Reproducible and Efficient Deep Reinforcement Learning

A Thesis
Submitted to the Faculty
of
Drexel University
by
Shengyi Huang
in partial fulfillment of the
requirements for the degree
of
Doctor of Philosophy
June 2023

Acknowledgments

My journey to achieving a Ph.D. has been a challenging and fun ride. To have been able to navigate through it and see the light at the other end of the tunnel, I am grateful to many individuals for their kind support along the way.

Foremost, I would like to express my sincere gratitude to my advisor, Santiago Ontañón. His mentorship has been invaluable, providing me with guidance and support throughout my research and career progression. My thanks also go to my dissertation committee members, Edward Kim, Vasilis Gkatzelis, Shahin Jabbari, and Simon Lucas.

To my friends and colleagues at the GAIMS lab, I offer my heartfelt thanks. Our lively discussions and collaborations have always been a source of enjoyment and inspiration. I would like to specifically mention David Grethlein and Zuozhi Yang, whose companionship has been truly valuable during this journey.

I would like to recognize Luladay Price, my mentor during my internship at Curai Health. This being my first industrial internship, I learned much about machine learning engineering and research in the industry. Additionally, I am grateful for my manager François Huet and colleagues Anitha Kannan, Li Deng, Luke Diliberto, Namit Katariya, Rhys Compton, and Vignesh Venkataraman.

My appreciation also extends to my mentor, Morgan McGuire, for his support during my time at Weights and Biases. There, I had the opportunity to explore and create videos on deep reinforcement learning topics. My colleagues Angelica Pan, Scott Condron, Ivan Goncharov, Jeremy Salwen, Cayla Sharp, Lavanya Shukla, and Aakarshan Chauhan all deserve my thanks. I am also grateful to Weights and Biases for providing a free academic license which has been instrumental in tracking my experiments.

At NVIDIA, I received immense support from my mentor, Markel Sanz Ausin, in learning to apply deep reinforcement learning to simulated robotics control in Isaac Gym. My sincere thanks go out to my manager Ashwath Aithal and the Isaac Gym Team Viktor Makoviychuk, Arthur Allshire, Aleksei Petrenko, and Gavriel State.

Furthermore, I extend my gratitude to Ben Kasper, my mentor at Riot Games. He has been instrumental in supporting me as I applied deep reinforcement learning to state-of-the-art games. He is always willing to provide detailed feedback on my work which has helped me learn to write better code. My thanks also go to my manager Albert Wang and my dedicated team members Phil Wardlaw, Tiffany Hwu, Wesley Kerr, Chase McDonald, Nate Tsang, and Ran Cao.

My work also greatly benefited from working with my collaborators. I would like to especially thank Rousslan Fernand Julien Dossa, Antonin Raffin, Anssi Kanervisto, Weixun Wang, Chang Ye, Jeff Braga, Dipam Chakraborty, Kinal Mehta, João G.M. Araújo, Chris Bamford, Lukasz Grela, Jiayi Weng, Min Lin, Rujikorn Charakorn, Zhongwen Xu, Bo Liu, Denys Makoviichuk, Viktor Makoviychuk, Zichen Liu, Shuicheng Yan.

At a personal level, my partner Siyi Lin has been a constant source of support. Her encouragement and belief in my abilities, particularly during challenging times, have greatly helped me overcome numerous difficulties in this journey.

Finally, my deepest gratitude is reserved for my parents. They have always been there for me, and I would not be able to be where I am today without their enduring love and unwavering support.

Table of Contents

List of Tables

List of Figures

Abstract

Reproducible and Efficient Deep Reinforcement Learning
Shengyi Huang
Santiago Ontañón, Ph.D.

Deep reinforcement learning (DRL), a paradigm by which agents learn how to do tasks through trial and error, has achieved great success in many domains. Researchers have successfully applied DRL to train autonomous agents that learn to play video games from pixels and control simulated robots, all the way up to design microchips. Despite these impressive accomplishments, DRL algorithms can be hard to reproduce due to their sensitivity to hyperparameters and seemingly unimportant implementation details. Additionally, running DRL algorithms can be computationally inefficient, especially in challenging domains such as real-time strategy (RTS) games which pose a significant challenge to DRL due to their large action space, sparse rewards, and partial observability. This thesis makes contributions toward making DRL more reproducible and efficient. First, we study how implementation details of DRL, often left out of academic publications, have a significant impact on algorithm behavior. We then propose a new framework to mitigate reproducibility issues, and this framework is encapsulated in a DRL library called `CleanRL`. We further identify and address reproducibility issues in distributed DRL with a new platform called Cleanba. Second, we build Gym-μRTS, an efficient RTS testbed for conducting DRL novel research topics, such as game presentation designs and efficient learning techniques for dealing with invalid actions and sparse rewards. We also propose methods to scale agents to perform unit-level control in RTS games, lifting the artificial action space restriction of past works.

