more than one object so it needs to drop one in order to pick up another.

Figure 5.7 and Table 5.2 indicate that RIDE rewards the agent significantly for interacting with various objects (e.g. opening the box, picking up the key, opening the door, dropping the key, picking up the ball) relative to other actions such as moving forward or turning left and right. In

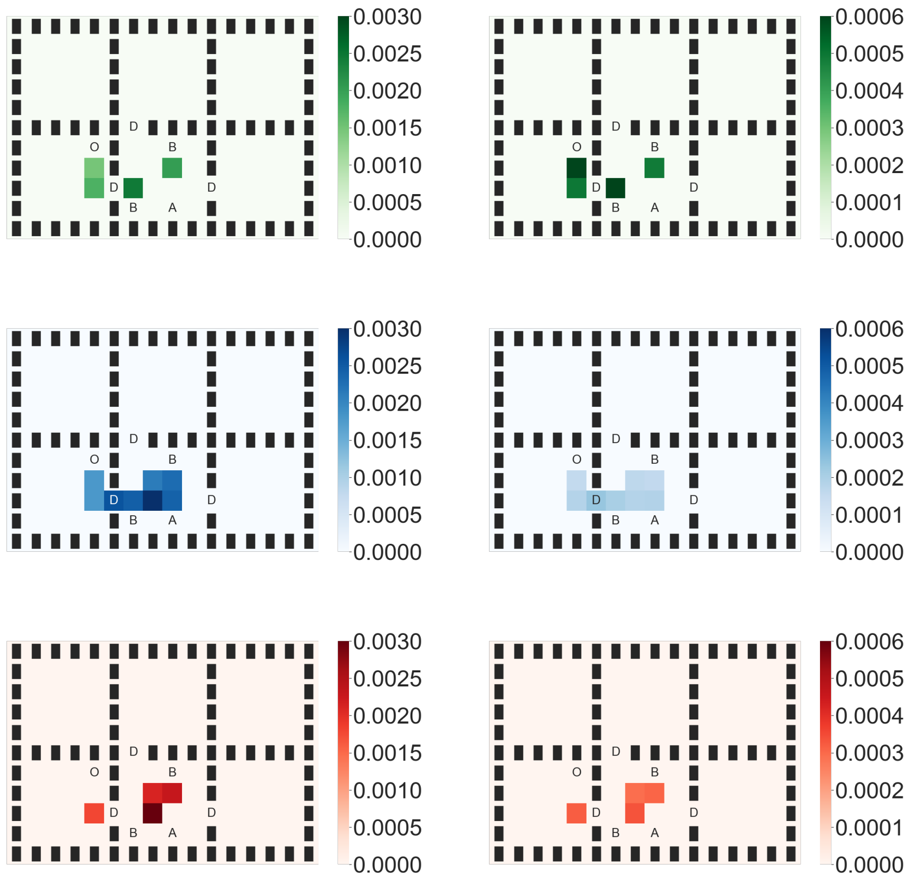


Figure 5.7: Intrinsic reward heatmaps for RND (left) and RIDE (right) for interacting with objects (i.e. open doors, pick up / drop keys or balls) (green), moving forward (blue), or turning left or right (red) on a random map from ObstructedMaze2Dlh. A is the agent's starting position, K are the keys hidden inside boxes (that need to be opened in order to see their colors), D are colored doors that can only be opened by keys with the same color, and B is the ball that the agent needs to pick up in order to win the game. After passing through the door the agent also needs to drop the key in order to be able to pick up the ball since it can only hold one object at a time.

	Open Door		**Pick Ball**		**Pick Key**		**Drop Key**		**Other**	
Model	**Mean**	**Std**	**Mean**	**Std**	**Mean**	**Std**	**Mean**	**Std**	**Mean**	**Std**
RIDE	0.0005	0.0002	0.0004	0.0001	0.0004	0.00001	0.0004	0.00007	0.0003	0.00001
RND	0.0034	0.0015	0.0027	0.0006	0.0026	0.0060	0.0030	0.0010	0.0025	0.0006

Table 5.2: Mean intrinsic reward per action computed over 100 episodes on a random map from ObstructedMaze2Dlh.

contrast, RND again rewards all actions much more uniformly and often times, within an episode, it rewards the interactions with objects less than the ones for moving around inside the maze.

5.5.6 Singleton versus Procedurally Generated Environments

It is important to understand and quantify how much harder it is to train existing deep RL exploration methods on tasks in procedurally generated environments compared to a singleton environment.

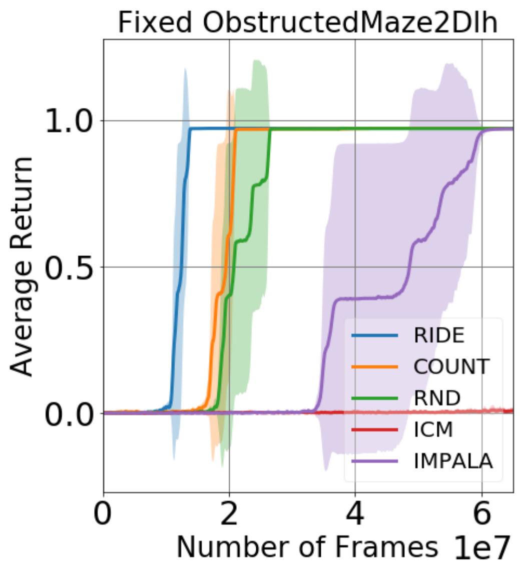


Figure 5.8: Training on a singleton instance of ObstructedMaze2Dlh.

To investigate this dependency, we trained the models on a singleton environment of the the *ObstructedMaze2Dlh* task so that at the beginning of every episode, the agent is spawned in exactly the same maze with all objects located in the same positions. In this setting, we see that Count, RND, and IMPALA are also able to solve the task (see Figure 5.8 and compare with the center-right plot in the bottom row of Figure 5.3 for procedurally generated environments of the same task). As expected, this emphasizes that training an agent in procedurally generated environments creates significant challenges over training on a singleton environment for the same task. Moreover, it highlights the importance of training on a variety of environments to avoid overfitting to the idiosyncrasies of a particular environment.

5.5.7 No Extrinsic Reward

To analyze the way different methods explore environments without depending on the chance of running into extrinsic reward (which can dramatically change the agent's policy), we analyze agents that are trained without any extrinsic reward on both singleton and procedurally generated environments.

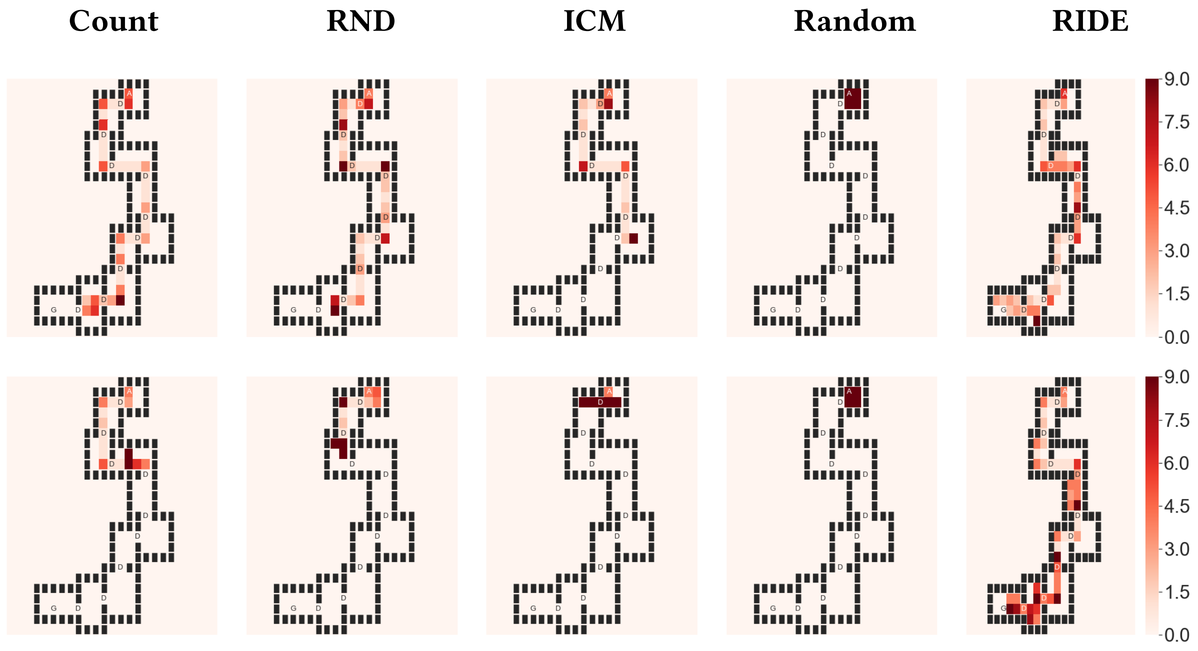


Figure 5.9: State visitation heatmaps for Count, RND, ICM, Random, and RIDE models (from left to right) trained for 50m frames without any extrinsic reward on a singleton maze (top row) and on procedurally generated mazes (bottom row) in MultiRoomN10S6.

The top row of Figure 5.9 shows state visitation heatmaps for all the models in a singleton environment on the MultiRoomN10S6 task, after training all of them for 50M frames with intrinsic reward only. The agents are allowed to take 200 steps in every episode. The figure indicates that all models have effective exploration strategies when trained on a singleton maze, the 10th, 9th and 6th rooms are reached by RIDE, Count/RND, and ICM, respectively. The Random policy fully explores the first room but does not get to the second room within the time limit.

When trained on procedurally generated mazes, existing models are exploring much less efficiently as can be seen in the bottom row of Figure 5.9. Here, Count, RND, and ICM only make it to the 4th, 3rd and 2nd rooms respectively within an episode, while RIDE is able to explore all rooms. This further supports that RIDE learns a state representation that allows generalization across different mazes and is not as distracted by less important details that change from one procedurally generated environment to another.

5.5.8 Mario and VizDoom

In order to compare to Pathak et al. [2017], we evaluate RIDE on the first level of the Mario environment. Our results (see Figure 5.10 a and b) suggest that this environment may not be as challenging as previously believed, given that all the methods evaluated here, including vanilla IMPALA, can learn similarly good policies after training on only 1m frames even without any intrinsic reward (left figure). Note that we are able to reproduce the results mentioned in the original ICM paper [Pathak et al. 2017]. However, when training with both intrinsic and extrinsic reward (center figure), the curiosity-based exploration bonus (ICM) hurts learning, converging later and to a lower value than the other methods evaluated here.

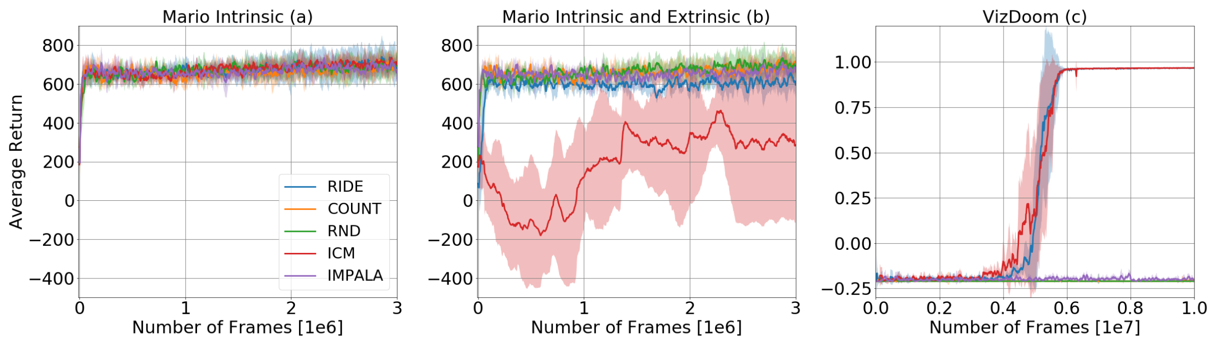


Figure 5.10: Performance on Mario with intrinsic reward only (a), with intrinsic and extrinsic reward (b), and VizDoom (c). Note that IMPALA is trained with extrinsic reward only in all cases.

For VizDoom (see Figure 5.10 c) we observe that RIDE performs as well as ICM, while all the other baselines fail to learn effective policies given the same amount of training. Note that our ICM implementation can reproduce the results in the original paper on this task, achieving a 100% success rate after training on approximately 60m frames [Pathak et al. 2017].

5.6 Conclusion

In this chapter, we presented Rewarding Impact-Driven Exploration (RIDE), an intrinsic reward bonus that encourages agents to explore actions that substantially change the state of the environment, as measured in a learned latent space. RIDE has a number of desirable properties: it attracts agents to states where they can affect the environment, it provides a signal to agents even after training for a long time, and it is conceptually simple as well as compatible with other intrinsic or extrinsic rewards and any deep RL algorithm.

Our approach is particularly effective in procedurally generated sparse reward environments where it significantly outperforms IMPALA [Espeholt et al. 2018b], as well as some of the most popular exploration methods such as Count [Bellemare et al. 2016], RND [Burda et al. 2019b], and ICM [Pathak et al. 2017]. Furthermore, RIDE explores procedurally generated environments more efficiently than other exploration methods.

However, there are still many ways to improve upon RIDE. For example, one can make use of symbolic information to measure or characterize the agent's impact, consider longer-term effects of the agent's actions, or promote diversity among the kinds of changes the agent makes to the environment. Another interesting avenue for future research is to develop algorithms that can distinguish between desirable and undesirable types of impact the agent can have in the environment, thus constraining the agent to act safely and avoid distractions (*i.e.* actions that lead to large changes in the environment but that are not useful for a given task). The different kinds of impact might correspond to distinctive skills or low-level policies that a hierarchical controller could use to learn more complex policies or better exploration strategies.

Our work was the first to study how the problem of exploration changes when considering a wide range of task instances rather than a single one. During our investigation, we discovered that some of the state-of-the-art methods in singleton environments have significant limitations in procedurally generated ones. Since our work was published, RIDE has become a common baseline for papers studying exploration in procedurally generated environnments [Song and Kushnir 2020; Zhang et al. 2020d; Campero et al. 2020; Fang et al. 2020]. Other researchers have extended our method and proposed alternative approaches, thus making further progress in this area. For example, Song and Kushnir [2020] use the same impact-driven exploration bonus as introduced in this chapter, but learn the state representations via a contrastive loss instead of forward and inverse losses. Inspired by our episodic visitation counts, Zhang et al. [2020d] use a regulated difference of inverse visitation counts to encourage exploration beyond the boundary of the already visited regions.

6 | Fast Adaptation to New Environments via Policy-Dynamics Value Functions

Standard RL algorithms assume fixed environment dynamics and require a significant amount of interaction to adapt to new environments. In this chapter, we introduce Policy-Dynamics Value Functions (PD-VF), a novel approach for rapidly adapting to dynamics different from those previously seen in training. PD-VF explicitly estimates the cumulative reward in a space of policies and environments. An ensemble of conventional RL policies is used to gather experience on training environments, from which embeddings of both policies and environments can be learned. Then, a value function conditioned on both embeddings is trained. At test time, a few actions are sufficient to infer the environment embedding, enabling a policy to be selected by maximizing the learned value function (which requires no additional environment interaction). We show that our method can rapidly adapt to new dynamics on a set of MuJoCo domains.

6.1 Introduction

Recent studies have pointed out that RL agents trained and tested on the same environment tend to overfit to that environment's idiosyncracies and are unable to generalize to even small perturbations [Whiteson et al. 2011; Rajeswaran et al. 2017b; Zhang et al. 2018d,a; Henderson et al. 2018; Cobbe et al. 2019d; Raileanu and Rocktäschel 2020; Song et al. 2020]. It is often the case that besides the test environments being different from the train environments, they will also have costly interactions, scarce or unavailable feedback, and irreversible consequences. For example, a self-driving car might have to adjust its behavior depending on weather conditions, or a prosthetic control system might have to adapt to a new human. In these cases it is crucial for RL agents to find and execute appropriate policies as quickly as possible.

Our approach is inspired by Sutton et al. [2011] who introduced the notion of general value functions (GVFs), which can be used to gather knowledge about the world in the form of predictions. A GVF estimates the expected return of an arbitrary policy on a certain task (as defined by a reward function, a termination function and a terminal-reward function). Similarly, in this work, we aim to learn a value function conditioned on elements of a space of policies and tasks, but here, a "task" is specified by the transition function of the MDP instead of the reward function.

More specifically, we propose PD-VF, a novel framework for rapid adaptation to new environment dynamics. PD-VF consists of four phases: (i) a *reinforcement learning phase* in which individual policies are learned for each environment in our training set using standard RL algorithms, (ii) a *self-supervised phase* in which trajectories generated by these policies are used to learn embeddings for both policies and environments, (iii) a *supervised training phase* in which a neural network is used to learn the value function of a certain policy acting in some environment. The network takes as inputs the initial state of the environment, as well as the corresponding policy and environment embeddings (as learned in the previous phase) and is trained with supervision of the cumulative reward obtained during an episode, and finally (iv) an *evaluation phase* in which, given a new environment, its dynamics embedding is inferred using the first few steps of an episode. Then, a policy is selected by finding the policy embedding that maximizes the learned value function. The selected policy is used to act in the environment until the episode ends.

Our framework uses self-supervised interactions with the environment to learn an embedding space of both dynamics and policies. By learning a value function in the policy-dynamics space, PD-VF can discover useful patterns in the complex relation between a family of environment dynamics, various behaviors, and the expected return. The value function is designed to model non-optimal policies along with optimal policies in given environments so that it can understand how changes in dynamics relate to changes in the return of different policies. PD-VF uses the learned space of dynamics to rapidly embed a new environment in that space using only a few interactions. At test time, PD-VF can evaluate or rank policies (from a certain family) on unseen environments without the need of full rollouts (*i.e.* it does not require full trajectories or rewards to update the policy). We evaluate our method on a set of continuous control tasks (with varying dynamics) in MuJoCo [Todorov et al. 2012a]. The dynamics of each task instance are determined by physical parameters such as wind direction or limb length and can be sampled from a continuous or discrete distribution. Performance is evaluated on a single episode at test time to emphasize rapid adaptation. We show that PD-VF outperforms other meta-learning and transfer learning approaches on new environments with unseen dynamics.

6.2 Related Work

Our work draws inspiration from multiple research areas such as transfer learning [Taylor and Stone 2009; Higgins et al. 2017], skill and task embedding [Devin et al. 2016; Zhang et al. 2018c; Hausman et al. 2018; Petangoda et al. 2019], and general value functions [Precup et al. 2001; Sutton et al. 2011; White et al. 2012].

Multi-Task and Transfer Learning. Taylor and Stone [2009] presents an overview of transfer learning methods in RL. A popular approach for transfer in RL is multi-task learning [Taylor and Stone 2009; Teh et al. 2017], a paradigm in which an agent is trained on a family of related tasks. By simultaneously learning about different tasks, the agent can exploit their common structure, which can lead to faster learning and better generalization to unseen tasks from the same family [Taylor and Stone 2009; Lazaric 2012; Ammar et al. 2012, 2014; Parisotto et al. 2015; Borsa et al. 2016; Gupta et al. 2017; Andreas et al. 2017; Oh et al. 2017; Hessel et al. 2019]. A large body of work has been inspired by the Horde architecture [Sutton et al. 2011], which consists of a number of RL agents with different policies and goals. Each agent is tasked with estimating the value function of a particular policy on a given task, thus collectively representing knowledge about the world. Building on these ideas, other methods leverage the shared dynamics of the tasks [Barreto et al. 2017; Zhang et al. 2017; Madjiheurem and Toni 2019] or the similarity among value functions and the associated optimal policies [Schaul et al. 2015; Borsa et al. 2018; Hansen et al. 2019; Siriwardhana et al. 2019]. These approaches assume the same underlying transition function for all tasks. In contrast, we focus on transferring knowledge across tasks with different dynamics.

Meta-Learning and Robust Transfer. A popular approach for fast adaptation to new environments is meta reinforcement learning (meta RL) [Cully et al. 2015; Finn et al. 2017; Wang et al. 2017; Duan et al. 2016; Xu et al. 2018; Houthooft et al. 2018; Sæmundsson et al. 2018; Nagabandi et al. 2018; Humplik et al. 2019; Rakelly et al. 2019]. Meta RL methods have been designed to work well with dense reward and recent work has shown that they struggle to learn from a limited number of interactions and optimization steps at test time [Yang et al. 2019]. In contrast, our framework is capable of rapid adaptation to new environment dynamics and does not require dense reward or a large number of interactions to find a good policy. Moreover, PD-VF does not update the model parameters at test time, which makes it less computationally expensive than meta RL. Another common approach for transfer across dynamics is model-based RL, which uses Gaussian processes (GPs) or Bayesian neural networks (BNNs) to estimate the transition function [Doshi-Velez and Konidaris 2013; Killian et al. 2017]. However, such methods require fictional rollouts to train a policy from scratch at test time, which makes them computationally expensive and limits their applicability for real-world tasks. Yao et al. [2018] uses a fully-trained BNN to further optimize latent variables during a single test episode, but requires an optimal policy for each training instance, which makes it harder to scale. Robust transfer methods either require a large number of interactions at test time [Rajeswaran et al. 2017b] or assume that the distribution over hidden variables is known or controllable [Paul et al. 2018]. An alternative approach was proposed by Pinto et al. [2017] who use an adversary to perturb the system, achieving robust transfer across physical parameters such as friction or mass.