Chapter 1: Introduction

Deep reinforcement learning (DRL), a paradigm by which agents learn how to do tasks through trial and error, has achieved great success in many domains. Researchers have successfully applied DRL to train autonomous agents that learn to play video games from pixels and control simulated robots, all the way up to designing microchips. Despite these impressive accomplishments, many open challenges still need to be addressed. First, the field of DRL suffers from a significant reproducibility problem. The DRL algorithms are usually brittle to hyperparameters and seemingly unimportant implementation details, which could lead to incorrect baselines and unverifiable results in new papers. Second, DRL algorithms can be quite computationally expensive and inefficient to run — Rainbow DQN takes more than 14,000 GPU hours to train agents in 57 Atari games. Third, scaling DRL to some domains, such as Real-time Strategy (RTS) games, remains challenging, primarily due to their large action space, sparse rewards, and partial observability. While the recent work AlphaStar impressively demonstrated a DRL agent that could defeat professional players in StarCraft (a popular RTS game), AlphaStar used 3072 TPUs and 50,400 CPUs for 44 days to train and poses artificial restrictions on the agent to share the human-style action space, making it impossible to issue different actions to different units simultaneously.

The contributions of this thesis consist of two prongs. The first part of this thesis addresses the reproducibility challenge. We study how implementation details of DRL, often left out of academic publications, have a significant impact on algorithm behavior. In particular, we identified 37 implementation details that are relevant to reproducing Proximal Policy Optimization (PPO)'s performance, but many of them are left out of the original paper. We then propose a new framework to mitigate reproducibility issues, and this framework is encapsulated in a DRL library called `CleanRL`. We also identify reproducibility issues in popular distributed DRL frameworks and address them with a new architecture called Cleanba that performs competitively with existing distributed DRL systems. The second part of this thesis addresses the efficiency challenge by making more efficient DRL testbeds and researching more efficient DRL techniques. We introduce Gym-μRTS, an efficient RL interface to the popular μRTS testbed for RTS research. By leveraging Gym-μRTS, we research game presentation designs and efficient learning techniques for dealing with invalid actions and sparse rewards. We also propose methods to scale agents to perform unit-level control, lifting the artificial action space restriction of AlphaStar.

1.1 Reinforcement Learning

Reinforcement learning (RL) is a learning framework through which an agent tries to learn specific tasks by trial-and-error interaction with the environment[2]. The idea is that the agent will *observe* the environment, *perform actions*, and *get reward signals* from the environment that suggest if the executed actions are good or bad. By repeating these three steps, the agent could improve its actions to maximize the reward signal via trial and error.

Consider an example where we try to teach a dog to shake hands with humans using reinforcement learning. The dog acts as an agent and perceives its surrounding environments and humans as the *observation*. The dog might observe that the human would hold a treat in their hand. By random chance, the dog may perform the *action* of trying to paw at the treat, and then the human would immediately *reward* the dog with a treat and shout "shake". By repeating this process, the dog learns to associate the sound of "shake" and the behavior of pawing at a human's hand with rewards; in the human's perspective, the dog learns to shake when the human says "shake."

While RL refers to the learning framework, the term is often used as a shorthand for RL algorithms, which would be considerably different. Using the aforementioned example, we are usually more interested in *how* the dog associates observations and actions with rewards, which is certainly

one of the dog's biological capabilities. More specifically, RL algorithms refer to computational algorithms that continuously take the environment interactions (e.g., observations, actions, rewards) as inputs and train an agent to output actions given observations. The two most common classes of RL algorithms are as below:

1. Policy iteration[2] (Chap. 4.3) — this type of algorithm lets the agent learn a policy directly by increasing the probability of actions that lead to a better sum of future rewards.

2. Value iteration[2] (Chap. 4.4) — this type of algorithm lets the agent learn the values of the states so that the agent can choose actions that lead to states with better values.

1.2 Deep Learning

Deep learning (DL) is a subfield of *machine learning (ML)* that is based on *artificial neural networks (ANNs)*. Inspired by prior findings in neuroscience that mental activities consist primarily of activities from networks of brain cells called *neurons*[3], ANNs aim to model how learning happens in the brain[4;5]. ANNs are composed of a mathematical model of neurons that, roughly speaking, "fires" when a linear combination of inputs and weights exceeds some threshold[4;6]. More formally, the neuron has some inputs x_i, weights w_i, biases b_i, and an activation function g. The neuron computation can be modeled as follows:

$$h = g\left(\sum_i x_i w_i + b_i\right)$$

Next, we can construct an ANN by connecting neurons. One of the most common ANN is a feed-forward network with only connections in one direction[4]. This means some previous neurons will generate outputs as inputs for following neurons, and that the outputs of the following neurons are never used as inputs for previous neurons. Feed-forward networks are usually organized by layers of nodes, and the number of nodes in the layers is called *hidden units*. The term "deep learning" originates from investigating neural networks with many more layers than traditionally used[5].

We can represent the feed-forward network as a function f with θ encoding its weights and biases that take in some input to produce an output y; the process is denoted as $f_\theta(x) = y$. f_θ can help approximate some arbitrary function f^*. Note that g is usually chosen to be a non-linear function such as the hyperbolic tangent tanh so that the networks can model some non-linear f^* such as the XOR function[5]. Given some dataset $\mathcal{D}$ that contains example inputs x_i and outputs $\hat{y}_i$ of f^*, we can draw data x_i from it and calculate some predicted y_i. We can use a loss function L to measure the prediction error $\sum_i L(y_i, \hat{y}_i)$. An example of the loss function is the *mean squared loss (MSE)* function which is defined as $L(x_i, y_i) = (x_i - y_i)^2$. Then we can use calculate the gradient of the prediction error w.r.t. the weights and biases θ using the backpropagation algorithm[7]. Doing gradient descent will produce a new set of weights and biases that will likely produce less prediction error, corresponding to the process in which the feed-forward networks learn.

DL has been on the rise for the past decade, permeating all kinds of applications. Incredible scientific breakthroughs have emerged, especially in computer vision[8], natural language processing[9], and reinforcement learning which we will discuss in the next section.

1.3 Deep Reinforcement Learning

Deep Reinforcement Learning (DRL) is an emerging field that augments RL with deep learning techniques. Researchers have extended the traditional RL algorithms such as Q-learning[10] (an example of value iteration methods) with deep neural networks, which produces an algorithm called *deep Q-learning (DQN)* that can learn to play Atari games from pixel inputs[11]. Atari's state space of pixels has a very large dimension. For example, if we represent the state as a grayscale image with resolution 84×84, and where each pixel can take 256 different values, then we would have $256^{(84*84)} \simeq 3.476 * 10^{16992}$ states. It is not practical to account for these many states in the

computer's memory, so functional approximations are often used[12]. DQN uses a deep convolutional neural network[13] to approximate the Q-value of the states and two additional techniques to stabilize training, which we will discuss more in Section 2.4.1. Mnih et al.[11] show that DQN can outperform the best linear learner in past work[12] in Atari games, and it can train agents to surpass human-level performance in more than half of the games.

DQN has been a pioneering work that inspired countless development in the field of DRL. Similarly, researchers have successfully extended policy gradient[14] algorithms (examples of policy iteration methods) with deep learning techniques to perform simulated robotics manipulations and Atari games[15;16].

1.4 Problem Statement

DRL is a fast-moving research area. Despite the intensive research in this field, many open challenges still need to be addressed. Below are some open problems motivating the work presented in this thesis:

1. **Reproducibility**: The field of DRL suffers a major reproducibility problem, which have resulted in unverifiable results and wasted effort. Specifically, this problem stems from the following three major causes.

   (a) **Brittleness.** DRL methods are brittle to hyperparameter settings. For example, the number of hidden neurons in the neural network (NN) architecture could hugely impact the performance and, even more strangely, increasing the size of the NN does not necessarily result in better performance[17]. Additionally, DRL methods can be unreliable across runs conducted with different random seeds[17;18].

   (b) **Unaccounted implementation details.** A few foundational DRL algorithms such as Trust Region Policy Optimization (TRPO)[19] and Proximal Policy Optimization (PPO)[16] have included seemingly small optimization techniques only found in their open source implementations but not described in the original paper. However, later works have found these optimization techniques can have a surprisingly large impact on the performance of the algorithms – they could stabilize the training[20], help the algorithm reach better empirical results[21], and offer utilities such as gracefully handling long-horizon environments[22].

   (c) **Under-controlled evaluation environments.** It is not uncommon for a simulation environment to evolve over time, fixing bugs and adding new features. For example, the MuJoCo Gym environments[16] are popular among researchers yet there are 4 versions (v1, v2, v3, v4). Sometimes the version difference could cause a series of reproducibility issues. For example, when we introduced the v4 environments in Gym, fixing a contact force bug actually resulted in poorer performance[1] and poses a reproducibility challenge.