Skill and Task Embeddings. A large body of work proposes the use of learned skill and task embeddings for transfer in RL [Da Silva et al. 2012; Sahni et al. 2017; Oh et al. 2017; Gupta et al. 2017; Hausman et al. 2018; He et al. 2018]. For example, Hausman et al. [2018] use approximate variational inference to learn a latent space of skills. Similarly, Arnekvist et al. [2018] learn a stochastic embedding of optimal Q-functions for various skills and train a universal policy conditioned on this embedding. In both Hausman et al. [2018] and Arnekvist et al. [2018], adaptation to a new task is done in the latent space with no further updates to the policy network. Co-Reyes et al. [2018] learn a latent space of low-level skills that can be controlled by a higher-level policy, in the context of hierarchical reinforcement learning. This embedding is learned using a variational autoencoder [Kingma and Welling 2013] to encode state trajectories and decode states and actions. Zintgraf et al. [2018] use a meta-learning approach to learn a deterministic task embedding. Wang et al. [2017] and Duan et al. [2017] learn embeddings of expert demonstrations to aid imitation learning using variational and deterministic methods, respectively. More recently, Perez et al. [2018] learn dynamic models with auxiliary latent variables and use them for model-predictive control. Zhang et al. [2018c] use separate dynamics and reward modules to learn a task embedding. They show that conditioning a policy on this embedding helps transfer to changes in transition or reward function. While the above approaches might learn embeddings of skills or tasks, none of them leverage *both* the latent space of policies and that of the environments for estimating the expected return and using it to select an effective policy at test time.

More similar to our work is that of Yang et al. [2019], who also focus on fast adaptation to new environment dynamics and evaluate performance on a single episode at test time. Yang et al. [2019] train an inference model and a probe to estimate the underlying latent variables of the dynamics, which are then used as input to a universal control policy. While similar in scope, our approach is significantly different from that of Yang et al. [2019]. Importantly, Yang et al. [2019] does not learn a latent space of policies and instead trains a universal policy on all the environments. Learning a value function in a space of policies and dynamics allows the function approximator to capture relations among dynamics, behaviors (both optimal as well as non-optimal), and rewards that a universal policy cannot learn. Moreover, the learned structure can aid transfer to new dynamics.

6.3 Background

In this work, we aim to design an approach that can quickly find a good policy in an environment with new and unknown dynamics, after being trained on a family of environments with related dynamics. The problem can be formalized as a family of Markov decision processes (MDPs) defined by $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$, where $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \gamma)$ are the corresponding state space, action space, reward function, and discount factor. Each instance of the family is a stationary MDP with transition function $\mathcal{P}_d(s'|s, a) \in \mathcal{P}$. Each $\mathcal{P}_d$ has a hidden parameter d that is sampled once from a distribution $\mathcal{D}$ and held constant for that instance (*i.e.* episode). $\mathcal{P}_d$ can be continuous or discrete in d. By design, the latent variable d that defines the MDP's dynamics cannot be observed from individual states. The dynamics distribution $\mathcal{D}$ is partitioned into two disjoint sets $\mathcal{D}_{train}$ and $\mathcal{D}_{test}$. These are used to generate a set of training and test environments, each having different transition functions, drawn from their respective distributions.

6.4 Approach

We present Policy-Dynamics Value Functions (PD-VF), a novel framework for rapid adaptation across such MDPs with different dynamics. PD-VF is an extension of a value function that not only conditions on a state, but also on a policy and a transition function.

A conventional value function $V : \mathcal{S} \to \mathcal{R}$ is defined as the expected future return from state s of policy π:

$$V(s) = \mathbb{E}\left[G_t | S_t = s\right] = \mathbb{E}\left[\sum_{k=t+1}^{T} \gamma^k r_k | S_t = s\right].$$

Formally, we define a *policy-dynamics value function* or PD-VF as a function $W : \mathcal{S} \times \Pi \times \mathcal{T} \to \mathcal{R}$ with two auxiliary inputs representing the policy π and the dynamics d:

$$W(s, \pi, d) = \mathbb{E}\left[G_t | S_t = s, A_t \sim \pi, S_{t+1} \sim \mathcal{T}_d\right].$$

Our model is learned on the training environments in three stages: (i) a reinforcement learning phase, (ii) a self-supervised phase and (iii) a supervised phase. The resulting PD-VF model is evaluated on test environments, where it only experiences a single episode in each. This evaluation setting probes PD-VF's ability to very quickly adapt to previously unseen dynamics.

6.4.1 Reinforcement Learning Phase

The first phase of training uses standard model-free RL algorithms to acquire experience in the training environments. An ensemble of N policies are trained, each with a different random seed on one of the training environments. For each policy, we save a number of checkpoints at different stages throughout training. Then, we collect trajectories using each of these checkpoints in each of our training environments. This results in experience from a diverse set of policies (some good, some bad) across environments with different dynamics. Importantly, this dataset contains the behaviors of policies in environments they haven't been trained on. In the next section, we describe how the collected trajectories are used to learn policy and dynamics embeddings.

6.4.2 Self-Supervised Learning Phase

The goal of this phase is to learn an embedding space of the dynamics that captures variations in the transition function, as well as an embedding space of the policies that captures variations in the agent behavior. The space of dynamics is learned using an encoder E_d parameterised as a

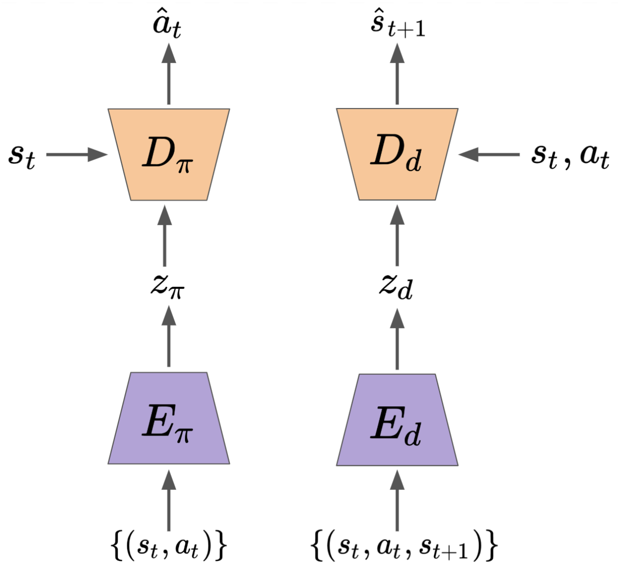


Figure 6.1: In the **self-supervised learning phase**, a pair of autoencoders is trained using transitions generated by a diverse set of policies in a set of environments with different dynamics. By exploiting the Markov property of the environment, distinct latent embeddings of the dynamics z_d and policy z_π are produced.

Transformer [Vaswani et al. 2017], and a decoder D_d parameterised as a feed-forward network. The encoder takes as input a *set* of transitions $\{(s_t, a_t, s_{t+1})\}$ from the first N_d steps in each episode and outputs a vector embedding for the dynamics z_d. The decoder takes as inputs the state s_t, action a_t and dynamics embedding z_d, and predicts the next state $\hat{s}_{t+1}$. The parameters θ_d and ϕ_d of the encoder and decoder are trained to minimize the ℓ_2 error of $\hat{s}_{t+1}$ and s_{t+1}. Formally,

$$z_d = E_d(\{(s_t, a_t, s_{t+1})\};\ \theta_d)$$

$$\hat{s}_{t+1} = D_d(s_t, a_t, z_d;\ \phi_d).$$

This arrangement exploits the inductive bias that, conditioned on d, the environment is Markovian. By using no positional encoding in the Transformer, the input transitions lack any temporal ordering, thus preserving the Markov property. The decoder receives no historical information (since it is unnecessary in a Markovian setting), so it is forced to embed information about the dynamics into z_d to make good predictions. Because the input set contains the actions in each triple, the encoder has no incentive to encode policy information into z_d. This modeling choice encourages z_d to only contain information about the dynamics, rather than the policy used to generate the transitions.

Similarly, the space of policies is learned using an encoder E_π parameterised as a Transformer and a decoder D_π parameterised as a feed-forward network. The encoder takes as input a set (again using the Markov property as an inductive bias) of state-action pairs $\{(s_t, a_t)\}$ from a full episode and outputs a vector embedding for the policy z_d. The decoder takes as inputs the state s_t and the policy embedding z_π to predict the action taken by the policy $\hat{a}_t$. Since the policy encoder does not have direct access to full environment transitions, z_π is constrained to capture information about the policy without elements of the dynamics. The parameters θ_π and ϕ_π of the encoder and decoder are trained to minimize the ℓ_2 error of $\hat{a}_t$ and a_t. Formally,

$$z_\pi = E_\pi(\{(s_t, a_t)\};\ \theta_\pi)$$

$$\hat{a}_t = D_\pi(s_t, z_\pi;\ \phi_\pi).$$

Both the policy and the dynamics embeddings are normalized to have unit ℓ_2-norm.

See Figure 6.1 for an overview of the self-supervised learning phase.

6.4.3 Supervised Learning Phase

In this phase, the goal is to train an estimator W of the expected return $\hat{G}$ for a space of policies and dynamics. More specifically, W is a function approximator conditioned on the learned policy

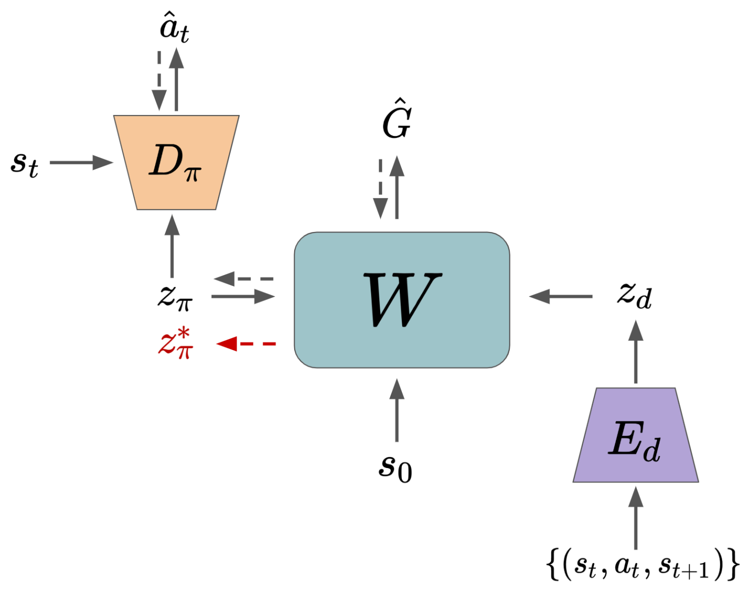


Figure 6.2: In the **supervised learning phase**, a parametric value function W is trained to predict the expected return G for an entire space of policies and dynamics. W takes as inputs the initial state s_0, policy embedding z_π, and dynamics embedding z_d (estimated from a small set of transitions). We train W in a supervised fashion, using Monte-Carlo estimates of the expected return G for policy π in environment with dynamics $\mathcal{T}_d$. At test time, z_π is optimized to maximize $\hat{G}$ (red dashed arrow), resulting in z_π^* which is then decoded to an actual policy via D_π.

and dynamics embeddings, z_π and z_d.

A central idea of our PD-VF framework is that W provides a scoring function over the policy embedding space. It thus provides a mechanism to allow on-the-fly optimization of z_π with respect to the estimated return $\hat{G}$, without the need for any environment interaction, given an estimate (or embedding) of the environment's dynamics. This is key to PD-VF's ability to rapidly find an effective policy in a new environment, only requiring enough environment interaction to give a reliable estimate of the dynamics embedding z_d (just a few steps in practice). We choose W to have a quadratic form to permit easy optimization with respect to z_π:

$$\hat{G} = W(s_0, z_\pi, z_d) = z_\pi^T A(s_0, z_d; \psi)\, z_\pi.$$

The matrix $A(s_0, z_d; \psi)$ is a function of the initial environment state s_0 as well as the dynamics embedding z_d. Note that A only needs to model the initial state s_0 rather than an arbitrary state s since the optimization w.r.t z_π occurs only once, at the start of an episode. Since A must be Hermitian positive-definite, a feed-forward network with parameters ψ is first used to obtain a lower triangular matrix $L(s_0, z_d; \psi)$. Then A is constructed from LL^T.

Optimizing the policy embedding z_π: The optimization of the policy embedding z_π has a closed-form solution which is achieved by performing a singular value decomposition, $A = USV^T$, and taking the top singular vector of this decomposition z_π^*. Unit ℓ_2 normalization is then applied to z_π^*. We refer to this vector z_π^* as the *optimal policy embedding* (OPE) of the PD-VF.

Learning ψ – Initial stage: We collect training data for the PD-VF in the following manner. First, we randomly select a policy and an environment from our training set (described in Section 6.4.1). Second, we generate full trajectories of that policy in the selected environment and cache the average return obtained across all episodes. This gives us a Monte-Carlo estimate for the expected return of the corresponding policy in that particular environment. Then, we use the first N_d steps of that trajectory to infer the dynamics embedding. Similarly, we use the full trajectory to infer the policy embedding (via E_π, not the above optimization procedure). After collecting this data into a buffer, we train the estimator W in a supervised fashion by predicting the expected return G given an initial state s_0, a policy embedding z_π and a dynamics embedding z_d.

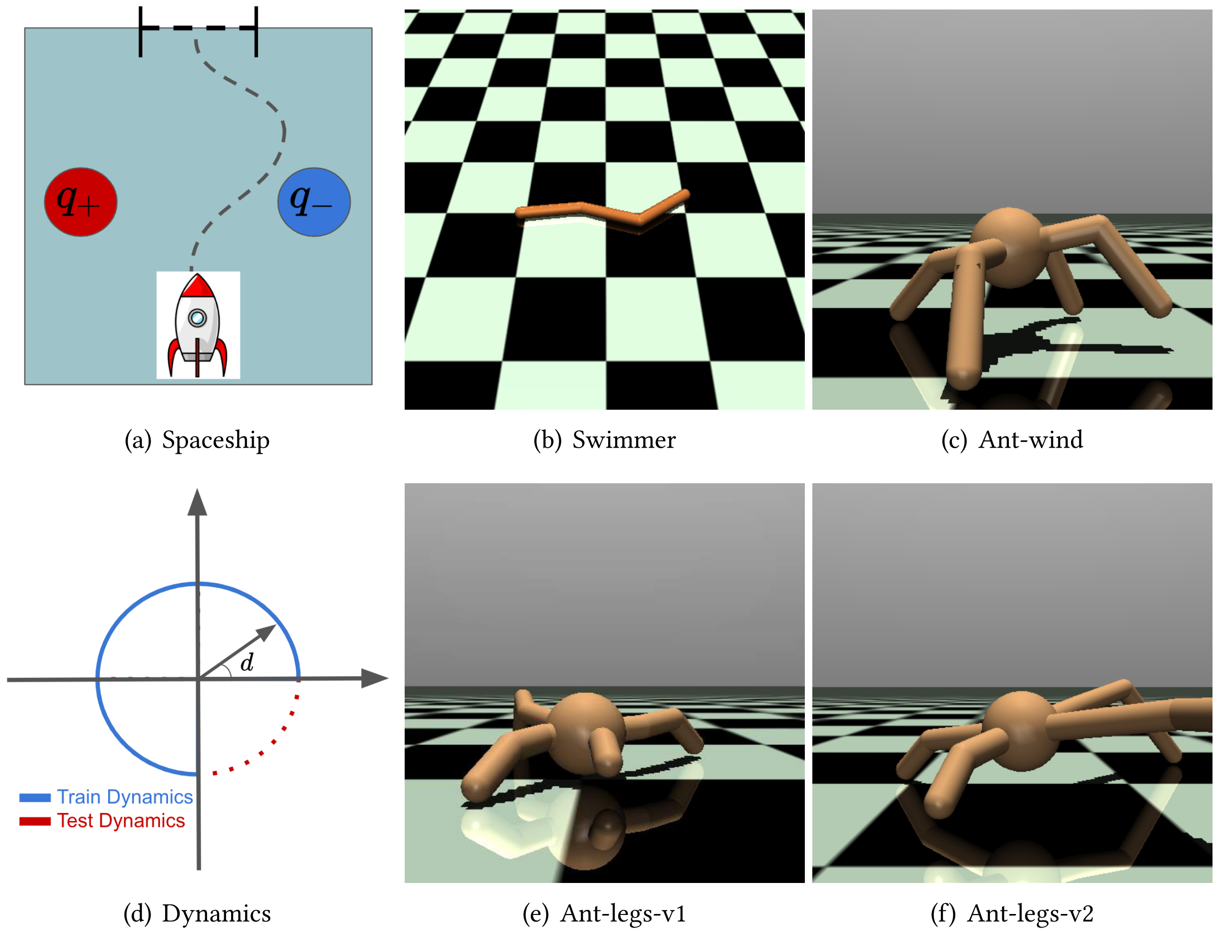
Learning ψ – Data Aggregation for the Value Function: For the method to work well, it is important that the learned value function W makes accurate predictions for the entire policy space, and especially for the OPE z_π^* (which correspond to the policies selected to act in the environment). One way to ensure that these estimates are accurate is by adding the OPEs to the training data. After initial training of the PD-VF on the original dataset of policy and dynamics embeddings, we use an iterative algorithm that alternates between collecting a new dataset of OPEs and training the PD-VF on the aggregated data (including the original data as well as data added from all previous iterations). We use early stopping to select the best value function (*i.e.* the one with the lowest loss) to be used at test time.

Learning ψ – Data Aggregation for the Policy Decoder: Similarly, the policy decoder may poorly estimate an agent's actions in states not seen during training. Thus, we iteratively train the policy decoder using a combination of the original set of states as well as new states generated by the policy embeddings that maximize the current value function. More specifically, we use the current OPEs (corresponding to the policies that PD-VF thinks are best) as inputs to the policy decoder to generate actions and interact with the environment. Then, we add the states visited by this policy to the data. The policy decoder is trained using the aggregated collection of states which includes both the states visited by the original collection of policies as well as the states visited by the current OPEs selected by the PD-VF.

See Figure 6.2 for an overview of the supervised learning phase.

6.4.4 Evaluation Phase

At test time, we want to find a policy that performs well on a single episode of an environment with unseen dynamics. This proceeds as follows: (i) the agent uses one of the pretrained RL policies to act for N_d steps; (ii) the generated transitions are then used to infer the dynamics embedding z_d; (iii) once an estimate of the dynamics is obtained, the matrix $A(s_0, z_d;\ \psi)$ can be computed; (iv) we employ the closed-form optimization described above to compute the optimal policy embedding z_π^*; (v) the policy decoder, conditioned on the z_π^* embedding, is then used to take actions in the environment until the end of the episode. Note that only a small number of interactions with a new environment is needed in order to adapt, the policy selection being performed internally within the PD-VF model. Performance is evaluated on a single trajectory of each environment instance.



(a) Spaceship (b) Swimmer (c) Ant-wind

(d) Dynamics (e) Ant-legs-v1 (f) Ant-legs-v2

Figure 6.3: (a) - (c) illustrate the continuous control domains used for testing adaptation to unseen environment dynamics. In Spaceship, Swimmer, and Ant-wind, the train and test distribution of the dynamics is continuous as illustrated in (d). (e) and (f) show two instances of the Ant-legs task in which limb lengths sampled from a discrete distribution determine the dynamics.

6.5 Experiments

6.5.1 Experimental Setup

We evaluate PD-VF on four continuous control domains, and compare it with an upper bound, four baselines, and four ablations. For each domain, we create a number of environments with different dynamics. Then, we split the set of environments into training and test subsets, so that at test time, the agent has to find a policy that behaves well on unseen dynamics. For all our

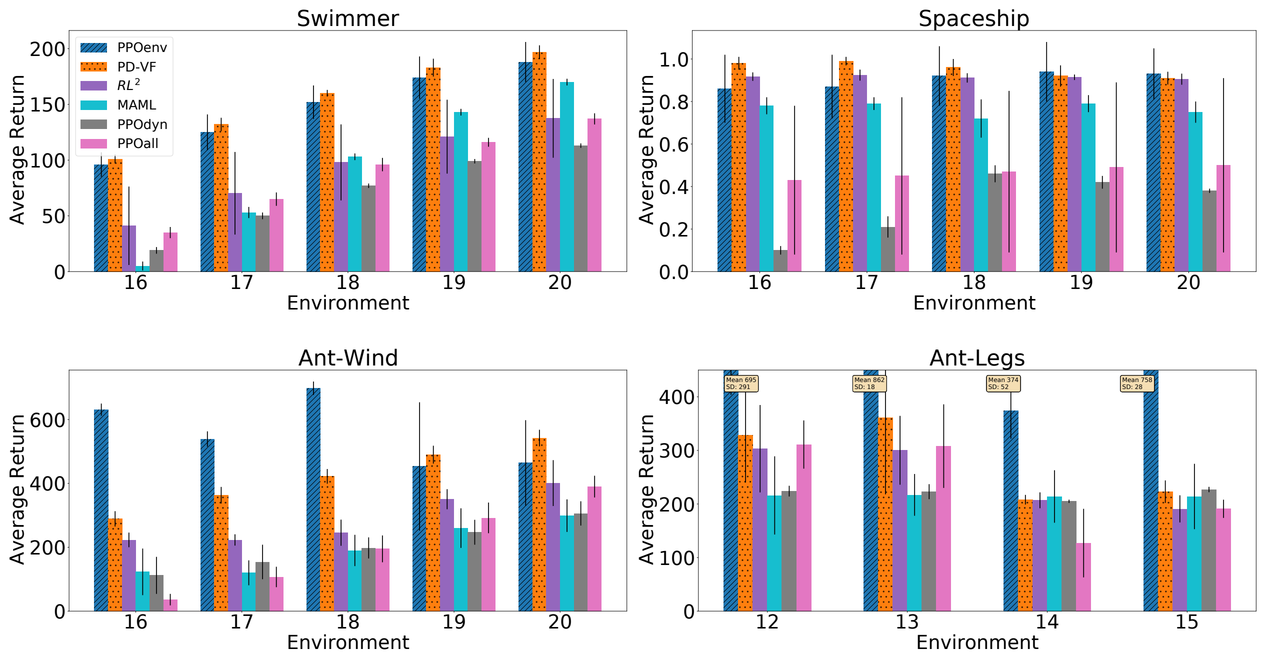


Figure 6.4: Test Performance. Average return on test environments with unseen dynamics in Swimmer (top-left), Spaceship (top-right), Ant-wind (bottom-left), and Ant-legs (bottom-right) obtained by PD-VF, the upper bound PPOenv, and baselines RL^2, MAML, PPOdyn, and PPOall. PD-VF outperforms these baselines on most test environments and, in some cases, it is comparable with PPOenv (which was trained directly on the test environments).

experiments, we show the mean and standard deviation of the average return (over 100 episodes) across 5 different seeds of each model. The dynamics embeddings are inferred using at most $N_d = 4$ interactions with the environment.