2. **Efficiency**: While there has been tremendous progress in the field of DRL, efficiency remains an ongoing issue and training good agents in many types of games can still be quite computationally expensive. Consider the ALE as an example, one of the most popular testbed for DRL algorithms. A common evaluation criteria requires running experiments for a collection of 57 Atari games. On one hand, many DRL algorithms take a long wall-time to train. For example, Rainbow DQN[23] takes roughly 250 GPU hours per game, which sums up to GPU 14250 for the 57 Atari games[24]. On the other hand, other wall-lock-time-faster DRL algorithms are computationally / architecturally expensive. For instance, SEED RL's R2D2[25;26] uses 8 Tensor Processing Unit (TPU) v3 cores, 213 CPU actors for 40 hours per game. To make matters worse, the common evaluation criteria often requires researchers to run at least 3 random seeds to account for stochasticity of the environments and algorithms, so the computational budget is instantly tripled.

[1] See discussion at https://github.com/openai/gym/pull/2762#discussion_r853488897

In other types of DRL applications such as in Real-time Strategy games[27], the computational costs can be astronomical. Notably, Vinyals et al.[27] reported that AlphaStar has used 3072 TPU course and 50,400 preemptible CPU cores to do the training for a duration of 44 days, which approximately equates to 3.6 million dollars using public pricing on Google Cloud platform[2].

As a result, these algorithms and environments could be less undesirable because they cost an arm and leg for researchers without big research labs footing the bill. Besides, as suggested by Obando-Ceron and Castro[28], this practice is less inclusive to the broader range of researchers with less compute budget. Furthermore, this magnitude of computations also produces huge carbon footprint, which is bad for the environment. Last but not least, large computational requirement also makes it harder to debug experiments and iterate research ideas faster.

3. **Real-time Strategy Games**: RTS games pose a significant challenge for game Artificial Intelligence (AI)[29;30]. They are complex due to a variety of reasons: (1) players need to issue actions in real-time, which means agents have minimal time to produce what is the following action to execute, (2) most RTS games are partially observable, i.e.,, a player might not always able to observe the opponents' strategies and actions, (3) RTS games have huge action spaces. In the context of RL, additional challenges include (4) dealing with extremely sparse rewards, (5) designing efficient observation and action space representations[31], and (6) generalization across an unseen set of opponents and maps. The open problems of *sparse rewards* and *generalization* extend beyond RTS games:

   (a) **Sparse Rewards**: Some tasks have very sparse rewards. For example, consider the task of learning to win professional players in chess. Because of the unlikeliness of winning a game by chance during the early stages of training, it is improbable that the agent could even obtain the reward of winning via trial and error. As a result, the agent would only receive negative reward signals from which it can learn nothing.

   (b) **Generalization**: In the field of RL, it is common to evaluate agents on the environments they are trained on[32]. As a result, the agents usually overfit the training environment. For example, if the agent learns to play video games from looking at pixels, they could overfit the background features instead of the core entities[32]. In robotics tasks, the agent usually overfits the dynamics such as the fraction coefficient. Because of these overfitting behaviors, the agents often fail in environments they have never seen before.

1.5 Contributions

The contributions of this thesis consist of two prongs. **Part I** of this thesis addresses the reproducibility challenge by studying the importance of implementation details and proposing a new framework that promotes the reproducibility and transparency of DRL algorithms. In **Chapter 3**, we led the investigation of Proximal Policy Optimization (PPO)[16;33] and summarized 37 implementation details that are relevant to reproducing PPO's performance, whereas most of these details are not explicitly listed out in the original PPO paper. In **Chapter 4**, we introduce a new framework that puts tremendous emphasis on the reproducibility and transparency of implementation details of DRL algorithms. The framework is encapsulated in a library called `CleanRL`, which produces standalone and self-contained implementations of algorithm variants that are easier to understand, reproduce, and customize for research needs. In **Chapter 5**, we study reproducibility issues in the context of distributed DRL. We show that popular distributed DRL architecture such as IMPALA[34] can have reproducibility issues even if random seeds are controlled. We then propose a new distributed DRL architecture called Cleanba that is highly reproducible in different hardware configurations.

Part II of this thesis addresses the efficiency challenge by making more efficient DRL testbeds and researching more efficient DRL techniques. We introduce Gym-μRTS, an efficient RL interface

[2]see Appendix B.1 for calculation details

to the popular μRTS testbed for RTS research. Despite having a simplified game, μRTS still captures the core challenges of RTS games. In **Chapter 6**, we compare different observation and action space representations for μRTS. In **Chapter 7**, we look into invalid action masking, an optimization technique used in high-profile prior works but not described in detail. Our work provides a detailed description of how invalid action masking works, establishes a sound theoretical foundation, and presents empirical evidence to show this technique scales well. In **Chapter 8**, we propose a novel method called "action guidance" that can better leverage shaped rewards and sparse rewards at the same time to make optimizing against the real objective. In **Chapter 9**, we leverage our accumulated knowledge to scale our RL-based approach to the full-game mode of μRTS — that is, to control all the player-owned units simultaneously, which allows us to evaluate against bots in previous μRTS competition and discover novel strategies. Finally, we summarize the thesis and discuss future work in **Chapter 10**.