6.5.2 Environments

Spaceship is a new continuous control domain designed by us. The task consists of moving a spaceship with a unit point charge from one end of a 2D room through a door at the other end. The action space consists of a fixed-magnitude force vector that is applied at each timestep. The room contains two fixed electric charges that deflect / attract the ship as it moves through the environment (see Figure 6.3(a)). The polarity and magnitude of these charges are parameterised by d and determine the environment dynamics. The distribution of dynamics $\mathcal{D}$ is chosen to be circular and centered (see Figure 6.3(d)). Samples d are drawn at intervals of $\pi/10$, each forming a different environment instance with charge configuration $(\cos(d), \sin(d))$. The 5 samples in the range $[\frac{3}{4}2\pi, \ldots, 2\pi]$ are held out as evaluation environments, the rest being used for training.

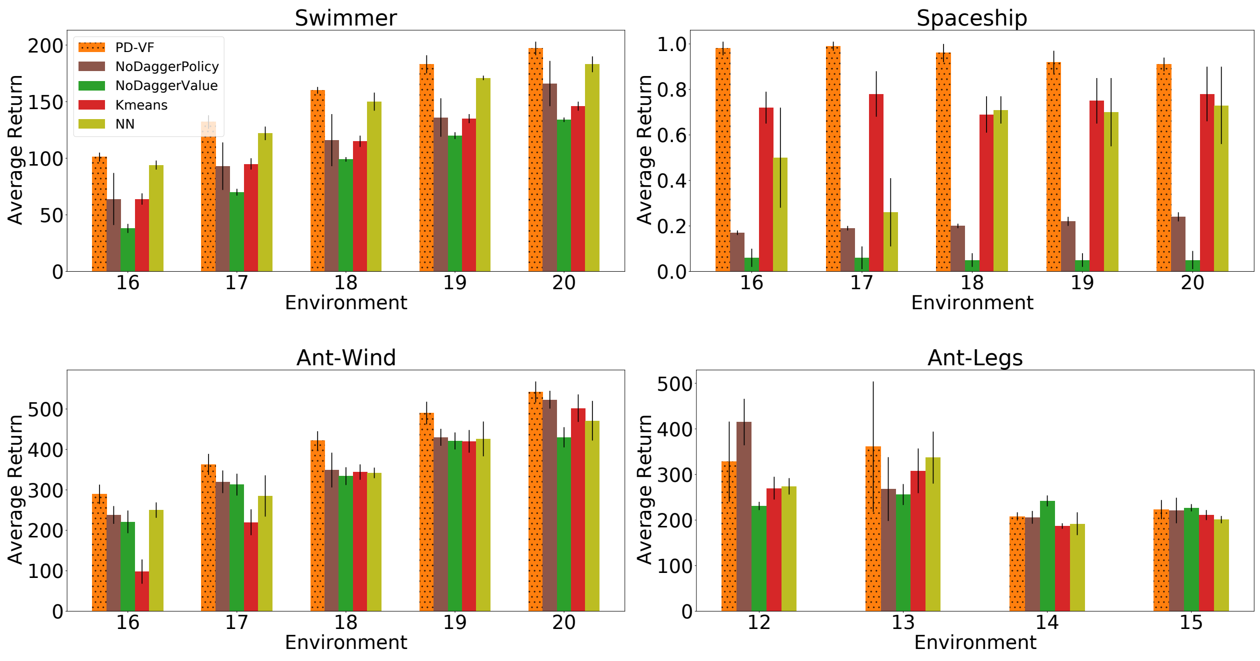


Figure 6.5: Test Performance. Average return in Swimmer (top-left), Spaceship (top-right), Ant-wind (bottom-left), and Ant-legs (bottom-right) obtained by PD-VF, NoDaggerPolicy, NoDaggerValue, Kmeans, and NN. PD-VF is better than these ablations overall.

Swimmer is a family of environments with varying dynamics based on MuJoCo's Swimmer-v3 domain [Todorov et al. 2012a]. The goal is to control a three-link robot in a viscuous fluid to swim forward as fast as possible (Figure 6.3(b)). The dynamics are determined by a 2D current within the fluid, whose direction changes between environments (but has fixed magnitude). The current direction is determined by an angle d, which is sampled in the same manner as for Spaceship above, *i.e.* train on 3/4 of all possible directions and hold out the other 1/4 for evaluation.

Ant-wind is a family of environments based on MuJoCo's Ant-v3 domain in which the goal is to make a four-legged creature walk forward as fast as possible (Figure 6.3(c)). The environment dynamics are determined by the direction of a wind d, which is sampled from a continuous distribution in the same way as for Swimmer.

Ant-legs is a second task based on MuJoCo's Ant-v3 domain, in which the dynamics are sampled from a discrete distribution. The training environments are generated by fixing three ankle lengths (short, medium, and long) and generating all possible permutations for the four legs. The length of the ant leg is fixed to medium across all training environments. Symmetries in the training environments are removed by considering ants with the same number of short, medium, or long legs to be the same and choosing one ant from each equivalency class. There are four test environments with both the leg and ankle lengths being either short or long. Note that the test environments are significantly different from all the training ones, thus making Ant-legs a challenging setting for our method. Figures 6.3(e) and 6.3(f) show two instances of this environment.

6.5.3 Baselines

We use PPO [Schulman et al. 2017] as the base RL algorithm for all the baselines and for the reinforcement learning phase of training the PD-VF (Sec. 6.4.1). We use Adam [Kingma and Ba 2015] for optimization. All models use the same network architecture for the policy and value functions. For a given environment, all methods use the same number of steps N_d (at the beginning of each episode) to infer the embedding of the environment dynamics. Then, they each use a single policy network to act in the environment until the end of the episode. We report the cumulative reward obtained by each method throughout an episodes (in which they first infer the environment dynamics which determines the policy used for acting until the end of the episode). We compare with the following baselines:

trains a PPO policy for each environment in our set. This is used as an upper bound for the other models.

is the meta-learning algorithm from Finn et al. [2017]. generally requires some amount of training on the test environments, so to make it more comparable to our method and the other

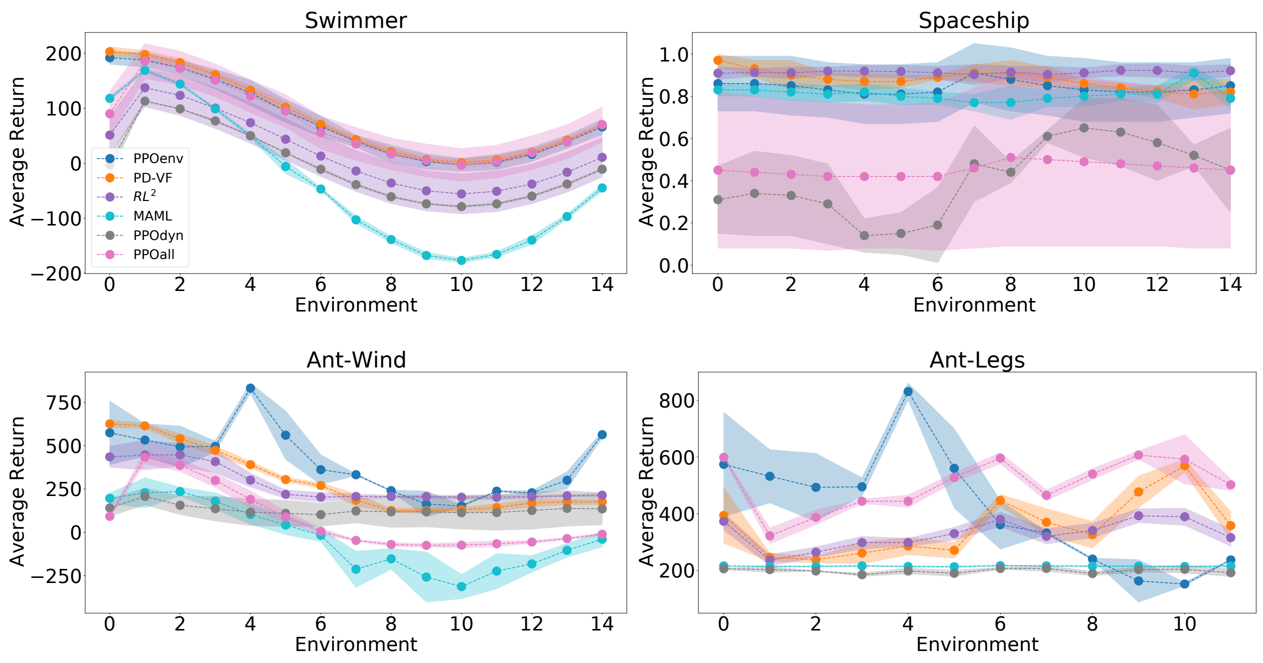


Figure 6.6: Train Performance. Average return on train environments in Swimmer (top-left), Spaceship (top-right), Ant-wind (bottom-left), and Ant-legs (bottom-right) obtained by PD-VF, the upper bound PPOenv, and baselines RL^2, MAML, PPOdyn, and PPOall. PD-VF outperforms the baselines and ablations on most test environments and, in some cases, it is comparable with PPOenv (which was trained directly on the test environments). While other methods also perform reasonably well on the training environments, they generalize poorly to new environments with unseen dynamics.

baselines, we allow one gradient step using a trajectory of length N_d (i.e. the same length as the one used by PD-VF to infer the embedding of the environment dynamics). Thus, has an advantage over PD-VF which does not make any parameter updates at test time.

RL^2 is the meta-learning algorithm from Wang et al. [2016a] and Duan et al. [2016], which uses a recurrent policy that takes as input the previous action and reward.

trains (using PPO) a single policy network conditioned on the dynamics embedding. At test time, it first infers the dynamics embedding and then conditions the pretrained policy network on that vector. This is a close implementation of the approach in Yang et al. [2019][1].

trains a single PPO policy on all the training environments and uses it on the test environments without any additional fine-tuning.

[1]An exact match was not feasible as code for Yang et al. [2019] was not available.

We also compare PD-VF with four ablations:

finds the environment that is closest (in Euclidean metric) to the test environment's embedding and uses the policy trained on that environment to act. This ablation aims to tease out the effect of using both the learned space of policies and that of dynamics to adapt to new environments, from that of only using the learned dynamics space.

clusters the environment embeddings (using trajectories collected in Section 6.4.1) into K clusters. Then, for each cluster, we train a new PPO policy on all the environments assigned to that cluster. At test time, we find the closest cluster for the given environment embedding and use the policy corresponding to that cluster to act in the environment.

trains a PD-VF without using dataset aggregation for the value function (see Section 6.4.3).

uses PD-VF without using dataset aggregation for the policy decoder (see Section 6.4.3).

6.5.4 Adaptation to New Environment Dynamics

As seen in Figures 6.4 and 6.5, PD-VF outperforms all other methods on test environments with new dynamics. In some cases (particularly on Spaceship and Swimmer), our approach is comparable to the PPOenv upper bound which was directly trained on the respective test environment (in contrast, PD-VF has never interacted with that environment before). While the strength of PD-VF lies in quickly adapting to new dynamics, its performance on training environments is still comparable to that of the other baselines, as shown in Figure 6.6. This result is not surprising since current state-of-the-art RL algorithms such as PPO can generally learn good policies for the environments they are trained on, given enough interactions, updates, and the right hyperparamters. However, as predicted, standard model-free RL methods such as the baseline PPOall do not generalize well to environments with dynamics different from the ones experiences during training. Even meta-learning approaches like MAML or RL^2 struggle to adapt when they are allowed to use only a short trajectory for updating the policy at test time, as is the case here.

But most importantly, PD-VF also outperforms the approaches that use the dynamics embedding such as NN, Kmeans, and PPOdyn. This supports our claim that learning a value function for an entire space of policies (rather than for a single optimal policy as standard RL methods do) can be beneficial for adapting to unseen dynamics. By simultaneously estimating the return of a collection of policies in a family of environments with different but related dynamics, PD-VF can learn how variations in dynamics relate to differences in the performance of various policies. This allows the model to rank different policies and understand that sub-optimal behaviors in certain environments might be optimal in others. Thus, at least in theory, PD-VF has the ability to find policies that are better than the ones seen during training. Our empirical results indicate that this might also hold true in practice. Overall, PD-VF proves to be more robust to changes in dynamics relative to the other methods, especially in completely new environments.

6.5.5 Analysis of Learned Embeddings

The performance of PD-VF relies on learning useful policy and dynamics embeddings that capture variations in agent behaviors and transition functions, respectively. In this section, we analyze the learned embeddings.

Figure 6.7 shows a t-SNE plot of the learned policy embeddings for Spaceship, Swimmer, and Ant (from left to right). The top and bottom rows color the embeddings by the policy and environment that generated the corresponding trajectory, respectively. Environments 1 - 15 are used for training, while 16 - 20 are used for evaluation. Trajectories produced by the same policy have similar embeddings, while those generated in the same environment are not necessarily close in this embedding space. This shows that the policy embedding preserves information about the policy while disregarding elements of the environment (that generated the corresponding embedded trajectory).

Similarly, Figure 6.8 shows a t-SNE plot of the learned dynamics embeddings on the three continuous control domains used for evaluating our method. The top row colors each point by the

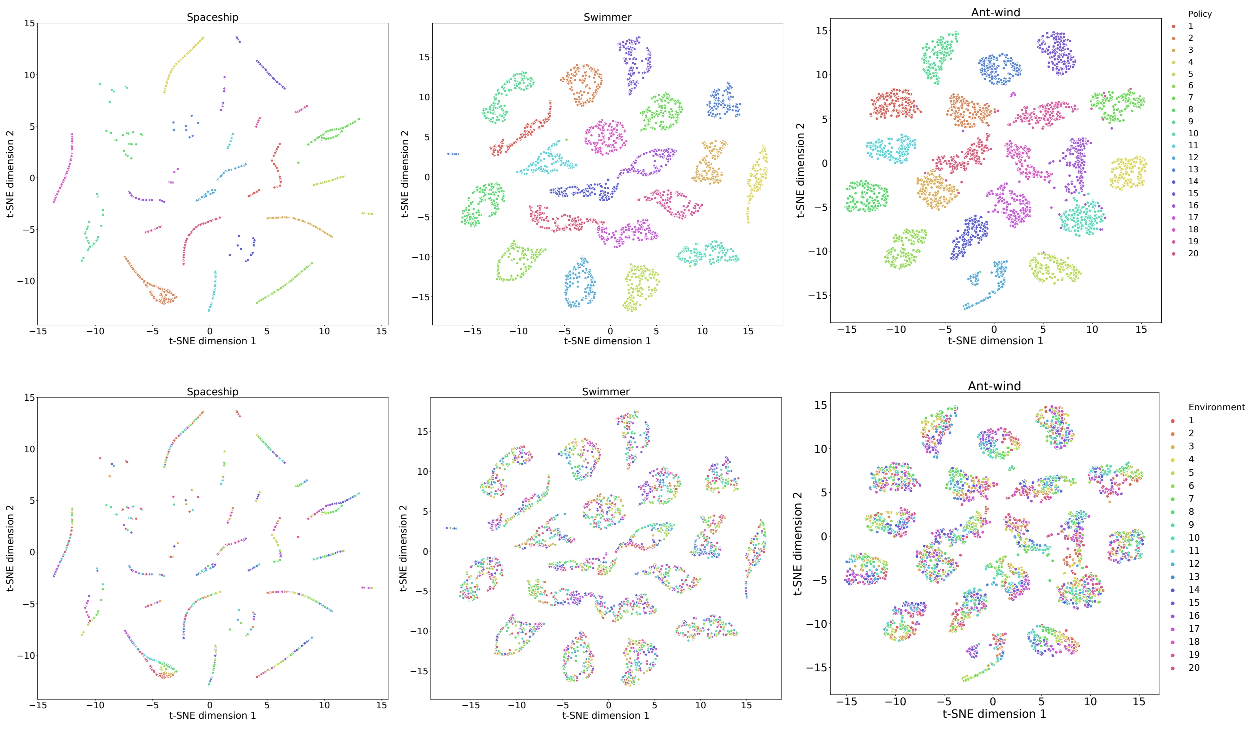


Figure 6.7: t-SNE plots of the learned policy embeddings z_π for Spaceship, Swimmer, and Ant-wind (from left to right). The points are colored by the *policy* (top) and *environment* (bottom) used to generate the trajectory of the corresponding policy embedding.

corresponding environment used to generate the trajectory (from which the embedding is inferred), while the bottom row colors each point by the corresponding policy. The latent space captures the continuous nature (*i.e.* smoothness) of the distribution used to generate the environment dynamics. For example, in Figure **??**, one can see the wind direction corresponding to a particular environment, indicating that the learned embedding space uncovers the one-dimensional (1D) manifold structure of the true dynamics distribution. Even if, during training, the dynamics model never sees trajectories through the test environments, it is still able to embed them within the 1D manifold, thus preserving smoothness in the latent space.

Importantly, this analysis shows that the learned policy and dynamics embeddings are generally disentangled (i.e. information about the dynamics is not contained in the policy space and vice versa). This is notable as we want the dynamics space to mostly capture information about the

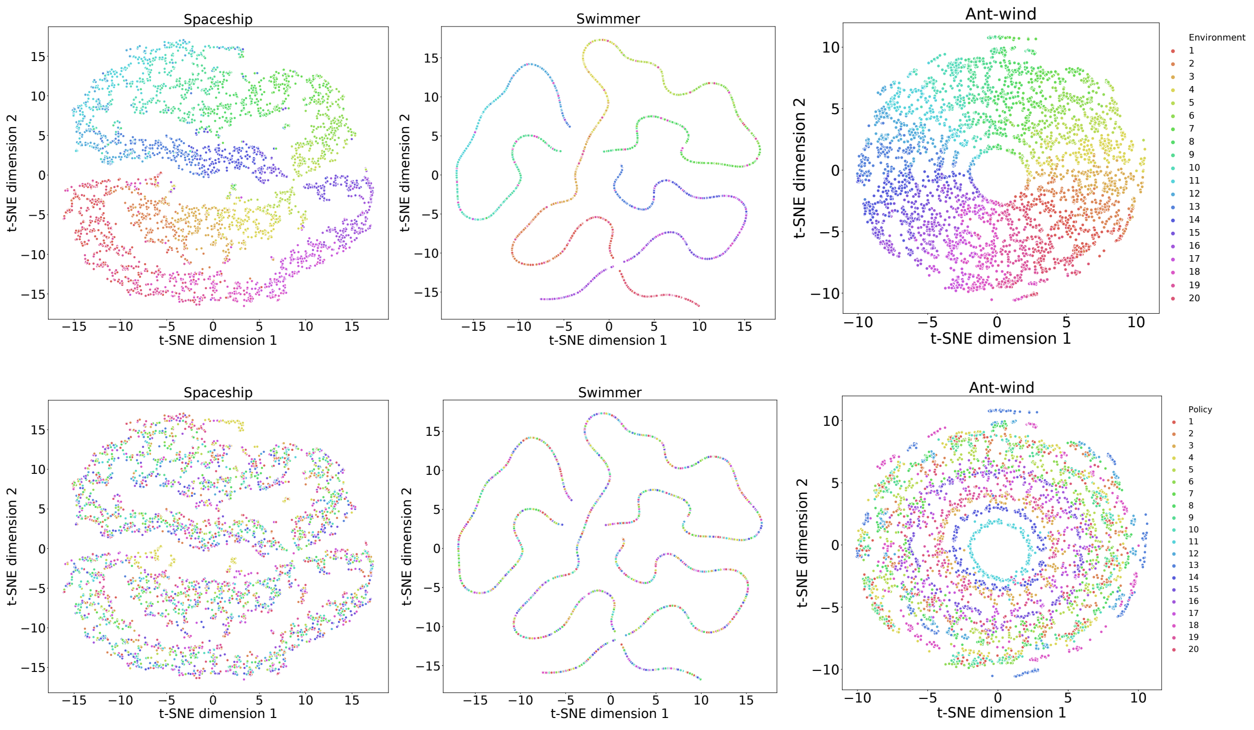


Figure 6.8: t-SNE plots of the learned environment embeddings z_d for Spaceship, Swimmer, and Ant-wind (from left to right). The points are colored by the *environment* (top) and *policy* (bottom) used to generate the trajectory of the corresponding dynamics embedding.

transition function and similarly, we want the policy space to capture variation in the agent behavior. The only exception is the dynamics space of Ant-wind, which contains information about both the environment and the policy. This is because in this environment, the policy is dominated by the force applied to the body of the ant, whose goal is to move forward (while incurring a penalty proportional to the applied force). Thus, depending on the wind direction in the training environment, the agent learns to apply a force of a certain magnitude, a characteristic captured in the embedding space. When evaluated on environments with different dynamics, that policy will still apply a similar force. Our experiments indicate that even if the dynamics space is not fully disentangled (yet it contains information about the environment), the PD-VF is still able to make effective use of the embeddings to find good policies for unseen environments and even outperform other state-of-the-art RL methods.

6.6 Conclusion

In this chapter, we propose policy-dynamics value functions (PD-VF), a novel framework for fast adaptation to new environment dynamics. The key idea is to learn a value function conditioned on both a policy and a dynamics embedding which are learned in a self-supervised way. At test time, the environment embedding can be inferred from only a few interactions, which allows the selection of a policy that maximizes the learned value function. PD-VF has a number of desirable properties: it leverages the structure in both the policy and the dynamics space to estimate the expected return, it only needs a small number of steps to adapt to unseen dynamics, it does not update any parameters at test time, and it does not require dense reward or long rollouts to find an effective policy in a new environment. Empirical results on a set of continuous control domains show that PD-VF outperforms other methods on unseen dynamics, while being competitive on training environments.

PD-VF opens up many promising directions for future research. First of all, the formulation can be extended to estimate the value function not only for a family of policies and environment dynamics, but also for a family of reward functions. Another avenue for future research is to use a more general class of function approximators (such as neural networks) to parameterise the value estimator instead of a quadratic form. The PD-VF framework can, in principle, also be used to evaluate a family of policies and environments on other metrics of interest besides the expected return, such as, for example, reward variance, agent prosociality, deviation from expert behavior, and so on. Another interesting direction is to integrate additional constraints (or prior knowledge) to the optimization problem (*e.g.* maximize expected return while only using policies in a certain region of the policy space). As noted by Precup et al. [2001], Sutton et al. [2011], and White et al. [2012], learning about multiple policies in parallel via general value functions can be useful for lifelong learning. Similarly, PD-VF can be a useful tool for an agent to continually gather knowledge about various policies and dynamics in the world. Finally, PD-VF can also be applied to multi-agent settings for adapting to different opponents or teammates whose behaviors determine the environment dynamics.

Since this work was published, researchers have proposed alternative methods for quickly adapting to changes in the environment's dynamics. For example, Totaro and Jonsson [2021] use Kalman filtering to deal with nonstationarity, Lee and Chung [2021] train RL agents on imaginary tasks generated from mixtures of learned latent dynamics for fast adaptation to new dynamics, while Yang et al. [2020] train robots in an adversarial fashion for improving robustness to joint damage. In a review of continual reinforcement learning, Khetarpal et al. [2020] mention that context detection, which can be framed as learning task and policy embeddings as we propose here, is a promising approach for learning about task relatedness and making progress on continual RL.

7 | Modeling Others using Oneself in Multi-Agent Reinforcement Learning

In the previous chapters, we studied how a single reinforcement learning agent can learn policies which are effective in a wide range of settings and can zero-shot generalize or quickly adapt to new task instances. In this chapter, we consider the multi-agent RL setting with imperfect information in which each agent is trying to maximize its own utility. The reward function depends on the hidden state (or goal) of both agents, so the agents must infer the other players' hidden goals from their observed behavior in order to solve the tasks. Our aim is to learn policies which generalize to opponents or collaborators with different goals. Here, we propose a new approach for learning in these domains: Self Other-Modeling (SOM), in which an agent uses its own policy to predict the other agent's actions and update its belief of their hidden state in an online manner. We evaluate this approach on three different tasks and show that the agents are able to learn better policies using their estimate of the other players' hidden states, in both cooperative and adversarial settings.

7.1 Introduction

Reasoning about other agents' intentions and being able to predict their behavior is important in multi-agent systems, in which the agents might have a diverse, and sometimes competing, set of goals. This remains a challenging problem due to the inherent non-stationarity of such domains.

In this chapter, we introduce a new approach for estimating the other agents' unknown goals from their behavior and using those estimates to choose actions.

We demonstrate that in the proposed tasks, using an explicit model of the other player in the game leads to better performance than simply considering the other agent to be part of the environment.

We frame the problem as a (not-necessarily zero-sum) two-player stochastic game [Shapley 1953], otherwise known as a two-player Markov game, in which the agents have full visibility of the environment, but no explicit knowledge about other agents' goals and there is no communication channel. The reward received by each agent at the end of an episode depends on the goals of both agents, so the optimal policy of each agent must take into account both of their goals.

Research in cognitive science suggests that humans maintain models of other people they interact with, which capture their goals, beliefs, or preferences gopnik1992child, premack1978does. In some cases, humans use their own mental process to simulate others' behavior by adopting their perspective [Gordon 1986; Gallese and Goldman 1998]. This allows them to understand others' intentions or motives and act accordingly in social settings. Inspired by these studies, the key idea of our approach is that as a first approximation, to understand what the other player in the game is doing, an agent should ask itself "what would be my goal if I had acted as the other player had?". We instantiate this idea by parametrizing the agent's action and value functions with a (multi-layer recurrent) neural network that takes the state and a goal as an input. As the agent plays the game, it infers the other agent's unknown goal by directly optimizing over the goal (using its own action function) to maximize the likelihood of the other's actions.

7.2 Related Work

Opponent modeling has been extensively studied in games of imperfect information. However, most previous approaches focus on developing models with domain-specific probabilistic priors or strategy parametrizations. In contrast, our work proposes a more general framework for opponent modeling. Davidson [1999] uses an MLP to predict opponent actions given a game history, but the agents cannot adapt to their opponents' behavior in an online manner. Lockett et al. [2007] designs a neural network architecture to identify the opponent type by learning a mixture of weights over a given set of cardinal opponents. However, the game does not unfold within the reinforcement learning framework.

A large body of work in deep multi-agent RL focuses on partially visible, fully cooperative settings [Foerster et al. 2016a,b; Omidshafiei et al. 2017] and emergent communication [Lazaridou et al. 2016; Foerster et al. 2016a; Sukhbaatar et al. 2016; Das et al. 2017; Mordatch and Abbeel 2017] Our setting is different since we do not allow any communication among the agents, so the players have to indirectly reason about their opponents' intentions from their observed behavior. In contrast, Leibo et al. [2017] considers semi-cooperative multi-agent environments in which the agents develop cooperative and competitive strategies depending on the task type and reward structure. Similarly, Lowe et al. [2017] proposes a centralized actor-critic architecture for efficient training in settings with such mixed strategies. Lerer and Peysakhovich [2017] design RL agents that are able to maintain cooperation in complex social dilemmas by generalizing a well-known game theoretic strategy called tit-for-tat [Axelrod 2006] to multi-agent Markov games. Recent work in cognitive science attempts to understand human decision-making by using a hierarchical model of social agency that infers the intentions of other human agents in order to decide whether to play a cooperative or competitive strategy [Kleiman-Weiner et al. 2016]. However, none of these papers design algorithms that explicitly model other artificial agents in the environment or estimate their intentions, with the purpose of improve their decision making.

The field of inverse reinforcement learning (IRL) [Russell 1998; Ng et al. 2000; Abbeel and Ng 2004], is also related to the problem considered here. IRL's aim is to infer the reward function of an agent by observing its behavior, which is assumed to be nearly optimal. In contrast, our approach uses the observed actions of the other player to directly infer its goal in an online manner, which is then used by the agent when acting in the environment. This avoids the need for collecting offline samples of the other's (state, action) pairs in order to estimate its reward function and then use this to learn a separate policy that maximizes that utility. The more recent papers by Hadfield-Menell et al. [2016, 2017] are also concerned with the problem of inferring others' intentions, but their focus is on human-robot interaction and value alignment. Motivated by similar goals, chandrasekaran2017takes consider the problem of building a theory of AI's mind, in order to improve human-AI interaction and the interpretability of AI systems. For this purpose, they show that people can be trained to predict the responses of a Visual Question Answering model, using a small number of examples.

The closest work to ours is Foerster et al. [2017] and He et al. [2016]. Foerster et al. [2017] designs RL agents that take into account the learning of other agents in the environment when updating their own policies. This enables the agents to discover self-interested yet collaborative strategies such as tit-for-that in the iterated prisoner's dilemma. While our work does not explicitly attempt to shape the learning of other agents, it has the advantage that the agents can update their beliefs during an episode and change their strategies in an online manner to gain more reward. Our setting is also different in that it considers that each agent has some hidden information needed by their the other player in order to maximize its return.

Our work is very much in line with He et al. [2016], where the authors build a general framework for modeling other agents in the reinforcement learning setting. He et al. [2016] proposes a model that jointly learns a policy and the behavior of opponents by encoding observations of the opponent into a DQN. Their Mixture of Experts architecture is able to discover different opponent strategy patterns in two purely adversarial tasks. One difference between our work and He et al. [2016]'s is that we do not aim to infer other agents' strategies, but rather focus on explicitly estimating their goals in the environment. Moreover, rather than using a hand designed featurization of the other agent's actions, in this work, the agent learns its model of the other end-to-end, based on its own model. Another difference is that in this work, the agent runs an optimization to infer the other agent's hidden state, instead of inferring the other agent's hidden state via a feed-forward network. In the experiments below, we show that SOM outperforms an adaptation of the method of He et al. [2016] to our setting.

7.3 Background

A Markov game for two agents is defined by a set of states $\mathcal{S}$ describing the possible configurations of all agents, a set of actions $\mathcal{A}_1, \mathcal{A}_2$ and a set of observations $\mathcal{O}_1, \mathcal{O}_2$ for each agents, and a transition function $P : \mathcal{S} \times \mathcal{A}_1 \times \mathcal{A}_2 \to \mathcal{S}$ which gives the probability distribution on the next state as a function of current state and actions. Each agent i chooses actions by sampling from a stochastic policy $\pi_{\theta_i} : \mathcal{S} \times \mathcal{A}_i \to [0, 1]$. Each agent has a reward function which depends on agent's state and action: $r_i : \mathcal{S} \times \mathcal{A}_i \to \mathbb{R}$. Each agent i tries to maximize its own total expected return $R_i = \sum_{t=0}^{T} \gamma^t r_i^t$, where γ is a discount factor and T is the time horizon. In this work, we consider both cooperative, as well as adversarial settings.

7.4 Approach

We now describe Self Other-Modeling (SOM), a new approach for inferring the other agents' goals in an online fashion during an episode and using these estimates to choose actions. To decide an action and to estimate the value of a state, we use a neural network f that takes as input its own goal z_{self}, an estimate of the other player's goal $\tilde{z}_{other}$, and the observation state from its own perspective s_{self}, and outputs a probability distribution over actions π and a value estimate

V, such that for each agent i playing the game we have:

$$\begin{bmatrix} \pi^i \\ V^i \end{bmatrix} = f^i(s^i_{self}, z^i_{self}, \tilde{z}^i_{other};\ \theta^i)\ .$$

Here θ^i are agent i's parameters for f, which has one softmax output for the policy, one linear output for the value function, and all the non-output layers shared. The actions are sampled from the policy π. The observation state s^i_{self} explicitly contains the location of the acting agent (the one whose action is decided by f^i), as well as the location of the other agent.

Because an agent computes both its own actions and values, as well as estimates of the other agent's, each agent has two networks (omitting the agent index i for brevity):

$$f_{self}(s_{self}, z_{self}, \tilde{z}_{other};\ \theta_{self}) \tag{7.1}$$

and

$$f_{other}(s_{other}, \tilde{z}_{other}, z_{self};\ \theta_{self})\ . \tag{7.2}$$

The two networks are used in different ways: f_{self} is used for computing the agent's own actions and values, and operates in a feed-forward manner. The agent uses f_{other} to infer the other agent's goal via an optimization over $\tilde{z}_{other}$ given the other agent's observed actions.

We propose that each agent models the behavior of the other player using its own policy, so that the parameters of f_{other} are the same as the parameters of f_{self}. However, note that the two networks differ in their relative placement of the inputs z_{self} and $\tilde{z}_{other}$. Additionally, since the environment is fully observed, the observation state of the two agents differs only by the specification of the agent's identity on the map (i.e. each agent will be able to distinguish between its own location and the other's location). Hence, in acting mode, the network f_{self} will take as input s_{self} and in inference mode, the network f_{other} will take as input s_{other}.

At each step of the game, the agent needs to infer $\tilde{z}_{other}$ in order to input its estimate into (7.1)

Algorithm 5 SOM training for one episode

```
1:  procedure Self Other-Modeling
2:      for k := 1, num_players do
3:          $\tilde{z}^{k}_{other} \leftarrow \frac{1}{ngoals}\mathbf{1}_{ngoals}$
4:      end for
5:      game.reset()
6:      for step := 1, episode_length do
7:          $i \leftarrow game.get\_acting\_agent()$
8:          $j \leftarrow game.get\_non\_acting\_agent()$
9:          $s^{i}_{self} \leftarrow game.get\_state()$
10:         $s^{j}_{other} \leftarrow game.get\_state()$
11:         $\tilde{z}^{OH,i}_{other} = one\_hot[argmax(\tilde{z}^{i}_{other})]$
12:         $\pi^{i}_{self}, V^{i}_{self} \leftarrow f^{i}_{self}(s^{i}_{self}, z^{i}_{self}, \tilde{z}^{OH,i}_{other}; \theta^{i}_{self})$
13:         $a^{i}_{self} \sim \pi^{i}_{self}$
14:         $game.action(a^{i}_{self})$
15:         for k : = 1, num_inference_steps do
16:             $\tilde{z}^{G,j}_{other} = gumbel\_softmax(\tilde{z}^{j}_{other})$
17:             $\tilde{\pi}^{j}_{other} \leftarrow f^{j}_{other}(s^{j}_{other}, \tilde{z}^{G,j}_{other}, z^{j}_{self}; \theta^{j}_{self})$
18:             $loss = cross\_entropy\_loss(\tilde{\pi}^{j}_{other},\ a^{i}_{self})$
19:             $loss.backward()$
20:             $update(\tilde{z}^{j}_{other})$
21:         end for
22:     end for
23:     for k := 1, num_players do
24:         $policy.update(\theta^{k}_{self})$
25:     end for
26: end procedure
```

and choose its action. For this purpose, at each step, the agent observes the other taking an action and, at the next step, the agent uses the previously observed action of the other as supervision, in order to back-propagate through (7.2) and optimize over $\tilde{z}_{other}$. Figure 7.1 illustrates this technique.

The number of steps taken by the optimizer in this inference procedure is a hyperparameter that can be varied depending on the game. Hence, the estimate of the other agent's goal $\tilde{z}_{other}$ is updated multiple times at each step during the game. The parameters θ_{self} are updated at the end of each episode using Asynchronous Advantage Actor-Critic (A3C) [Mnih et al. 2016b] with reward signal obtained by the self agent.

Algorithm 5 represents the pseudo-code for training SOM agents for one episode. Since the goals are discrete in all the tasks considered here, the agent's goal $\tilde{z}_{self}$ is encoded as a one-hot vector of dimension equal to the total number of possible goals in the game. The embedding of the other player's goal $\tilde{z}_{other}$ has the same dimension. In order to estimate the gradients going through $\tilde{z}_{other}$, which is a discrete variable and thus non-differentiable, we replace it with a differentiable sample from the Gumbel-Softmax distribution [Jang et al. 2016; Maddison et al. 2016], $\tilde{z}^{G}_{other}$. This reparametrization trick was shown to efficiently produce low-variance biased gradients. After optimizing $\tilde{z}_{other}$ at each step using this method, $\tilde{z}_{other}$ usually deviates from a one-hot vector. At the next step, f_{self} takes as input the one-hot vector $\tilde{z}^{OH}_{other}$ corresponding to the *argmax* of the previously updated $\tilde{z}_{other}$.

The agents' policies are parametrized by long short-term memory (LSTM) cells [Hochreiter and Schmidhuber 1997] with two fully-connected linear layers, and exponential linear unit (ELU) [Clevert et al. 2015] activations. The weights of the networks are initialized with semi-orthogonal matrices, as described in [Saxe et al. 2013] and zero bias.

Due to the recurrence of f_{other}, special care must be taken when the number of inference steps is > 1. Under this setting, at each step in the game, we save the recurrent state of f_{other} before the first forward pass in inference mode, and initialize the recurrent state to this value for every inference step. This procedure ensures f_{other} is unrolled the same number of steps during both

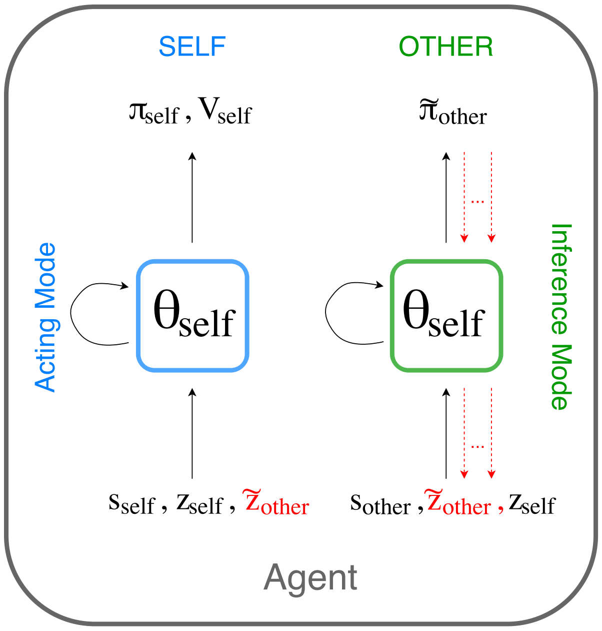


Figure 7.1: Our Self Other-Model (SOM) architecture for a given agent.

acting and inference mode.

7.5 Experiments

In this section, we evaluate our model SOM on three tasks:

- The coin game, in Section 7.5.2, which is a fully co-operative task where the agents' roles are symmetric.
- The recipe game, in Section 7.5.3, which is adversarial, but with symmetric roles.
- The door game, in Section 7.5.4, which is fully cooperative but has asymmetric roles for the two players.

We compare SOM to three other baselines and to a model that has access to the ground truth of the other agent's goal. All the tasks considered are created in the Mazebase grid-world environment [Sukhbaatar et al. 2015].

7.5.1 Baselines

True-Other-Goal (TOG): We provide an upper bound on the performance of our model given by a policy network which takes the other agent's **true** goal as input, z_{other}, as well as the state features s_{self} and its own goal z_{self}. Since this model has direct access to the true goal of the other agent, it does not need a separate network to model the behavior of the other agent. The architecture of TOG is the same as the one of SOM's policy network, f_{self}.

No-Other-Model (NOM): The first baseline we use only takes as inputs the observation state s_{self} and its own goal z_{self}. NOM has the same architecture as the one used for SOM's policy network, f_{self}. This baseline has no explicit model of the other agent or estimate of its goal.

Integrated-Policy-Predictor (IPP): Starting with the architecture and inputs of NOM, we construct a stronger baseline, IPP, which has an additional final linear layer that outputs a probability distribution over the next action of the other agent. Besides the A3C loss used to train the policy of this network, we also add a cross-entropy loss to train the prediction of the other agent's action, using observations of its behavior.

Separate-Policy-Predictor (SPP): He et al. [2016] propose an opponent modeling framework based on DQN. In their approach, a neural network (separate from the learned Q-network) is trained to predict the opponent's actions, given hand crafted state information specific to the opponent. An intermediate hidden representation from this network is given as input to the the Q-network.

We adapt the model of He et al. [2016] to our setting. In particular, we use A3C instead of DQN and we do not use the task-specific features used to represent the hidden state of the opponent.

The resulting model, SPP, consists of two *separate* networks, a policy network for deciding the agent's actions, and an opponent network for predicting the other's actions. The opponent network takes as input the state of the world s and its own goal z_{self}, and outputs a probability distribution for the action taken by the other agent at the next step, as well as its hidden state (given by the network's recurrence). As in IPP, we train the opponent policy predictor with a cross-entropy loss using the true actions of the other agent. At each step, the hidden state output by this network is taken as input by the agent's policy network, along with the observation state and its own goal. Both the policy network and the opponent policy predictor are LSTMs with the same architecture as SOM.

In contrast to SOM, SPP does not explicitly infer the other agent's goal. Rather, it builds an implicit model of the opponent by predicting the agent's actions at each time step. In SOM, an inferred goal is given as additional input to the policy network. The analog of the inferred goal in SPP is the hidden representation obtained from the opponent policy predictor which is given as an additional input to the policy network.