List of publications

Publications in international conferences with proceedings

- Shengyi Huang, Santiago Ontañón, Chris Bamford, and Lukasz Grela. Gym-μrts: Toward affordable full game real-time strategy games research with deep reinforcement learning. In *2021 IEEE Conference on Games (CoG)*, pages 1–8. IEEE, 2021
- Shengyi Huang, Rousslan Fernand Julien Dossa, Antonin Raffin, Anssi Kanervisto, and Weixun Wang. The 37 implementation details of proximal policy optimization. In *ICLR Blog Track*, 2022. URL https://iclr-blog-track.github.io/2022/03/25/ppo-implementation-details/
- Shengyi Huang and Santiago Ontañón. A closer look at invalid action masking in policy gradient algorithms. volume 35, May 2022. doi: 10.32473/flairs.v35i.130584. URL https://journals.flvc.org/FLAIRS/article/view/130584
- **(Not presented in this thesis)** Jiayi Weng, Min Lin, Shengyi Huang, Bo Liu, Denys Makoviichuk, Viktor Makoviychuk, Zichen Liu, Yufan Song, Ting Luo, Yukun Jiang, Zhongwen Xu, and Shuicheng YAN. Envpool: A highly parallel reinforcement learning environment execution engine. In *Thirty-sixth Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2022. URL https://openreview.net/forum?id=BubxnHpuMbG
- **(Not presented in this thesis)** Rhys Compton, Ilya Valmianski, Li Deng, Costa Huang, Namit Katariya, Xavier Amatriain, and Anitha Kannan. Medcod: A medically-accurate, emotive, diverse, and controllable dialog system. In *Machine Learning for Health*, pages 110–129. PMLR, 2021

Workshop presentations in international conferences or preprints

- Shengyi Huang and Santiago Ontañón. Comparing observation and action representations for deep reinforcement learning in μrts. *AIIDE Workshop on Artificial Intelligence for Strategy Games*, 2019
- **(Not presented in this thesis)** Chris Bamford, Shengyi Huang, and Simon Lucas. Griddly: A platform for ai research in games, 2020
- Shengyi Huang and Santiago Ontañón. Action guidance: Getting the best of sparse rewards and shaped rewards for real-time strategy games. *AIIDE Workshop on Artificial Intelligence for Strategy Games*, abs/2010.03956, 2020. URL https://arxiv.org/abs/2010.03956
- **(Not presented in this thesis)** Shengyi Huang, Anssi Kanervisto, Antonin Raffin, Weixun Wang, Santiago Ontañón, and Rousslan Fernand Julien Dossa. A2c is a special case of ppo, 2022

- **(Not presented in this thesis)** Chris Bamford, Shengyi Huang, and Simon Lucas. Griddly: A platform for ai research in games, 2020
- **(Not presented in this thesis)** Shengyi Huang and Santiago Ontañón. Measuring generalization of deep reinforcement learning with real-time strategy games. *AAAI Reinforcement Learning in Games Workshop*, 2021

Publications in international journals

- **(Not presented in this thesis)** Rousslan Fernand Julien Dossa, Shengyi Huang, Santiago Ontañón, and Takashi Matsubara. An empirical investigation of early stopping optimizations in proximal policy optimization. *IEEE Access*, 9:117981–117992, 2021
- Shengyi Huang, Rousslan Fernand Julien Dossa, Chang Ye, Jeff Braga, Dipam Chakraborty, Kinal Mehta, and João G.M. Araújo. Cleanrl: High-quality single-file implementations of deep reinforcement learning algorithms. *Journal of Machine Learning Research*, 23(274):1–18, 2022. URL http://jmlr.org/papers/v23/21-1342.html

Chapter 2: Background

This chapter goes over the background of the field. In section 2.1, we will first set up the RL problem more formally in the context of the Markov Decision Process and list out the notations and important concepts. Then, in section 2.2 we will discuss the objective of RL, which is usually to maximize the rewards the agent can get. In section 2.3, we list some popular RL research testbeds such as arcade games and robotics control suites.

Next, we will discuss and review popular RL algorithms, which are often classified into value-based and policy-based algorithms. We cover the value-based algorithms in section 2.4 and the policy-based algorithms in 2.5. In both of these sections, we discuss their basic concepts and discuss their recent extensions powered by the emergence of neural networks and deep learning.