Training Details. In all our experiments, we train the agents' policies using A3C [Mnih et al. 2016b] with an entropy coefficient of 0.01, a value loss coefficient of 0.5, and a discount factor of 0.99. The parameters of the agents' policies are optimized using Adam [Kingma and Ba 2015] with $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 1 \times 10^{-8}$, and weight decay 0. SGD with a learning rate of 0.1 was used for inferring the other agent's goal, $\tilde{z}_{other}$.

The hidden layer dimension of the policy network was 64 for the Coin and Recipe Games and 128 for the Door Game. We use a learning rate of 1×10^{-4} for all the games and models.

The observation state s is represented by few-hot vectors indicating the locations of all the objects in the environment, as well as the locations of the self and the other. The dimension of this input state is $1 \times nfeatures$, where the number of features is 384, 192, and 900 for the Coin, Recipe, and Door games, respectively.

For each experiment, we trained the models using 5 different random seeds. All the results shown are for 10 optimization updates of $\tilde{z}_{other}$ at each step in the game, unless mentioned otherwise.

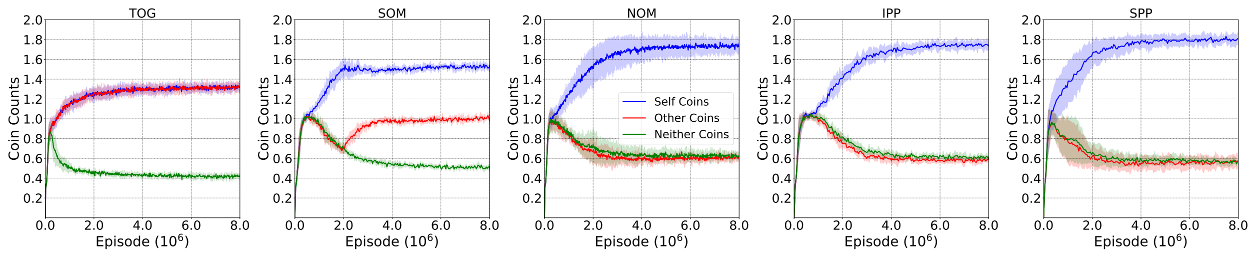


Figure 7.2: Coin Strategy: Average number of collected coins per episode corresponding to the color of the Self (blue), Other (red), or Neither (green) by the agents using TOG (left), SOM (center-left), NOM (center), IPP (center-right), and SPP (right). The optimal strategy is to pick up as many Self as Other coins on average, across a number of episodes, and no Neither coins. Being able to collect more Other than Neither coins indicates that the agent is able to accurately infer the other agent's color early enough during some of the episodes and uses this information to collect more Other, instead of Neither coins, which increases its reward. The TOG model learns to collect just as many Self as Other coins, while all the baselines only learn to collect more Self coins, but cannot distinguish between the Other and Neither coins. SOM learns to collect significantly more Other coins than Neither. This shows that SOM converges to a closer-to-optimal strategy using its guess of the other's goal.

7.5.2 Coin Game.

First, we evaluate the model on a fully cooperative task, in which the agents can gain more reward when using both of their goals rather than only their own goal. So it is in the best interest of each agent to estimate the other player's goal and use that information when taking actions. The game, shown in the left diagram of Figure 7.4, takes place on a 8×8 grid containing 12 coins of 3 different colors (4 coins of each color). At the beginning of each episode, the agents are randomly assigned one of the three colors. The action space consists of: go up, down, left, right, or pass. Once an agent steps on a coin, that coin disappears from the grid. The game ends after 20 steps (i.e. each agent takes 10 steps). The reward received by both agents at the end of the game

$$R = (n^{self}_{C_{self}} + n^{other}_{C_{self}})^2 + (n^{self}_{C_{other}} + n^{other}_{C_{other}})^2 - (n^{self}_{C_{neither}} + n^{other}_{C_{neither}})^2,$$

where $n^{other}_{C_{self}}$ is the number of coins of the self's goal-color, which were collected by the other agents, and $n^{self}_{C_{neither}}$ is the number of coins corresponding to neither of the agents' goals, collected by the self. For the example in Figure 7.4, agent 1 has C_{self} = orange and C_{other} = cyan, while agent 2's C_{self} is cyan and C_{other} is orange. $C_{neither}$ is red for both agents.

The role of the penalty for collecting coins that do not correspond to any of the agents' goals is to avoid convergence to a brute force policy in which the agents can gain a non-negligible amount of reward by collecting all the coins in their proximity, without any regard to their color.

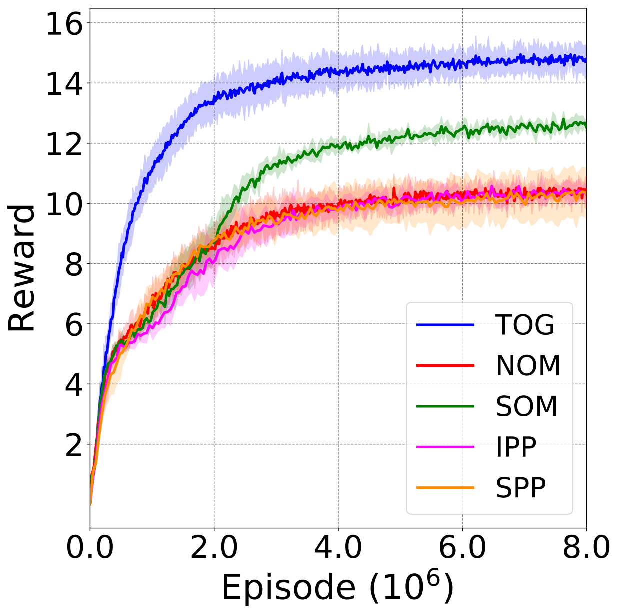


Figure 7.3: Coin Performance: Average reward obtained on the Coin game by SOM (green), TOG (blue), NOM (red), IPP (magenta), and SPP (orange). SOM performs better than all the baselines.

To maximize its return, each agent needs to collect coins of its own or its collaborator's color, but not those of the remaining color. Thus, when both agents are able to infer their collaborators' goals with high accuracy and as early as possible in the game, they can use that information to maximize their shared utility.

Figure 7.3 shows the mean and standard deviation of the reward across 5 runs with different random seeds obtained by SOM. Our model clearly outperforms all the baselines on this task. We also show the empirical upper bound on the reward using the model which takes as input the true color assigned to the other agent.

Figure 7.2 analyzes the strategies of the different models by looking at the proportion of coins of each type collected by the agents. The optimal strategy is for each agent to maximize $n^{self}_{C_{self}} + n^{self}_{C_{other}}$ and $n^{self}_{C_{neither}} = 0$. Due to the randomness in the initial conditions (in particular, the locations of coins in the environment), this amounts to each agent collecting an equal number of coins of its own color and coins of the other's color on average, across a large number of episodes (i.e. $\bar{n}^{self}_{C_{self}} = \bar{n}^{self}_{C_{other}}$).

Indeed, this is the strategy learned by the model with perfect information of the other agent's goal (TOG). SOM also learns to collect significantly more Other than Neither coins (although not as many as Self coins), indicating its ability to distinguish between the two types, at least during some of the episodes. This means that SOM can accurately infer the other agent's goal early enough during the episode and use that information to collect more Other Coins, thus gaining more reward than if it were only using its own goal to direct its actions.

In contrast, the agents trained with the three baseline models collect significantly more Self coins, and as many Other as Neither coins on average. This shows that they learn to use their own goal for gaining reward, but they are unable to use the hidden goal of the other agent for further increasing their reward. Even if IPP and SPP are able to predict the actions of the other player with an accuracy of about 50%, they do not learn to distinguish between the coins that would increase (Other) and those that would decrease (Neither) their reward. This shows the weaknesses of using an implicit model of the other agent to maximize reward on certain tasks.

7.5.3 Recipe Game.

Agents in adversarial scenarios can also benefit from having a model of their opponents, which would enable them to exploit the weaknesses of certain players. With this motivation in mind, we evaluate our model on a game in which the agents have to craft certain compositional recipes,

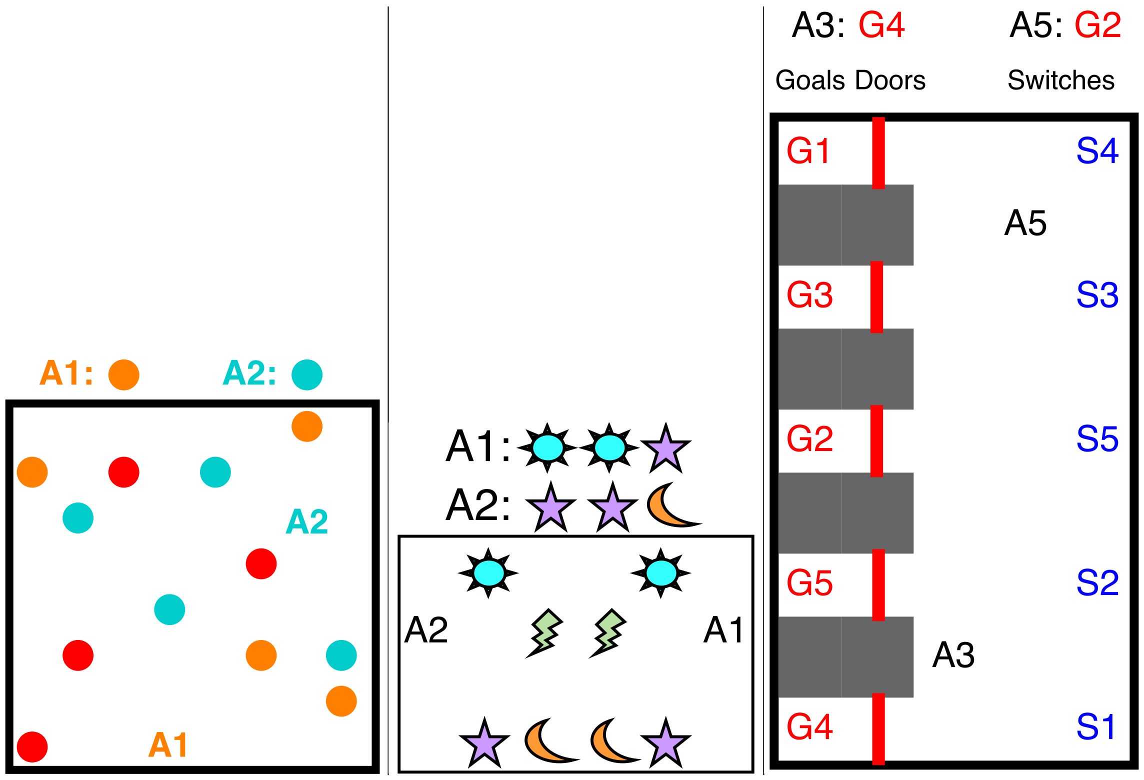


Figure 7.4: Illustration of the Coin (left), Recipe (center), and Door (right) games Above each ones we show the agents' goals (not visible to one another).

each containing multiple items found in the environment. The agents are given as input the names of their goal-recipes, without the corresponding components needed to make it. The resources in the environment are scarce, so only one of the agents can craft its recipe within one episode.

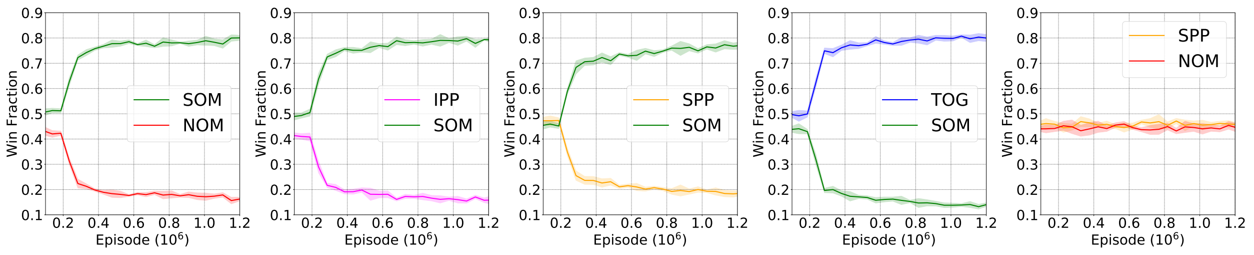


Figure 7.5: Recipe Performance: Average fraction of success in the Recipe game by SOM-NOM (left), SOM-IPP (center-left), SOM-SPP (center-center), SOM-TOG (center-right), NOM-AcrPredSep (right). The plots show the performance of SOM with 5 optimization updates of $\tilde{z}_{other}$ at each step in the game.

As illustrated in Figure 7.4 (center), there are 4 types of items: {sun, star, moon, lightning} and 4 recipes: {sun, sun, star}; {star, star, moon}; {moon, moon, lightning}; {lightning, lightning, sun}. The game is played in a 4×6 grid, which contains 8 items in total, 2 of each type.

At the beginning of each episode, we randomly assign a recipe to one of the agents, and then we randomly pick a recipe for the other agent so that it has overlapping items with the recipe of the first agent. This ensures that the agents are competing for resources within each episode. At the end of the episode, each agent receives a reward of +1 for crafting its own recipe and a penalty of -0.1 for each item it picked up not needed for making its recipe.

We designed the layout of the grid so that neither agent has an initial advantage by being closer to the scarce resource. At the beginning of each episode, one of the agents starts on the left-most column of the grid, while the other one starts on the right-most column, at the same y-coordinate. Their initial y-coordinate as well as which agent starts on the left/right is randomized. Similarly, one item of each of the 4 different types is placed at random in the grid formed by the second and third columns of the maze, from left to right. The rest of the items are placed in the forth and fifth columns, so that the symmetry with respect to the vertical axis is preserved (i.e. items of the same type are placed at the same y-coordinate, and symmetric x-coordinates).

Agents have six actions to choose from: pass, go up, down, left, right, or pick (for picking an item, which then disappears from the grid). The first agent to take an action is randomized. The game ends after 50 steps.

We pretrain all the baselines on a version of the game which does not have overlapping recipes, in order to ensure that all the models learn to pick up the corresponding items, given a recipe as goal. All of the models learn to craft their assigned recipes $\sim 90\%$ of the time on this simpler task. Then, we continue training the models on the adversarial task in which their recipes overlap in each episode. SOM is initialized with a pretrained NOM network.

Figure 7.5 shows the winning fraction for different pairs played against each other in the Recipe game. For the first 100k episodes, the models are not being trained. We can see that SOM significantly outperfroms NOM, IPP, and SPP, winning $\sim 75-80\%$ of the time, while the baselines can only win $\sim 15-20\%$ of the games. SPP ties against NOM, and TOG outperforms SOM by a large margin. We also played the same types of agents against each other and they all win $\sim 40-50\%$ of the games.

7.5.4 Door Game.

In this section, we show that on a collaborative task with asymmetric roles and multiple possible partners, the agents can learn to figure out what role they should be playing in each game based on their partners actions.

In the Door game, two agents are located in a 5×9 grid, with 5 goals behind 5 doors on the left wall, and 5 switches on the right wall of the grid. The game starts with the two players in random squares on the grid, except for the ones occupied by the goals, doors, or switches, we illustrated in Figure 7.4. Agents can take any of the five actions: go up, down, left, right or pass. An action is invalid if it moves the player outside of the border or to a square occupied by a block or closed door. Both agents receive +3 reward when either one of them steps on its goal and they are penalized -0.1 for each step they take. The game ends when one of them gets to its goal or

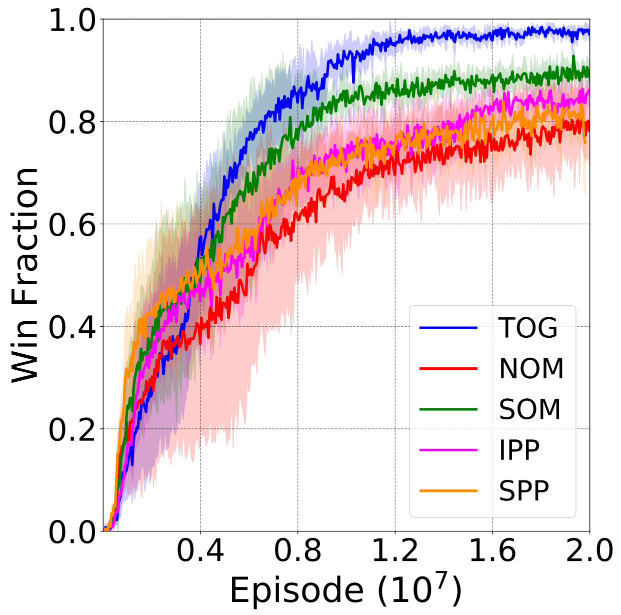


Figure 7.6: Door Performance: Average fraction of success on the Door game by SOM (green), TOG (blue), NOM (red), IPP (magenta), and SPP (orange). On average, SOM performs better than all the baselines.

after 22 steps. All the goals are behind doors which are open only as long as one of the agents sits on the corresponding switch for that door.

At the beginning of an episode, each of the two players is randomly selected from a pool of 5 agents and receives as input a random number from 1 to 5 corresponding to its goal. Each of the 5 agents has its own policy which gets updated at the end of each episode they play. Note that the agents' identities are not visible (i.e. there is no indication in the state features that specifies the id's of the agents playing during a given episode). This restriction is important in order to ensure that the agents cannot gain advantage by specializing into the two roles needed to win (i.e. goal-goer and switch-puller) and identifying the specialization of the other player by simply observing its unique id.

The agents need to cooperate in order to receive reward. In contrast to our previous tasks, the two players must take different roles. In fact, the player who sits on the switch should ignore its own goal and instead infer the other's goal, while the player who goes to its goal does not need to infer the other's goal, but only use its own. In order to sit on the correct switch, an agent has to infer the other player's goal from their observed actions. The only way in which an agent can use its own policy to model the other player is if each agent learns to play both roles of the game, i.e. go to its own goal and also open its collaborator's door by sitting on the corresponding switch. Indeed, we see that the agents learn to play both roles and they are able to use their own policies to infer the other player's goals when needed.

Fig 7.6 shows the mean and standard deviation of the winning fraction obtained by one of the agents on the Door game. While our model is still able to outperform the three baselines, the gap between the performance of our model and that of IPP or SPP (an approximate version of [He et al. 2016]) is smaller than in the previous tasks. However, this is a more difficult task for our model since it needs the agents to learn performing both roles before effectively use its own policy to infer the other agent's goal. Nevertheless, we see that SOM training allows the agents to play both roles in an asymmetric cooperative game, and to infer the goal and role of the other player.

7.5.5 Analyzing the goal inference

In this section we further analyze the ability of the SOM models to infer other's intended goals.

Figure 7.7 shows the fraction of episodes in which the goal of the other agent is correctly inferred. We consider that the goal is correctly inferred only when the estimate of the other's goal remains accurate until the end of the game, so that we avoid counting the episodes in which the agent might infer the correct goal by chance at some intermediate step in the game. In all the games, the SOM agent learns to infer the other player's goal with a mean accuracy ranging from $\sim 60 - 80\%$. Comparing the second plot in Figure 7.2 with the left plot in Figure 7.7, one can observe that the SOM agent starts distinguishing Other from Neither coins after approximately 2M training epochs, which coincides with the time when the mean accuracy of the inferred goal converges to $\sim 75\%$. The Door Game (right) presents higher variance since the agents learn to use

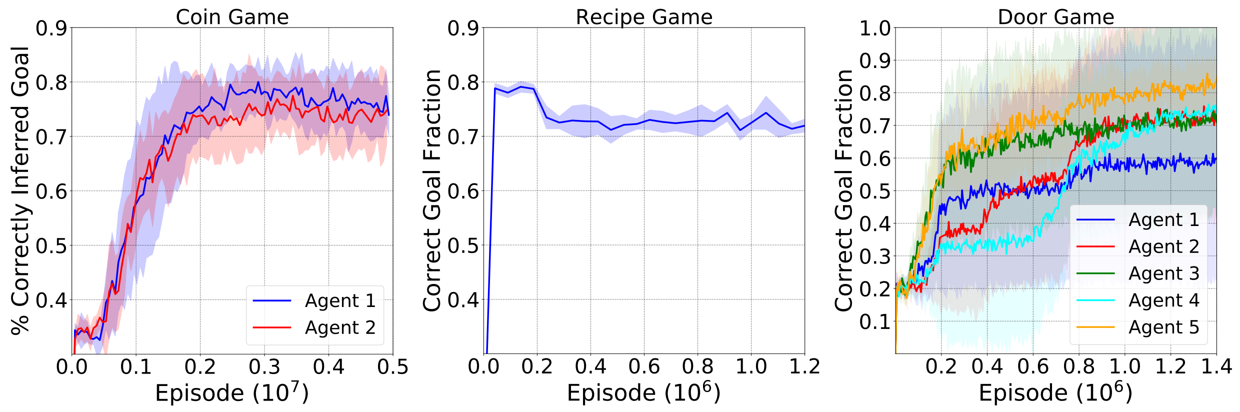


Figure 7.7: Inference Accuracy during Training: The mean fraction of episodes in which the agent correctly infers the other's goal for the Coin (left), Recipe (center), and Door (right) games, as a function of training epoch. The estimate of the other's goal is considered correct if it remains accurate during all the following steps in the game.

and infer the other's goal at different stages during training.