2.1 Markov Decision Processes

Let us consider the RL problem in a *Markov Decision Process (MDP)*[45] which can help us model the sequential decision problem in which the agent continuously interacts with the environments. The agent and the environment interact with each other using discrete time steps $t = 0, 1, 2, 3, ..., T$. In this thesis, we will mainly consider the popular *episodic* MDP setting that is popular among DRL researchers[11;16]. Let us use $\mathbf{Pr}\{x_t = x\}$ to denote the probability of the random variable x_t taking on the value x. Such MDP setting is usually formulated as a tuple $(\mathcal{S}, \mathcal{A}, R, P, \rho_0, \gamma, T)$ of the following components:

- $\mathcal{S}$ is the state space.
- $\mathcal{A}$ is the action space.
- $R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the reward function.
- $P(s', r \mid s, a) : \mathcal{S} \times \mathcal{R} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ is the *dynamics* of the MDP, which specifies the probability of the environment entering s' and obtaining r if the environment was in s executing a. $\mathcal{R} \doteq \{R(s, a) \mid \forall s \in \mathcal{S}, \forall a \in \mathcal{A}\} \subseteq \mathbb{R}$ is the set of possible rewards.
- $\rho_0 : \mathcal{S} \to [0, 1]$ is the initial state distribution.
- $\gamma \in (0, 1)$ is the discount factor.
- $T \in \mathbb{Z}$ is the maximum episode length.

Under this setting, the agent-environment interactions naturally break into *episodes*. The episode starts at some initial state $s_0 \in S$. We use the notation $s_0 \sim \rho_0$ to denote s_0 is sampled from the distribution ρ_0. Then, at each time step t the agent takes an action $a_t \in \mathcal{A}$ according to its policy $\pi : S \times \mathcal{A} \to [0, 1]$ which assigns a probability to each possible action given a state. The environment returns a state $s_{t+1} \in S$ and a reward of the action r_t according to the MDP dynamics, defined as:

$$P(s', r \mid s, a) \doteq \mathbf{Pr}\{s_t = s', r_{t-1} = r \mid s_{t-1} = s, a_{t-1} = a\} \tag{2.1}$$

Eventually, the agent enters into a terminate state s_T. The agent-environment interactions yield a trajectory:

$$\tau = s_0, a_0, r_0, s_1, a_1, r_1, s_2, a_2, ..., s_{T-1}, a_{T-1}, r_{T-1}, s_T \tag{2.2}$$

We can measure the agent's performance by looking at the *return* of a trajectory at time step 0 G_0. A return G_t is the sum of future rewards starting at t:

$$G_t \doteq r_t + r_{t+1} + r_{t+2} + \cdots + r_{T-1}, \tag{2.3}$$

Sometimes rewards at different time steps could have different importance — we usually consider intermediate rewards to be more valuable. For example, a $100 bill today is more important to us than a $100 bill in 3 years because we could save the money in the bank to earn interest. One related concept is *discounting*. We introduce a discounting factor $0 \leq \gamma \leq 1$ and look at the *discounted return*:

$$G_t \doteq r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots + \gamma^{T-1} r_{T-1} = \sum_{t'=t}^{T-1} \gamma^{t'-t} r_{t'}, \tag{2.4}$$

The discounting factor could help the agent focus more on intermediate rewards instead of future rewards, which will be discounted exponentially when $\gamma < 1$. Since the *return* is a special case of *discounted return* when $\gamma = 1$, we will use G_t to denote discounted return unless otherwise specified.

Remark 2.1 (Discrepant Notations). The literature of RL sometimes uses discrepant sets of notations and can be confusing. One of the most common discrepancies is the notation of reward steps. Consider this question — if we have an initial observation s_0 and execute an action a_0 to obtain state s_1, should the reward be r_0 or r_1? In literature, we see both approaches. Schulman et al.[46] uses r_0 while Sutton and Barto[2]; Machado et al.[47] uses r_1. Other discrepancies include capitalization (e.g., P or p for the state transition probability function), setting (e.g., discounted MDP vs regular MDP), and others. In this thesis, we use the notations from Schulman[48].

Remark 2.2 (Markov Property). One important aspect of the MDP is the Markov property, which suggests that given a sequence of state random variables $\{s_0, s_1, ..., s_n\}$, $\mathbf{Pr}(s_{n+1}|s_n, s_{n-1}, ..., s_0) = \mathbf{Pr}(s_{n+1}|s_n)$. This property essentially says "the future is independent of the past and given the present." While many real-world problems do not satisfy the Markov property, we can often approximate the Markov property by somehow encoding information in past states. For example, Mnih et al.[11] stacks four past frames of the Atari games as the observation so that the observation contains some past information for the agent to infer a velocity and the direction of objects.