Figure 7.8 shows the cumulative distribution of the step at which the goal of the other player is correctly inferred (and remains the same until the end of the game). The cumulative distribution is computed over the episodes in which the goal is correctly inferred before the end of the game. In the Coin (blue) and Recipe (red) games, 80% of the times the agent correctly infers the goal of the other, it does so in the first five steps. The distribution for the Door (green) game indicates that the agent needs more steps on average to correctly infer the goal. This explains in part why the SOM agent only slightly outperforms the SPP baseline. If the agent does not infer the other's goal early enough in the episode, it cannot efficiently use it to maximize its reward.

Figure 7.9 shows how the performance of the agent varies with the number of optimization updates performed on $\tilde{z}_{other}$ at each step in the game. As expected, the agent's reward (blue) generally increases with the number of inference steps, as does the fraction of episodes in which the goal is correctly inferred. One should note that increasing the number of inference steps from 10 to 20 only translates into less than 0.45% performance gain, while increasing it from 1 to 5 translates into a performance gain of 6.9% on the Coin game, suggesting that there is a certain

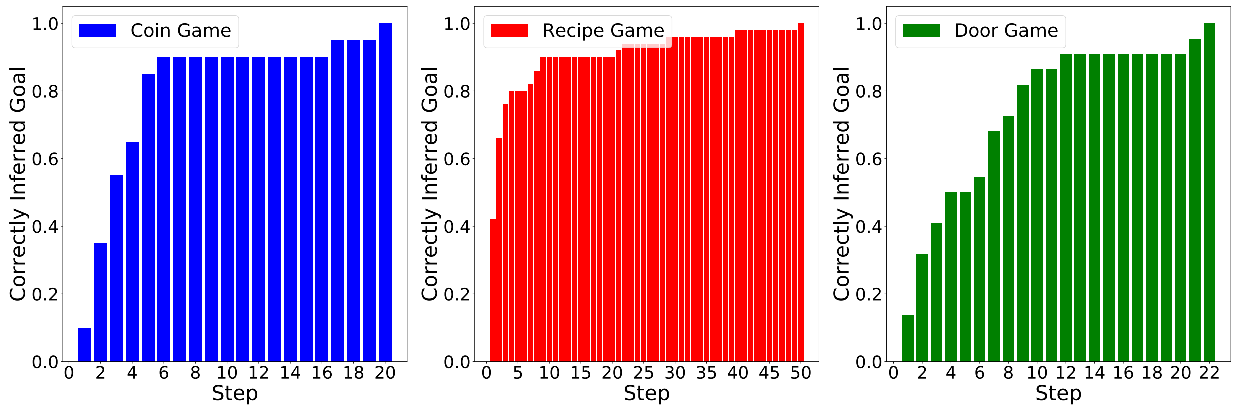


Figure 7.8: Inference Step Distribution: Cumulative distribution of the step t_{inf} at which the goal of the other player is correctly inferred (i.e. $\tilde{z}^t_{other} = z_{other}, \forall t \geq t_{inf}$) for the Coin (left), Recipe (center) and Door (right) games. We define this step so that $\tilde{z}_{other} = z_{other}$ for all the remaining steps in the game. The distribution is computed over the subset of runs in which the goal is correctly inferred before the end of the game ($\sim 70 - 80\%$ of all runs). A total of 1000 runs with trained SOM models were used to compute this distribution.

threshold above which increasing the number of inference steps will not significantly improve performance.

7.6 Conclusion

In this chapter, we introduced a new approach for inferring other agents' hidden states from their behavior and using those estimates to choose actions. We demonstrated that the agents are able to estimate the other players' hidden goals in both cooperative and competitive settings, which enables them to converge to better policies and gain higher rewards. In the proposed tasks, using an explicit model of the other player led to better performance than simply considering the other agent to be part of the environment. One limitation of SOM is that it requires a longer training time than the other baselines, since we backpropagate through the network at each step. However, their online nature is essential in adapting to the behavior of other agents in the environment.

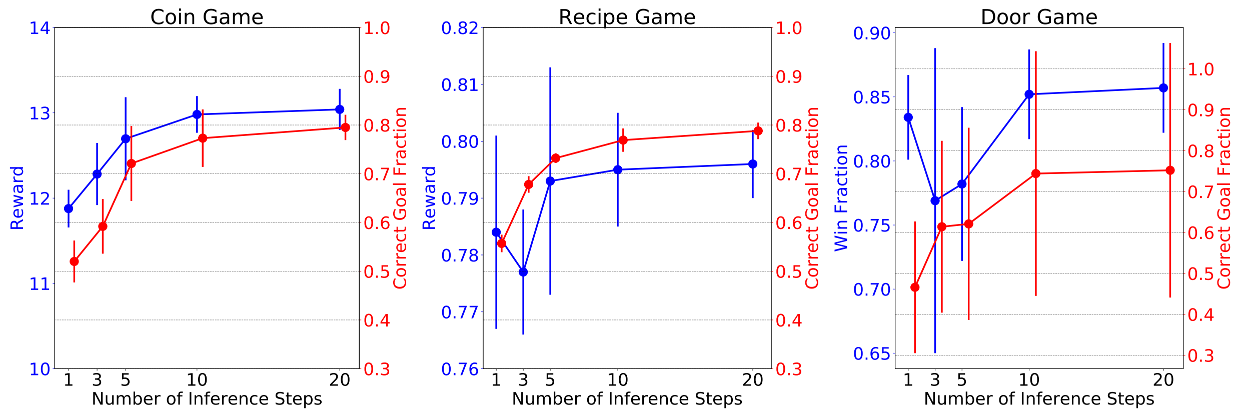


Figure 7.9: Performance Variation with Number of Inference Steps: Average reward (blue) and average fraction of episodes in which the goal of the other agent is correctly inferred (red) obtained by the SOM agent as a function of the number of inference steps used for estimating the other's goal for the Coin (left), Recipe (center), and Door (right) games. The error bars represent 1 standard deviation.

Some of the main advantages of our method are its simplicity and flexibility. This method does not require any extra parameters to model the other agents in the environment, can be trained with any reinforcement learning algorithm, and can be easily integrated with any policy parametrization or network architecture. The SOM concept can be adapted to settings with more than two players, since the agent can use its own policy to model the behavior of any number of agents and infer their goals. Moreover, it can be easily generalized to many different environments and tasks.

Some interesting directions for future work are to evaluate this method on more complex environments with more than two players, mixed strategies, a more diverse set of agent types (e.g. agents with different action spaces, reward functions, roles or strategies), and to model deviations from the assumption that the other player is just like the self. Other important avenues for future research are to design models that can adapt to non-stationary strategies of others in the environment, handle tasks with hierarchical goals, and perform well when playing with new agents at test time.

Finally, many research areas could benefit from having a model of other agents that allows reasoning about their intentions and predicting their behavior. Such models might be useful in human-robot or teacher-student interactions [Dragan et al. 2013; Fisac et al. 2017], as well as for value alignment problems [Hadfield-Menell et al. 2016]. Additionally, these methods could be useful for model-based reinforcement learning in multi-agent settings, since the accuracy of the forward model strongly depends on the ability of predicting others' behavior.

Concurrent with our work, Rabinowitz et al. [2018] proposed a theory of mind neural network, which uses meta-learning to build models of agents. In contrast with our work, Rabinowitz et al. [2018] do not evaluate their approach on settings where agents are simultaneously learning how to maximize their individual rewards. Instead, they aim to model the behaviors of pretrained agents from the perspective of a static observer which does not take actions in the environment. Since our work was published, others have been building on it and proposing alternative methods for modeling other agents. For example, Tacchetti et al. [2018] use relational forward models to predict agent's future behavior in multi-agent environments, Jaques et al. [2019] reward agents for having causal influence on other agents' actions to improve coordination and cooperation in Multi-Agent Reinforcement Learning (MARL), while Lee et al. [2021] show that joint attention improves multi-agent coordination and social learning. Recently, Fuchs et al. [2021] use an intrinsic reward enabled by theory of mind which incentivizes agents to share strategically relevant private knowledge with their teammates. The authors show promising results on the partially observable card game Hanabi [Bard et al. 2020].

8 | Conclusion

This thesis explores different ways of designing deep reinforcement learning agents which can solve a task in a wide range of settings. In particular, we studied the following problems: (i) zero-shot generalization to new instances of a task with unseen states, (ii) exploration in procedurally generated environments, (iii) fast adaptation to new dynamics, and (iv) strategic interaction with other agents having various goals.. For each of these, we introduced a novel algorithm or technique that improves upon the previous state-of-the-art on some popular benchmarks.

In chapters 3 and 4, we investigate the problem of generalizing to new instances of a task after training on a relatively small number of such instances. This implies generalizing to completely new underlying states, high-dimensional observations, and sometimes dynamics.

In chapter 3, we introduce Data-regularized Actor Critic or DrAC, an approach which encourages the policy and value function of an image-based deep reinforcement learning agent to be invariant to various transformations of the observation. The use of data augmentation in RL is different from its use in supervised or self-supervised learning due to the nonstationarity of the data (*i.e.* both inputs and targets). We demonstrate that using data augmentation in a naive way (*i.e.* same as in supervised or self-supervised learning) as part of an agent's buffer is not theoretically sound for certain RL algorithms. Regularizing the policy and value function as DrAC does avoids this pitfall, resulting in a more principled way of using data augmentation in RL. Since each RL task can benefit from a different type of invariance, we also propose a method for automatically selecting the most effective type of data augmentation from a given set, which is based on the upper confidence bound algorithm or UCB [Auer 2002]. Combining these two ideas results in UCB-DrAC, which achieved state-of-the-art test performance on challenging procedurally generated tasks with image-based observations and discrete action spaces. An interesting extension of our work would be to assume that it is best to use multiple types of augmentations simultaneously (rather than a single one) to maximally improve generalization on a certain task. One limitation of our work is that it does not consider the timing of when the regularization is applied during the process of training the agent to solve the task. Since our paper was released, follow-up work has already studied this problem and concluded that the optimal timing depends on the type of task and augmentation considered [Ko and Ok 2021]. Furthermore, our method selects an augmentation from a fixed set, but a promising future direction is to explore meta-learning the parameters of a transformation using a more expressive function class such as a neural network. This has the potential of finding the best inductive biases for each task directly from data, without the need to rely on expert knowledge or human priors.

In chapter 4, we discuss a different problem that appears when we want to generalize to new instances of a task, particularly in partially observed environments with varying degrees of difficulty. In this setting, more information is needed to estimate the value function than to learn the optimal policy. When using a shared network to represent both the policy and the value function, as is common practice in deep RL from pixels, this leads to overfitting. To alleviate this problem, we propose Invariant Decoupled Advantage Actor-Critic or IDAAC, which predicts the advantage instead of the value as an auxiliary task to train the policy network and encourages the representations to be invariant to the level (*i.e.* task instance). At the time of publication, our approach achieved a new state-of-the-art on the a popular benchmark with procedurally generated games where the agent is trained directly from images. The solution we propose here is only a first step towards solving the policy-value representation asymmetry and we hope many other ideas will be explored in future work. A promising avenue for future work is to investigate other auxiliary losses in order to efficiently learn even more general behaviors. One desirable property of such auxiliary losses is to capture the minimal set of features needed to act in the environment. Further investigating why using the advantage instead of the value, particularly from a theoretical perspective, could improve our understanding of what is needed to learn good representations and could open up new research directions.

In chapter 5, we study the problem of exploration in procedurally generated environments where it is unlikely that the agent visits a state more than once. In this setting, the agent needs good generalization abilities to solve the task, especially when the reward is sparse. We show that popular exploration methods (which are very effective in singleton environments) fail when trained in procedurally generated environments where the agent needs to solve many different instances of the same task. We propose Rewarding Impact-Driven Exploration or RIDE, a new type of intrinsic reward which encourages the agent to take actions that lead to significant changes in its learned state representation. RIDE achieved a new state-of-the-art on challenging gridworld tasks at the time of publication. We also show that this intrinsic reward leads to learning dynamics and behavior which are better suited for solving tasks in procedurally generated environments. One limitation of our work is that it does not differentiate between desirable and undesirable types of impact the agent might have on the environment (since in our case, there were no possible negative consequences). Developing algorithms which can distinguish between different types of impact could lead to safer and more efficient agents. Other interesting avenues for future research are: using symbolic information to characterize the type of impact an agent can have, considering longer-term effects of the agent's actions, using counterfactuals to compare the consequences of various actions, or promoting diversity among the different types of changes which can be made in the environment.

In chapter 6, we look at the problem of quickly adapting to new environment dynamics within a single episode. To make progress on this problem, we introduce the Policy-Dynamics Value Function or PD-VF and demonstrate that it achieves strong results on a few continuous control tasks. PD-VF explicitly estimates the cumulative reward in a latent space of policies and environments. At test time, only a few actions are needed to infer the environment dynamics and find a policy which maximizes the learned value function. Our work uses a quadratic form to parameterize the PD-VF, but a more general class of function approximators (such as neural networks) could also be used in principle. In addition, our formulation can be extended to model not only the space of behaviors and dynamics, but also the space of reward functions or other agents' strategies. The PD-VF framework can also be used to evaluate a family of policies and environments on other metrics which might be useful for decision making, such as reward variance, agent prosociality, or deviation from expert behavior. Another interesting direction for future work is to integrate additional constraints or prior knowledge to the optimization problem. Finally, PD-VF can be a useful tool for an agent to continually integrate knowledge about different skills and dynamics in a lifelong learning setting.

In chapter 7, we consider the multi-agent RL setting with imperfect information in which each agent is trying to maximize its own utility. The agents' goals are uniformly sampled from a given set at the beginning of each episode. Thus, the agent must learn policies which can adapt to different types of opponents or cooperators. We propose Self Other-Modeling or SOM, a new approach in which an agent learns a policy conditioned on its own goal and its current guess of the other agent's goal. Each agent uses its own policy to predict the other's actions and update its belief of the other's goal. In both cooperative and adversarial settings, SOM achieves better results than other methods. Our approach is simple and flexible, and does not require any extra parameters to model the other agent. Since the agent uses its own policy to model others, SOM can be easily adapted to settings with more than two players. However, the assumption that others would behave just like you given a certain goal does not always hold. Modeling deviations from this assumption is an important direction for future work. Another limitation of our method is that it does not currently handle the case in which the other agents have new goals at test time. Other important avenues for future research are to design models that can adapt to nonstationary strategies of others, handle tasks with hierarchical goals, and perform well when playing with new agents at test time.

Despite rapid progress in recent years, current deep reinforcement learning agents still struggle to generalize or quickly adapt to new scenarios. This thesis makes a few steps towards designing more general and adaptive agents, but the problem is far from being solved. One of the main challenges is learning state representations which generalize beyond the training environments. While some of the methods proposed for improving generalization in supervised and self-supervised learning can help in RL as well, as emphasized here, a naive transfer of ideas can hurt performance in some cases. Thus, one needs to be careful when adapting ideas developed for different problem settings and domains in order to take into account the specific properties of RL such as its sequential nature and nonstationary distribution of data. In addition, many different facets of generalization need to be addressed in RL. In this thesis, we considered the problem of generalization with respect to the state space, observation space, transition function, and other agents. However, more research is needed to design agents that can simultaneously generalize across all these dimensions. While here we introduce a few approaches for improving generalization to new instances of the same task, it is also important to develop methods that are able to constantly expand their knowledge and capabilities by efficiently learning new skills. To make progress on these problems, advances in both algorithms and environments are needed. In particular, more complex, diverse, and open-ended environments where agents are constantly faced with new challenges could inspire breakthroughs in never-ending learning.

Bibliography

Abbeel, P. and Ng, A. Y. (2004). Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the twenty-first international conference on Machine learning*, page 1. ACM.

Achiam, J. and Sastry, S. (2017). Surprise-based intrinsic motivation for deep reinforcement learning. *CoRR*, abs/1703.01732.

Agarwal, R., Machado, M. C., Castro, P. S., and Bellemare, M. G. (2021). Contrastive behavioral similarity embeddings for generalization in reinforcement learning.

Ammar, H. B., Eaton, E., Taylor, M. E., Mocanu, D. C., Driessens, K., Weiss, G., and Tuyls, K. (2014). An automated measure of mdp similarity for transfer in reinforcement learning. In *Workshops at the Twenty-Eighth AAAI Conference on Artificial Intelligence*.

Ammar, H. B., Tuyls, K., Taylor, M. E., Driessens, K., and Weiss, G. (2012). Reinforcement learning transfer via sparse coding. In *Proceedings of the 11th international conference on autonomous agents and multiagent systems*, volume 1, pages 383–390. International Foundation for Autonomous Agents and Multiagent Systems ....

Andreas, J., Klein, D., and Levine, S. (2017). Modular multitask reinforcement learning with policy sketches. In *International Conference on Machine Learning*, pages 166–175.

Andrychowicz, M., Raichuk, A., Stanczyk, P., Orsini, M., Girgin, S., Marinier, R., Hussenot, L., Geist, M., Pietquin, O., Michalski, M., Gelly, S., and Bachem, O. (2020). What matters in on-policy reinforcement learning? a large-scale empirical study. *ArXiv*, abs/2006.05990.

Arjovsky, M., Bottou, L., Gulrajani, I., and Lopez-Paz, D. (2019). Invariant risk minimization. *ArXiv*, abs/1907.02893.

Arnekvist, I., Kragic, D., and Stork, J. A. (2018). Vpe: Variational policy embedding for transfer reinforcement learning. *2019 International Conference on Robotics and Automation (ICRA)*, pages 36–42.

Auer, P. (2002). Using confidence bounds for exploitation-exploration trade-offs. *Journal of Machine Learning Research*, 3(Nov):397–422.

Axelrod, R. (2006). Agent-based modeling as a bridge between disciplines. *Handbook of computational economics*, 2:1565–1584.

Aytar, Y., Pfaff, T., Budden, D., Paine, T., Wang, Z., and de Freitas, N. (2018). Playing hard exploration games by watching youtube. In *Advances in Neural Information Processing Systems*, pages 2930–2941.

Bahdanau, D., Cho, K., and Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.

Ball, P. J., Lu, C., Parker-Holder, J., and Roberts, S. (2021). Augmented world models facilitate zero-shot dynamics generalization from a single offline environment. *arXiv preprint arXiv:2104.05632*.

Bard, N., Foerster, J. N., Chandar, S., Burch, N., Lanctot, M., Song, H. F., Parisotto, E., Dumoulin, V., Moitra, S., Hughes, E., et al. (2020). The hanabi challenge: A new frontier for ai research. *Artificial Intelligence*, 280:103216.

Barreto, A., Dabney, W., Munos, R., Hunt, J. J., Schaul, T., van Hasselt, H. P., and Silver, D. (2017). Successor features for transfer in reinforcement learning. In *Advances in neural information processing systems*, pages 4055–4065.

Barth-Maron, G., Hoffman, M. W., Budden, D., Dabney, W., Horgan, D., Dhruva, T., Muldal, A., Heess, N., and Lillicrap, T. (2018). Distributed distributional deterministic policy gradients. *ArXiv*, abs/1804.08617.

Beattie, C., Leibo, J. Z., Teplyashin, D., Ward, T., Wainwright, M., Küttler, H., Lefrancq, A., Green, S., Valdés, V., Sadik, A., Schrittwieser, J., Anderson, K., York, S., Cant, M., Cain, A., Bolton, A., Gaffney, S., King, H., Hassabis, D., Legg, S., and Petersen, S. (2016). Deepmind lab. *CoRR*, abs/1612.03801.

Becker, S. and Hinton, G. E. (1992). Self-organizing neural network that discovers surfaces in random-dot stereograms. *Nature*, 355:161–163.

Bellemare, M., Srinivasan, S., Ostrovski, G., Schaul, T., Saxton, D., and Munos, R. (2016). Unifying count-based exploration and intrinsic motivation. In *Advances in Neural Information Processing Systems*, pages 1471–1479.

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

Bellman, R. (1957). A markovian decision process. *Journal of mathematics and mechanics*, pages 679–684.

Bengio, E., Pineau, J., and Precup, D. (2020). Interference and generalization in temporal difference learning. *ArXiv*, abs/2003.06350.

Berner, C., Brockman, G., Chan, B., Cheung, V., Dębiak, P., Dennison, C., Farhi, D., Fischer, Q., Hashme, S., Hesse, C., et al. (2019a). Dota 2 with large scale deep reinforcement learning. *arXiv preprint arXiv:1912.06680*.

Berner, C., Brockman, G., Chan, B., Cheung, V., Debiak, P., Dennison, C., Farhi, D., Fischer, Q., Hashme, S., Hesse, C., Józefowicz, R., Gray, S., Olsson, C., Pachocki, J. W., Petrov, M., de Oliveira Pinto, H. P., Raiman, J., Salimans, T., Schlatter, J., Schneider, J., Sidor, S., Sutskever, I., Tang, J., Wolski, F., and Zhang, S. (2019b). Dota 2 with large scale deep reinforcement learning. *ArXiv*, abs/1912.06680.

Bertrán, M., Martínez, N., Phielipp, M., and Sapiro, G. (2020). Instance based generalization in reinforcement learning. *ArXiv*, abs/2011.01089.

Borsa, D., Barreto, A., Quan, J., Mankowitz, D., Munos, R., van Hasselt, H., Silver, D., and Schaul, T. (2018). Universal successor features approximators. *arXiv preprint arXiv:1812.07626*.

Borsa, D., Graepel, T., and Shawe-Taylor, J. (2016). Learning shared representations in multi-task reinforcement learning. *arXiv preprint arXiv:1603.02041*.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. (2020). Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*.

Burda, Y., Edwards, H., Pathak, D., Storkey, A. J., Darrell, T., and Efros, A. A. (2019a). Large-scale study of curiosity-driven learning. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*.

Burda, Y., Edwards, H., Storkey, A. J., and Klimov, O. (2019b). Exploration by random network distillation. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*.

Campero, A., Raileanu, R., Küttler, H., Tenenbaum, J. B., Rocktäschel, T., and Grefenstette, E. (2020). Learning with amigo: Adversarially motivated intrinsic goals. *arXiv preprint arXiv:2006.12122*.

Chen, J. Z. (2020). Reinforcement learning generalization with surprise minimization. *ArXiv*, abs/2004.12399.

Chen, T., Kornblith, S., Norouzi, M., and Hinton, G. E. (2020). A simple framework for contrastive learning of visual representations. *ArXiv*, abs/2002.05709.

Chevalier-Boisvert, M., Willems, L., and Pal, S. (2018). Minimalistic gridworld environment for openai gym. `https://github.com/maximecb/gym-minigrid`.

Cho, K., Van Merriënboer, B., Bahdanau, D., and Bengio, Y. (2014). On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259*.

Choi, J., Guo, Y., Moczulski, M., Oh, J., Wu, N., Norouzi, M., and Lee, H. (2019). Contingency-aware exploration in reinforcement learning. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*.

Ciregan, D., Meier, U., and Schmidhuber, J. (2012). Multi-column deep neural networks for image classification. In *2012 IEEE conference on computer vision and pattern recognition*, pages 3642–3649. IEEE.

Ciresan, D. C., Meier, U., Masci, J., Gambardella, L. M., and Schmidhuber, J. (2011). High-performance neural networks for visual object classification. *ArXiv*, abs/1102.0183.

Cireşan, D. C., Meier, U., Masci, J., Gambardella, L. M., and Schmidhuber, J. (2011). High-performance neural networks for visual object classification. *arXiv preprint arXiv:1102.0183*.

Clevert, D.-A., Unterthiner, T., and Hochreiter, S. (2015). Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289*.

Co-Reyes, J. D., Liu, Y., Gupta, A., Eysenbach, B., Abbeel, P., and Levine, S. (2018). Self-consistent trajectory autoencoder: Hierarchical reinforcement learning with trajectory embeddings. In *ICML*.

Cobbe, K., Hesse, C., Hilton, J., and Schulman, J. (2019a). Leveraging procedural generation to benchmark reinforcement learning. *ArXiv*, abs/1912.01588.

Cobbe, K., Hesse, C., Hilton, J., and Schulman, J. (2019b). Leveraging procedural generation to benchmark reinforcement learning. *arXiv preprint arXiv:1912.01588*.

Cobbe, K., Hilton, J., Klimov, O., and Schulman, J. (2020). Phasic policy gradient. *ArXiv*, abs/2009.04416.

Cobbe, K., Klimov, O., Hesse, C., Kim, T., and Schulman, J. (2018). Quantifying generalization in reinforcement learning. *arXiv preprint arXiv:1812.02341*.

Cobbe, K., Klimov, O., Hesse, C., Kim, T., and Schulman, J. (2019c). Quantifying generalization in reinforcement learning. In *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, pages 1282–1289.

Cobbe, K., Klimov, O., Hesse, C., Kim, T., and Schulman, J. (2019d). Quantifying generalization in reinforcement learning. In *ICML*.

Cubuk, E. D., Zoph, B., Mané, D., Vasudevan, V., and Le, Q. V. (2019a). Autoaugment: Learning augmentation strategies from data. *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 113–123.

Cubuk, E. D., Zoph, B., Shlens, J., and Le, Q. V. (2019b). Randaugment: Practical automated data augmentation with a reduced search space. *arXiv: Computer Vision and Pattern Recognition*.

Cully, A., Clune, J., Tarapore, D., and Mouret, J.-B. (2015). Robots that can adapt like animals. *Nature*, 521:503–507.

Da Silva, B., Konidaris, G., and Barto, A. (2012). Learning parameterized skills. *arXiv preprint arXiv:1206.6398*.

Das, A., Kottur, S., Moura, J. M., Lee, S., and Batra, D. (2017). Learning cooperative visual dialog agents with deep reinforcement learning. *arXiv preprint arXiv:1703.06585*.

Davidson, A. (1999). Using artificial neural networks to model opponents in texas hold'em. *Unpublished manuscript*.

Denton, E. L. and Birodkar, V. (2017). Unsupervised learning of disentangled representations from video. In *NIPS*.

Devin, C., Gupta, A., Darrell, T., Abbeel, P., and Levine, S. (2016). Learning modular neural network policies for multi-task and multi-robot transfer. *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2169–2176.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

DeVries, T. and Taylor, G. W. (2017). Improved regularization of convolutional neural networks with cutout. *arXiv preprint arXiv:1708.04552*.

Dilokthanakul, N., Kaplanis, C., Pawlowski, N., and Shanahan, M. (2019). Feature control as intrinsic motivation for hierarchical reinforcement learning. *IEEE transactions on neural networks and learning systems*.

Doshi-Velez, F. and Konidaris, G. (2013). Hidden parameter markov decision processes: A semi-parametric regression approach for discovering latent task parametrizations. *IJCAI : proceedings of the conference*, 2016:1432–1440.

Dosovitskiy, A., Fischer, P., Springenberg, J. T., Riedmiller, M. A., and Brox, T. (2016). Discriminative unsupervised feature learning with exemplar convolutional neural networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 38:1734–1747.

Dragan, A. D., Lee, K. C., and Srinivasa, S. S. (2013). Legibility and predictability of robot motion. In *Human-Robot Interaction (HRI), 2013 8th ACM/IEEE International Conference on*, pages 301–308. IEEE.

Duan, Y., Andrychowicz, M., Stadie, B. C., Ho, J., Schneider, J., Sutskever, I., Abbeel, P., and Zaremba, W. (2017). One-shot imitation learning. In *NIPS*.

Duan, Y., Schulman, J., Chen, X., Bartlett, P. L., Sutskever, I., and Abbeel, P. (2016). Rl^2: Fast reinforcement learning via slow reinforcement learning. *ArXiv*, abs/1611.02779.

Ecoffet, A., Huizinga, J., Lehman, J., Stanley, K. O., and Clune, J. (2019). Go-explore: a new approach for hard-exploration problems. *arXiv preprint arXiv:1901.10995*.

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

Espeholt, L., Soyer, H., Munos, R., Simonyan, K., Mnih, V., Ward, T., Doron, Y., Firoiu, V., Harley, T., Dunning, I., Legg, S., and Kavukcuoglu, K. (2018b). IMPALA: scalable distributed deep-rl with importance weighted actor-learner architectures. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, pages 1406–1415.

Eysenbach, B., Gupta, A., Ibarz, J., and Levine, S. (2019). Diversity is all you need: Learning skills without a reward function. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*.

Fan, J. and Li, W. (2021). Robust deep reinforcement learning via multi-view information bottleneck. *arXiv preprint arXiv:2102.13268*.

Fan, L., Wang, G., Huang, D.-A., Yu, Z., Fei-Fei, L., Zhu, Y., and Anandkumar, A. (2021). Secant: Self-expert cloning for zero-shot generalization of visual policies. *arXiv preprint arXiv:2106.09678*.

Fang, B., Jiang, M., and Shen, J. J. (2019). Paganda : An adaptive task-independent automatic data augmentation.

Fang, K., Zhu, Y., Savarese, S., and Fei-Fei, L. (2020). Adaptive procedural task generation for hard-exploration problems. *arXiv preprint arXiv:2007.00350*.

Farebrother, J., Machado, M. C., and Bowling, M. H. (2018). Generalization and regularization in dqn. *ArXiv*, abs/1810.00123.

Finn, C., Abbeel, P., and Levine, S. (2017). 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.

Fisac, J. F., Gates, M. A., Hamrick, J. B., Liu, C., Hadfield-Menell, D., Palaniappan, M., Malik, D., Sastry, S. S., Griffiths, T. L., and Dragan, A. D. (2017). Pragmatic-pedagogic value alignment. *arXiv preprint arXiv:1707.06354*.

Foerster, J., Assael, Y. M., de Freitas, N., and Whiteson, S. (2016a). Learning to communicate with deep multi-agent reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 2137–2145.

Foerster, J. N., Assael, Y. M., de Freitas, N., and Whiteson, S. (2016b). Learning to communicate to solve riddles with deep distributed recurrent q-networks. *arXiv preprint arXiv:1602.02672*.

Foerster, J. N., Chen, R. Y., Al-Shedivat, M., Whiteson, S., Abbeel, P., and Mordatch, I. (2017). Learning with opponent-learning awareness. *arXiv preprint arXiv:1709.04326*.

Foley, J., Tosch, E., Clary, K., and Jensen, D. (2018). Toybox: Better atari environments for testing reinforcement learning agents. *CoRR*, abs/1812.02850.

Fortunato, M., Azar, M. G., Piot, B., Menick, J., Hessel, M., Osband, I., Graves, A., Mnih, V., Munos, R., Hassabis, D., Pietquin, O., Blundell, C., and Legg, S. (2018). Noisy networks for exploration. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*.

Fuchs, A., Walton, M., Chadwick, T., and Lange, D. (2021). Theory of mind for deep reinforcement learning in hanabi. *arXiv preprint arXiv:2101.09328*.

Gallese, V. and Goldman, A. (1998). Mirror neurons and the simulation theory of mind-reading. *Trends in cognitive sciences*, 2(12):493–501.

Gamrian, S. and Goldberg, Y. (2019). Transfer learning for related reinforcement learning tasks via image-to-image translation. *ArXiv*, abs/1806.07377.

Gordon, R. M. (1986). Folk psychology as simulation. *Mind & Language*, 1(2):158–171.

Goyal, A., Islam, R., Strouse, D., Ahmed, Z., Larochelle, H., Botvinick, M., Bengio, Y., and Levine, S. (2019). Infobot: Transfer and exploration via the information bottleneck. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*.

Grefenstette, E., Amos, B., Yarats, D., Htut, P. M., Molchanov, A., Meier, F., Kiela, D., Cho, K., and Chintala, S. (2019). Generalized inner loop meta-learning. *arXiv preprint arXiv:1910.01727*.

Gregor, K., Rezende, D. J., and Wierstra, D. (2017). Variational intrinsic control. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Workshop Track Proceedings*.

Grigsby, J. and Qi, Y. (2020). Measuring visual generalization in continuous control from pixels. *ArXiv*, abs/2010.06740.

Gu, S., Holly, E., Lillicrap, T., and Levine, S. (2017). Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates. In *2017 IEEE international conference on robotics and automation (ICRA)*, pages 3389–3396. IEEE.

Gupta, A., Devin, C., Liu, Y., Abbeel, P., and Levine, S. (2017). Learning invariant feature spaces to transfer skills with reinforcement learning. *arXiv preprint arXiv:1703.02949*.

Haarnoja, T., Zhou, A., Abbeel, P., and Levine, S. (2018). Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *ICML*.

Hadfield-Menell, D., Milli, S., Abbeel, P., Russell, S. J., and Dragan, A. (2017). Inverse reward design. In *Advances in Neural Information Processing Systems*, pages 6768–6777.

Hadfield-Menell, D., Russell, S. J., Abbeel, P., and Dragan, A. (2016). Cooperative inverse reinforcement learning. In *Advances in neural information processing systems*, pages 3909–3917.

Hansen, S., Dabney, W., Barreto, A., Van de Wiele, T., Warde-Farley, D., and Mnih, V. (2019). Fast task inference with variational intrinsic successor features. *arXiv preprint arXiv:1906.05030*.

Hausman, K., Springenberg, J. T., Wang, Z., Heess, N. M. O., and Riedmiller, M. A. (2018). Learning an embedding space for transferable robot skills. In *ICLR*.

He, H., Boyd-Graber, J., Kwok, K., and Daumé III, H. (2016). Opponent modeling in deep reinforcement learning. In *Proceedings of The 33rd International Conference on Machine Learning*, pages 1804–1813.

He, K., Fan, H., Wu, Y., Xie, S., and Girshick, R. B. (2019). Momentum contrast for unsupervised visual representation learning. *ArXiv*, abs/1911.05722.

He, Z., Julian, R., Heiden, E., Zhang, H., Schaal, S., Lim, J. J., Sukhatme, G., and Hausman, K. (2018). Zero-shot skill composition and simulation-to-real transfer by learning task representations. *arXiv preprint arXiv:1810.02422*.

Hénaff, O. J., Srinivas, A., Fauw, J. D., Razavi, A., Doersch, C., Eslami, S. M. A., and van den Oord, A. (2019). Data-efficient image recognition with contrastive predictive coding. *ArXiv*, abs/1905.09272.

Henderson, P., Islam, R., Bachman, P., Pineau, J., Precup, D., and Meger, D. (2018). Deep reinforcement learning that matters. In *Thirty-Second AAAI Conference on Artificial Intelligence*.

Hessel, M., Soyer, H., Espeholt, L., Czarnecki, W., Schmitt, S., and van Hasselt, H. (2019). Multi-task deep reinforcement learning with popart. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 3796–3803.

Higgins, I., Pal, A., Rusu, A. A., Matthey, L., Burgess, C., Pritzel, A., Botvinick, M. M., Blundell, C., and Lerchner, A. (2017). Darla: Improving zero-shot transfer in reinforcement learning. In *ICML*.

Hinton, G., Deng, L., Yu, D., Dahl, G. E., Mohamed, A.-r., Jaitly, N., Senior, A., Vanhoucke, V., Nguyen, P., Sainath, T. N., et al. (2012). Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal processing magazine*, 29(6):82–97.

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

Houthooft, R., Chen, R. Y., Isola, P., Stadie, B. C., Wolski, F., Ho, J., and Abbeel, P. (2018). Evolved policy gradients. *ArXiv*, abs/1802.04821.

Houthooft, R., Chen, X., Duan, Y., Schulman, J., De Turck, F., and Abbeel, P. (2016). Vime: Variational information maximizing exploration. In *Advances in Neural Information Processing Systems*, pages 1109–1117.

Humplik, J., Galashov, A., Hasenclever, L., Ortega, P. A., Teh, Y. W., and Heess, N. (2019). Meta reinforcement learning as task inference. *arXiv preprint arXiv:1905.06424*.

Igl, M., Ciosek, K., Li, Y., Tschiatschek, S., Zhang, C., Devlin, S., and Hofmann, K. (2019). Generalization in reinforcement learning with selective noise injection and information bottleneck. In *Advances in Neural Information Processing Systems*, pages 13956–13968.

Igl, M., Farquhar, G., Luketina, J., Böhmer, W., and Whiteson, S. (2020). The impact of non-stationarity on generalisation in deep reinforcement learning. *ArXiv*, abs/2006.05826.

Ilse, M., Tomczak, J. M., and Forré, P. (2020). Designing data augmentation for simulating interventions. *arXiv preprint arXiv:2005.01856*.

Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. *ArXiv*, abs/1502.03167.

Jaderberg, M., Czarnecki, W. M., Dunning, I., Marris, L., Lever, G., Castaneda, A. G., Beattie, C., Rabinowitz, N. C., Morcos, A. S., Ruderman, A., et al. (2019). Human-level performance in 3d multiplayer games with population-based reinforcement learning. *Science*, 364(6443):859–865.

Jaderberg, M., Mnih, V., Czarnecki, W. M., Schaul, T., Leibo, J. Z., Silver, D., and Kavukcuoglu, K. (2017). Reinforcement learning with unsupervised auxiliary tasks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*.

Jang, E., Gu, S., and Poole, B. (2016). Categorical reparameterization with gumbel-softmax. *arXiv preprint arXiv:1611.01144*.

Jaques, N., Lazaridou, A., Hughes, E., Gulcehre, C., Ortega, P., Strouse, D., Leibo, J. Z., and De Freitas, N. (2019). Social influence as intrinsic motivation for multi-agent deep reinforcement learning. In *International Conference on Machine Learning*, pages 3040–3049. PMLR.

Jernite, Y., Srinet, K., Gray, J., and Szlam, A. (2019). Craftassist instruction parsing: Semantic parsing for a minecraft assistant. *CoRR*, abs/1905.01978.

Jiang, M., Grefenstette, E., and Rocktäschel, T. (2020). Prioritized level replay. *ArXiv*, abs/2010.03934.

Johnson, M., Hofmann, K., Hutton, T., and Bignell, D. (2016). The malmo platform for artificial intelligence experimentation. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, pages 4246–4247.

Juliani, A., Khalifa, A., Berges, V., Harper, J., Teng, E., Henry, H., Crespi, A., Togelius, J., and Lange, D. (2019a). Obstacle tower: A generalization challenge in vision, control, and planning. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, pages 2684–2691.

Juliani, A., Khalifa, A., Berges, V.-P., Harper, J., Henry, H., Crespi, A., Togelius, J., and Lange, D. (2019b). Obstacle tower: A generalization challenge in vision, control, and planning. *ArXiv*, abs/1902.01378.

Justesen, N., Torrado, R., Bontrager, P., Khalifa, A., Togelius, J., and Risi, S. (2018a). Illuminating generalization in deep reinforcement learning through procedural level generation. *arXiv: Learning*.

Justesen, N., Torrado, R. R., Bontrager, P., Khalifa, A., Togelius, J., and Risi, S. (2018b). Illuminating generalization in deep reinforcement learning through procedural level generation. *arXiv preprint arXiv:1806.10729*.

Kauten, C. (2018). Super Mario Bros for OpenAI Gym. GitHub.

Kay, W., Carreira, J., Simonyan, K., Zhang, B., Hillier, C., Vijayanarasimhan, S., Viola, F., Green, T., Back, T., Natsev, A., Suleyman, M., and Zisserman, A. (2017). The kinetics human action video dataset. *ArXiv*, abs/1705.06950.

Kempka, M., Wydmuch, M., Runc, G., Toczek, J., and Jaśkowski, W. (2016). Vizdoom: A doom-based ai research platform for visual reinforcement learning. In *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–8. IEEE.

Khetarpal, K., Riemer, M., Rish, I., and Precup, D. (2020). Towards continual reinforcement learning: A review and perspectives. *arXiv preprint arXiv:2012.13490*.

Killian, T. W., Konidaris, G., and Doshi-Velez, F. (2017). Robust and efficient transfer learning with hidden parameter markov decision processes. *Advances in neural information processing systems*, 30:6250–6261.

Kingma, D. P. and Ba, J. (2015). Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980.

Kingma, D. P. and Welling, M. (2013). Auto-encoding variational bayes. *CoRR*, abs/1312.6114.

Kleiman-Weiner, M., Ho, M. K., Austerweil, J. L., Littman, M. L., and Tenenbaum, J. B. (2016). Coordinate to cooperate or compete: abstract goals and joint intentions in social interaction. In *COGSCI*.

Klyubin, A. S., Polani, D., and Nehaniv, C. L. (2005). All else being equal be empowered. In *European Conference on Artificial Life*, pages 744–753. Springer.

Ko, B. and Ok, J. (2021). Time matters in using data augmentation for vision-based deep reinforcement learning. *arXiv preprint arXiv:2102.08581*.

Kostrikov, I., Yarats, D., and Fergus, R. (2020). Image augmentation is all you need: Regularizing deep reinforcement learning from pixels. *arXiv preprint arXiv:2004.13649*.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105.

Küttler, H., Nardelli, N., Lavril, T., Selvatici, M., Sivakumar, V., Rocktäschel, T., and Grefenstette, E. (2019). TorchBeast: A PyTorch Platform for Distributed RL. *arXiv preprint arXiv:1910.03552.*

Kuttler, H., Nardelli, N., Miller, A. H., Raileanu, R., Selvatici, M., Grefenstette, E., and Rocktäschel, T. (2020). The nethack learning environment. *ArXiv*, abs/2006.13760.

Laskin, M., Lee, K., Stooke, A., Pinto, L., Abbeel, P., and Srinivas, A. (2020). Reinforcement learning with augmented data. *arXiv preprint arXiv:2004.14990.*

Lazaric, A. (2012). Transfer in reinforcement learning: a framework and a survey. In *Reinforcement Learning*, pages 143–173. Springer.

Lazaridou, A., Peysakhovich, A., and Baroni, M. (2016). Multi-agent cooperation and the emergence of (natural) language. *arXiv preprint arXiv:1612.07182.*

LeCun, Y., Boser, B. E., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W. E., and Jackel, L. D. (1989). Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1:541–551.

LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition.

Lee, D., Jaques, N., Kew, C., Eck, D., Schuurmans, D., and Faust, A. (2021). Joint attention for multi-agent coordination and social learning. *arXiv preprint arXiv:2104.07750.*

Lee, K., Lee, K., Shin, J., and Lee, H. (2020). Network randomization: A simple technique for generalization in deep reinforcement learning. In *International Conference on Learning Representations. https://openreview. net/forum.*

Lee, S. and Chung, S.-Y. (2021). Improving generalization in meta-rl with imaginary tasks from latent dynamics mixture. *arXiv preprint arXiv:2105.13524.*

Leibo, J. Z., Zambaldi, V., Lanctot, M., Marecki, J., and Graepel, T. (2017). Multi-agent reinforcement learning in sequential social dilemmas. In *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems*, pages 464–473. International Foundation for Autonomous Agents and Multiagent Systems.