2.2 The Learning Problem

In the episodic MDP, the objective of RL usually is to train a policy that maximizes the expected discounted return. In a slight abuse of notation, let us use $G(\tau) \doteq G_0 = \left[\sum_{t=0}^{T-1} \gamma^t r_t\right]$ to denote the discounted return of a trajectory τ. Here we express the expected discounted return following a policy π:

$$\begin{gathered} J(\pi) = \mathbb{E}_\tau \left[G(\tau)\right] \\ \text{where } \tau \text{ is the trajectory } (s_0, a_0, r_0, \ldots, s_{T-1}, a_{T-1}, r_{T-1}, s_T) \\ \text{and } s_0 \sim \rho_0, s_t \sim P(\cdot|s_{t-1}, a_{t-1}), a_t \sim \pi_\theta(\cdot|s_t), r_t = r\left(s_t, a_t\right) \end{gathered} \tag{2.5}$$

Here, the expectation notation means $\mathbb{E}_x[f(x)] \doteq \int dx \, \mathbf{Pr}(x) f(x)$ — the subscript is the random variable we are calculating the expectation over. To expand the expectation in Equation 5.1, we have

$$\mathbb{E}_\tau \left[G(\tau)\right] = \int dx \, \mathbf{Pr}(\tau) G(\tau) \tag{2.6}$$

where $\mathbf{Pr}(\tau)$ can be calculated as follows:

$$\begin{aligned} \mathbf{Pr}(\tau) &= \mathbf{Pr}(s_0, a_0, r_0, \ldots, s_{T-1}, a_{T-1}, r_{T-1}, s_T) \\ &= \rho_0(s_0)\pi(a_0 \mid s_0)P(s_1, r_0 \mid s_0, a_0)\pi(a_1 \mid s_1)P(s_1 \mid s_0, a_0)\ldots \\ &\quad \pi(a_{T-1} \mid s_{T-1})P(s_T, r_{T-1} \mid a_{T-1}, s_{T-1}) \\ &= \rho_0(s_0) \prod_{t=0}^{T-1} \pi\left(a_t \mid s_t\right) P\left(s_{t+1}, r_t \mid s_t, a_t\right) \end{aligned} \tag{2.7}$$

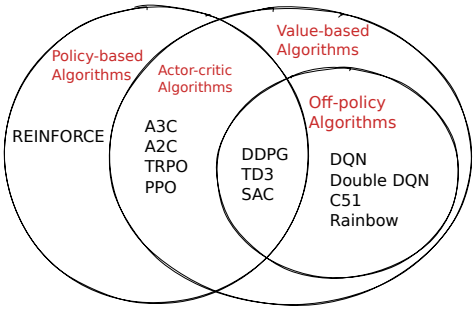


Figure 2.1: The taxonomy of popular DRL algorithms.

Let us denote Π the space of policies; the RL learning problem aims to find the optimal policy:

$$\pi_* = \arg\max_{\pi \in \Pi} J(\pi) \tag{2.8}$$

Namely, the policy π^* would obtain the best return in the expectation, which is dependent on π^*.

There are two primary ways to train the policy. First, we could learn a value function of the optimal policy that accurately determines how good the state the agent is in. Then the policy π could be trivial, always choosing an action that goes into a state of higher values. Second, we could learn a policy π directly to choose actions. The former approach is called *value-based methods* and the latter *policy-based methods*. The approaches that leverage both a policy and value function are called *actor-critic methods*. Furthermore, the policy used to generate rollout data is known as the *behavior policy*, and the policy RL algorithms aim to improve is called the *target policy*. The learning approaches that leverage the same behavior policy and target policy are known as *on-policy methods*, and the ones that use different behavior and target policies are called *off-policy methods*. Figure 2.1 shows the taxonomy of popular DRL algorithms.

2.3 Common RL Environments

A considerable amount of research progress in DRL has focused on testing and developing new DRL algorithms in games. Unlike real-world problems, games are *computationally cheap* to simulate and act as great testbeds to observe the learned behaviors of the agents. The two most common kinds of testbeds / environments / games are as follows. We also call them *benchmark environments* because we usually benchmark DRL algorithms on these environments to study the relative strengths and weaknesses of algorithms.

Arcade Learning Environment (ALE). The Arcade Learning Environment (ALE)[12] has been one of the most popular simulation environments for researchers to study DRL. ALE provides an easy-to-use interface to hundreds of Atari 2600 game environments, including Breakout, Space invader, Beam Rider, Seaquest, and others. For example, Figure 2.2 (left) shows the game Breakout in ALE. In Breakout, the agent observes the game's screen in pixels and controls the pedal to bounce a red ball into the bricks at the top. Every time the ball hits a brick, the agent receives a +1 reward. The sum of the rewards in an episode corresponds to the game scores in the top left corner. If the agent fails to bounce the falling ball up, the agent loses a life; when the agent exhausts five lives, the game ends.