Leike, J., Martic, M., Krakovna, V., Ortega, P. A., Everitt, T., Lefrancq, A., Orseau, L., and Legg, S. (2017). Ai safety gridworlds. *arXiv preprint arXiv:1711.09883*.

Lerer, A. and Peysakhovich, A. (2017). Maintaining cooperation in complex social dilemmas using deep reinforcement learning. *arXiv preprint arXiv:1707.01068*.

Lesort, T., Rodríguez, N. D., Goudou, J., and Filliat, D. (2018). State representation learning for control: An overview. *Neural Networks*, 108:379–392.

Li, Y., Hu, G., Wang, Y., Hospedales, T. M., Robertson, N. M., and Yang, Y. (2020). Dada: Differentiable automatic data augmentation. *ArXiv*, abs/2003.03780.

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

Little, D. Y.-J. and Sommer, F. T. (2013). Learning and exploration in action-perception loops. *Frontiers in neural circuits*, 7:37.

Lockett, A. J., Chen, C. L., and Miikkulainen, R. (2007). Evolving explicit opponent models in game playing. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 2106–2113. ACM.

Lowe, R., Wu, Y., Tamar, A., Harb, J., Abbeel, P., and Mordatch, I. (2017). Multi-agent actor-critic for mixed cooperative-competitive environments. *arXiv preprint arXiv:1706.02275*.

Machado, M. C., Bellemare, M. G., and Bowling, M. (2018a). Count-based exploration with the successor representation. *arXiv preprint arXiv:1807.11622*.

Machado, M. C., Bellemare, M. G., Talvitie, E., Veness, J., Hausknecht, M., and Bowling, M. (2018b). Revisiting the arcade learning environment: Evaluation protocols and open problems for general agents. *Journal of Artificial Intelligence Research*, 61:523–562.

Machado, M. C., Bellemare, M. G., Talvitie, E., Veness, J., Hausknecht, M. J., and Bowling, M. H. (2018c). Revisiting the arcade learning environment: Evaluation protocols and open problems for general agents. In *IJCAI*.

Maddison, C. J., Mnih, A., and Teh, Y. W. (2016). The concrete distribution: A continuous relaxation of discrete random variables. *arXiv preprint arXiv:1611.00712*.

Madjiheurem, S. and Toni, L. (2019). State2vec: Off-policy successor features approximators. *arXiv preprint arXiv:1910.10277*.

Marino, K., Gupta, A., Fergus, R., and Szlam, A. (2019). Hierarchical RL using an ensemble of proprioceptive periodic policies. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*.

Martin, J., Sasikumar, S. N., Everitt, T., and Hutter, M. (2017). Count-based exploration in feature space for reinforcement learning. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pages 2471–2478.

Mazoure, B., Ahmed, A. M., MacAlpine, P., Hjelm, R. D., and Kolobov, A. (2021). Cross-trajectory representation learning for zero-shot generalization in rl. *arXiv preprint arXiv:2106.02193*.

Mazoure, B., des Combes, R. T., Doan, T., Bachman, P., and Hjelm, R. D. (2020). Deep reinforcement and infomax learning. *ArXiv*, abs/2006.07217.

Miao, Y., Song, X., Peng, D., Yue, S., Brevdo, E., and Faust, A. (2021). Rl-darts: Differentiable architecture search for reinforcement learning. *arXiv preprint arXiv:2106.02229*.

Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013). Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*.

Misra, I. and van der Maaten, L. (2019). Self-supervised learning of pretext-invariant representations. *ArXiv*, abs/1912.01991.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., and Kavukcuoglu, K. (2016a). Asynchronous methods for deep reinforcement learning. *ArXiv*, abs/1602.01783.

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

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. A. (2013). Playing atari with deep reinforcement learning. *ArXiv*, abs/1312.5602.

Modhe, N., Chattopadhyay, P., Sharma, M., Das, A., Parikh, D., Batra, D., and Vedantam, R. (2019). Unsupervised discovery of decision states for transfer in reinforcement learning. *CoRR*, abs/1907.10580.

Mordatch, I. and Abbeel, P. (2017). Emergence of grounded compositional language in multi-agent populations. *arXiv preprint arXiv:1703.04908*.

Nagabandi, A., Clavera, I., Liu, S., Fearing, R. S., Abbeel, P., Levine, S., and Finn, C. (2018). Learning to adapt in dynamic, real-world environments through meta-reinforcement learning. *arXiv preprint arXiv:1803.11347*.

Ng, A. Y., Russell, S. J., et al. (2000). Algorithms for inverse reinforcement learning. In *Icml*, pages 663–670.

Nichol, A., Pfau, V., Hesse, C., Klimov, O., and Schulman, J. (2018a). Gotta learn fast: A new benchmark for generalization in rl. *arXiv preprint arXiv:1804.03720*.

Nichol, A., Pfau, V., Hesse, C., Klimov, O., and Schulman, J. (2018b). Gotta learn fast: A new benchmark for generalization in rl. *ArXiv*, abs/1804.03720.

O'Donoghue, B., Osband, I., Munos, R., and Mnih, V. (2018). The uncertainty bellman equation and exploration. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, pages 3836–3845.

Oh, J., Singh, S., Lee, H., and Kohli, P. (2017). Zero-shot task generalization with multi-task deep reinforcement learning. *arXiv preprint arXiv:1706.05064*.

Omidshafiei, S., Pazis, J., Amato, C., How, J. P., and Vian, J. (2017). Deep decentralized multi-task multi-agent rl under partial observability. *arXiv preprint arXiv:1703.06182*.

Osband, I., Blundell, C., Pritzel, A., and Van Roy, B. (2016). Deep exploration via bootstrapped dqn. In *Advances in neural information processing systems*, pages 4026–4034.

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

Oudeyer, P.-Y. and Kaplan, F. (2009). What is intrinsic motivation? a typology of computational approaches. *Frontiers in neurorobotics*, 1:6.

Oudeyer, P.-Y., Kaplan, F., et al. (2008). How can we define intrinsic motivation. In *Proc. of the 8th Conf. on Epigenetic Robotics*, volume 5, pages 29–31.

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

Packer, C., Gao, K., Kos, J., Krähenbühl, P., Koltun, V., and Song, D. (2018a). Assessing generalization in deep reinforcement learning. *arXiv preprint arXiv:1810.12282*.

Packer, C., Gao, K., Kos, J., Krähenbühl, P., Koltun, V., and Song, D. X. (2018b). Assessing generalization in deep reinforcement learning. *ArXiv*, abs/1810.12282.

Parisotto, E., Ba, J. L., and Salakhutdinov, R. (2015). Actor-mimic: Deep multitask and transfer reinforcement learning. *arXiv preprint arXiv:1511.06342*.

Pathak, D., Agrawal, P., Efros, A. A., and Darrell, T. (2017). Curiosity-driven exploration by self-supervised prediction. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 16–17.

Paul, S., Osborne, M. A., and Whiteson, S. (2018). Fingerprint policy optimisation for robust reinforcement learning. In *ICML*.

Perez, C. F., Such, F. P., and Karaletsos, T. (2018). Efficient transfer learning and online adaptation with latent variable models for continuous control. *ArXiv*, abs/1812.03399.

Petangoda, J. C., Pascual-Diaz, S., Adam, V., Vrancx, P., and Grau-Moya, J. (2019). Disentangled skill embeddings for reinforcement learning. *ArXiv*, abs/1906.09223.

Pinto, L., Andrychowicz, M., Welinder, P., Zaremba, W., and Abbeel, P. (2018). Asymmetric actor critic for image-based robot learning. *ArXiv*, abs/1710.06542.

Pinto, L., Davidson, J., Sukthankar, R., and Gupta, A. (2017). Robust adversarial reinforcement learning. In *ICML*.

Precup, D., Sutton, R. S., and Dasgupta, S. (2001). Off-policy temporal-difference learning with function approximation. In *ICML*, pages 417–424.

Rabinowitz, N., Perbet, F., Song, F., Zhang, C., Eslami, S. A., and Botvinick, M. (2018). Machine theory of mind. In *International conference on machine learning*, pages 4218–4227. PMLR.

Racanière, S., Weber, T., Reichert, D. P., Buesing, L., Guez, A., Rezende, D. J., Badia, A. P., Vinyals, O., Heess, N., Li, Y., Pascanu, R., Battaglia, P. W., Hassabis, D., Silver, D., and Wierstra, D. (2017). Imagination-augmented agents for deep reinforcement learning. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*, pages 5690–5701.

Raileanu, R. and Fergus, R. (2021). Decoupling value and policy for generalization in reinforcement learning. *arXiv preprint arXiv:2102.10330*.

Raileanu, R., Goldstein, M., Yarats, D., Kostrikov, I., and Fergus, R. (2020). Automatic data augmentation for generalization in deep reinforcement learning. *ArXiv*, abs/2006.12862.

Raileanu, R. and Rocktäschel, T. (2020). Ride: Rewarding impact-driven exploration for procedurally-generated environments. *ArXiv*, abs/2002.12292.

Rajeswaran, A., Lowrey, K., Todorov, E., and Kakade, S. M. (2017a). Towards generalization and simplicity in continuous control. *ArXiv*, abs/1703.02660.

Rajeswaran, A., Lowrey, K., Todorov, E. V., and Kakade, S. M. (2017b). Towards generalization and simplicity in continuous control. In *Advances in Neural Information Processing Systems*, pages 6550–6561.

Rakelly, K., Zhou, A., Finn, C., Levine, S., and Quillen, D. (2019). Efficient off-policy meta-reinforcement learning via probabilistic context variables. In *International conference on machine learning*, pages 5331–5340.

Rezende, D. J. and Mohamed, S. (2015). Variational inference with normalizing flows. In *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, pages 1530–1538.

Roy, J. and Konidaris, G. (2020). Visual transfer for reinforcement learning via wasserstein domain confusion. *arXiv preprint arXiv:2006.03465*.

Russell, S. (1998). Learning agents for uncertain environments. In *Proceedings of the eleventh annual conference on Computational learning theory*, pages 101–103. ACM.

Sæmundsson, S., Hofmann, K., and Deisenroth, M. P. (2018). Meta reinforcement learning with latent variable gaussian processes. *arXiv preprint arXiv:1803.07551*.

Sahni, H., Kumar, S., Tejani, F., and Isbell, C. (2017). Learning to compose skills. *arXiv preprint arXiv:1711.11289*.

Saxe, A. M., McClelland, J. L., and Ganguli, S. (2013). Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. *arXiv preprint arXiv:1312.6120*.

Schaul, T., Horgan, D., Gregor, K., and Silver, D. (2015). Universal value function approximators. In *International conference on machine learning*, pages 1312–1320.

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

Schmidhuber, J. (1991b). 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.

Schmidhuber, J. (2006). Developmental robotics, optimal artificial curiosity, creativity, music, and the fine arts. *Connection Science*, 18(2):173–187.

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

Schulman, J., Levine, S., Abbeel, P., Jordan, M. I., and Moritz, P. (2015a). Trust region policy optimization. In *ICML*.

Schulman, J., Moritz, P., Levine, S., Jordan, M., and Abbeel, P. (2015b). High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

Shapley, L. S. (1953). Stochastic games. *Proceedings of the National Academy of Sciences*, 39(10):1095–1100.

Shi, Y., Qin, T., Liu, Y., Lu, J., Gao, Y., and Shen, D. (2019). Automatic data augmentation by learning the deterministic policy. *ArXiv*, abs/1910.08343.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., and Hassabis, D. (2016). Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–489.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., et al. (2018). A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., Sifre, L., van den Driessche, G., Graepel, T., and Hassabis, D. (2017). Mastering the game of go without human knowledge. *Nature*, 550:354–359.

Simard, P. Y., Steinkraus, D., Platt, J. C., et al. (2003). Best practices for convolutional neural networks applied to visual document analysis. In *Icdar*, volume 3.

Siriwardhana, S., Weerasakera, R., Matthies, D. J., and Nanayakkara, S. (2019). Vusfa: Variational universal successor features approximator to improve transfer drl for target driven visual navigation. *arXiv preprint arXiv:1908.06376*.

Sonar, A., Pacelli, V., and Majumdar, A. (2020). Invariant policy optimization: Towards stronger generalization in reinforcement learning. *ArXiv*, abs/2006.01096.

Song, M. J. and Kushnir, D. (2020). Impact-driven exploration with contrastive unsupervised representations.

Song, X., Jiang, Y., Tu, S., Du, Y., and Neyshabur, B. (2020). Observational overfitting in reinforcement learning. *ArXiv*, abs/1912.02975.

Srinivas, A., Laskin, M., and Abbeel, P. (2020). Curl: Contrastive unsupervised representations for reinforcement learning. *ArXiv*, abs/2004.04136.

Srivastava, N., Hinton, G. E., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15:1929–1958.

Stadie, B. C., Levine, S., and Abbeel, P. (2015). Incentivizing exploration in reinforcement learning with deep predictive models. *arXiv preprint arXiv:1507.00814*.

Stanton, C. and Clune, J. (2018). Deep curiosity search: Intra-life exploration can improve performance on challenging deep reinforcement learning problems. *arXiv preprint arXiv:1806.00553*.

Still, S. and Precup, D. (2012). An information-theoretic approach to curiosity-driven reinforcement learning. *Theory in Biosciences*, 131(3):139–148.

Stooke, A., Lee, K., Abbeel, P., and Laskin, M. (2020). Decoupling representation learning from reinforcement learning. *ArXiv*, abs/2009.08319.

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

Sukhbaatar, S., Fergus, R., et al. (2016). Learning multiagent communication with backpropagation. In *Advances in Neural Information Processing Systems*, pages 2244–2252.

Sukhbaatar, S., Szlam, A., Synnaeve, G., Chintala, S., and Fergus, R. (2015). Mazebase: A sandbox for learning from games. *arXiv preprint arXiv:1511.07401*.

Sutskever, I., Vinyals, O., and Le, Q. V. (2014). Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112.

Sutton, R., McAllester, D. A., Singh, S., and Mansour, Y. (1999). Policy gradient methods for reinforcement learning with function approximation. In *NIPS*.

Sutton, R. S., Modayil, J., Delp, M., Degris, T., Pilarski, P. M., White, A., and Precup, D. (2011). Horde: a scalable real-time architecture for learning knowledge from unsupervised sensorimotor interaction. In *AAMAS*.

Tacchetti, A., Song, H. F., Mediano, P. A., Zambaldi, V., Rabinowitz, N. C., Graepel, T., Botvinick, M., and Battaglia, P. W. (2018). Relational forward models for multi-agent learning. *arXiv preprint arXiv:1809.11044*.

Tang, H., Houthooft, R., Foote, D., Stooke, A., Chen, O. X., Duan, Y., Schulman, J., DeTurck, F., and Abbeel, P. (2017). # exploration: A study of count-based exploration for deep reinforcement learning. In *Advances in neural information processing systems*, pages 2753–2762.

Tassa, Y., Doron, Y., Muldal, A., Erez, T., Li, Y., Casas, D., Budden, D., Abdolmaleki, A., Merel, J., Lefrancq, A., Lillicrap, T., and Riedmiller, M. A. (2018a). Deepmind control suite. *ArXiv*, abs/1801.00690.

Tassa, Y., Doron, Y., Muldal, A., Erez, T., Li, Y., Casas, D. d. L., Budden, D., Abdolmaleki, A., Merel, J., Lefrancq, A., et al. (2018b). Deepmind control suite. *arXiv preprint arXiv:1801.00690.*

Taylor, M. E. and Stone, P. (2009). Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research*, 10(Jul):1633–1685.

Teh, Y., Bapst, V., Czarnecki, W. M., Quan, J., Kirkpatrick, J., Hadsell, R., Heess, N., and Pascanu, R. (2017). Distral: Robust multitask reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 4496–4506.

Tieleman, T. and Hinton, G. (2012). Rmsprop: Divide the gradient by a running average of its recent magnitude. coursera: Neural networks for machine learning. *Tech. Rep., Technical report*, page 31.

Tobin, J., Fong, R. H., Ray, A., Schneider, J., Zaremba, W., and Abbeel, P. (2017). Domain randomization for transferring deep neural networks from simulation to the real world. *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 23–30.

Todorov, E., Erez, T., and Tassa, Y. (2012a). Mujoco: A physics engine for model-based control. *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033.

Todorov, E., Erez, T., and Tassa, Y. (2012b). Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033. IEEE.

Totaro, S. and Jonsson, A. (2021). Fast stochastic kalman gradient descent for reinforcement learning. In *Learning for Dynamics and Control*, pages 1118–1129. PMLR.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008.

Vinyals, O., Babuschkin, I., Czarnecki, W. M., Mathieu, M., Dudzik, A., Chung, J., Choi, D. H., Powell, R., Ewalds, T., Georgiev, P., et al. (2019). Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575(7782):350–354.

Wang, J. X., Kurth-Nelson, Z., Soyer, H., Leibo, J. Z., Tirumala, D., Munos, R., Blundell, C., Kumaran, D., and Botvinick, M. M. (2016a). Learning to reinforcement learn. *ArXiv*, abs/1611.05763.

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

Wang, K., Kang, B., Shao, J., and Feng, J. (2020). Improving generalization in reinforcement learning with mixture regularization. *ArXiv*, abs/2010.10814.

Wang, Z., Merel, J., Reed, S. E., de Freitas, N., Wayne, G., and Heess, N. M. O. (2017). Robust imitation of diverse behaviors. In *NIPS*.

Wang, Z., Schaul, T., Hessel, M., Hasselt, H. V., Lanctot, M., and Freitas, N. D. (2016c). Dueling network architectures for deep reinforcement learning. *ArXiv*, abs/1511.06581.

White, A., Modayil, J., and Sutton, R. S. (2012). Scaling life-long off-policy learning. In *2012 IEEE International Conference on Development and Learning and Epigenetic Robotics (ICDL)*, pages 1–6. IEEE.

Whiteson, S., Tanner, B., Taylor, M. E., and Stone, P. (2011). Protecting against evaluation overfitting in empirical reinforcement learning. *2011 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL)*, pages 120–127.

Xu, Z., van Hasselt, H., and Silver, D. (2018). Meta-gradient reinforcement learning. In *NeurIPS*.

Yang, F., Yang, C., Guo, D., Liu, H., and Sun, F. (2020). Fault-aware robust control via adversarial reinforcement learning. *arXiv preprint arXiv:2011.08728*.

Yang, J., Petersen, B., Zha, H., and Faissol, D. (2019). Single episode policy transfer in reinforcement learning. *arXiv preprint arXiv:1910.07719*.

Yao, J., Killian, T. W., Konidaris, G., and Doshi-Velez, F. (2018). Direct policy transfer via hidden parameter markov decision processes.

Yarats, D., Zhang, A., Kostrikov, I., Amos, B., Pineau, J., and Fergus, R. (2019). Improving sample efficiency in model-free reinforcement learning from images. *ArXiv*, abs/1910.01741.

Ye, C., Khalifa, A., Bontrager, P., and Togelius, J. (2020). Rotation, translation, and cropping for zero-shot generalization. *arXiv preprint arXiv:2001.09908*.

Ye, M., Zhang, X., Yuen, P. C., and Chang, S.-F. (2019). Unsupervised embedding learning via invariant and spreading instance feature. *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 6203–6212.

Zhang, A., Ballas, N., and Pineau, J. (2018a). A dissection of overfitting and generalization in continuous reinforcement learning. *arXiv preprint arXiv:1806.07937*.

Zhang, A., Ballas, N., and Pineau, J. (2018b). A dissection of overfitting and generalization in continuous reinforcement learning. *ArXiv*, abs/1806.07937.

Zhang, A., Lyle, C., Sodhani, S., Filos, A., Kwiatkowska, M., Pineau, J., Gal, Y., and Precup, D. (2020a). Invariant causal prediction for block mdps. *arXiv preprint arXiv:2003.06016*.

Zhang, A., McAllister, R., Calandra, R., Gal, Y., and Levine, S. (2020b). Learning invariant representations for reinforcement learning without reconstruction.

Zhang, A., McAllister, R., Calandra, R., Gal, Y., and Levine, S. (2020c). Learning invariant representations for reinforcement learning without reconstruction. *arXiv preprint arXiv:2006.10742*.

Zhang, A., Satija, H., and Pineau, J. (2018c). Decoupling dynamics and reward for transfer learning. *ArXiv*, abs/1804.10689.

Zhang, C., Vinyals, O., Munos, R., and Bengio, S. (2018d). A study on overfitting in deep reinforcement learning. *arXiv preprint arXiv:1804.06893*.

Zhang, C., Vinyals, O., Munos, R., and Bengio, S. (2018e). A study on overfitting in deep reinforcement learning. *ArXiv*, abs/1804.06893.

Zhang, J., Springenberg, J. T., Boedecker, J., and Burgard, W. (2017). Deep reinforcement learning with successor features for navigation across similar environments. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2371–2378. IEEE.

Zhang, J., Wetzel, N., Dorka, N., Boedecker, J., and Burgard, W. (2019). Scheduled intrinsic drive: A hierarchical take on intrinsically motivated exploration. *arXiv preprint arXiv:1903.07400*.

Zhang, T., Xu, H., Wang, X., Wu, Y., Keutzer, K., Gonzalez, J. E., and Tian, Y. (2020d). Bebold: Exploration beyond the boundary of explored regions. *arXiv preprint arXiv:2012.08621*.

Zintgraf, L. M., Shiarlis, K., Kurin, V., Hofmann, K., and Whiteson, S. (2018). Fast context adaptation via meta-learning. In *ICML*.