MuJoCo Robotics Environment. MuJoCo is a physics engine designed to help study model-based control[49]. In recent years, MuJoCo has also become a popular testbed for studying robotics control with RL[15]. The MuJoCo Robotics Environment[16] is a series of robotics environments implemented and OpenAI Gym[50]. Schulman et al.[16] introduce seven robotics environments, which are HalfCheetah, Hopper, InvertedDoublePendulum, InvertedPendulum, Reacher, Swimmer, and Walker2d. For example, Figure 2.2 (right) shows the game HalfCheetah environment; the RL agent observes the positional values of the joints, followed by velocity values, and controls the joints of

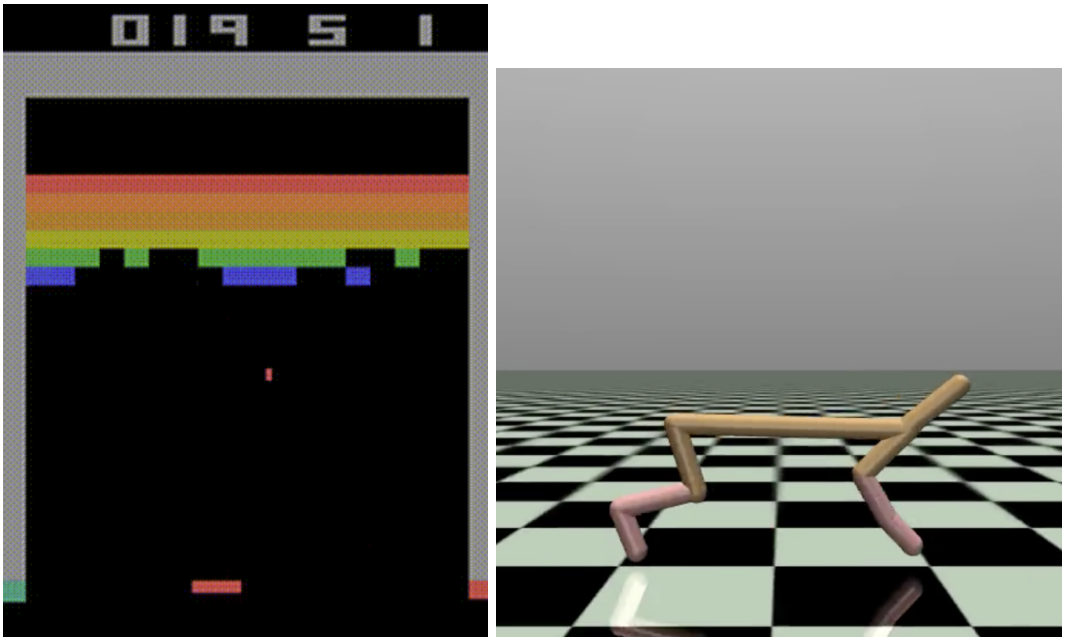


Figure 2.2: A screenshot of the game Breakout in the Arcade Learning Environment [12] (left figure) and a screenshot of the task HalfCheetah-v2 in the MuJoCo Robotics Environment [16;49].

the HalfCheetah to move. The agent gets rewards for not falling over and moving forward as far as possible.

These benchmark environments are usually available through a uniform software interface called "gym" [50].

2.4 Value-based Methods

One useful concept in optimizing for the best expected discounted return is the value functions, which would give a scalar estimate of a state, where the higher the scalar value, the better the state is. Here are five common value functions:

1. **Value Function**, which gives the expected return following π starting from state s:

$$V_\pi(s) \doteq \mathbb{E}_\tau \left[G_t \mid s_t = s \right] = \mathbb{E}_\tau \left[\sum_{t'=t}^{T-1} \gamma^{t'-t} r_{t'} \mid s_t = s \right], \text{ for all } s \in \mathcal{S} \tag{2.9}$$

2. **State-action Value Function**, also known as the **Q-function**, which gives the expected return following π starting from state s and executing action a:

$$Q_\pi(s,a) \doteq \mathbb{E}_\tau \left[G_t \mid s_t = s, a_t = a \right] = \mathbb{E}_\tau \left[\sum_{t'=t}^{T-1} \gamma^{t'-t} r_{t'} \mid s_t = s, a_t = a \right] \tag{2.10}$$

3. **Optimal Value Function**, which gives the expected return following an optimal policy π_* starting from state s:

$$V_*(s) \doteq \max_{\pi \in \Pi} V_\pi(s) \tag{2.11}$$

4. **Optimal State-action Value Function**, which gives the expected return following an opti-

mal policy π_* starting from state s and executing action a:

$$Q_*(s,a) \doteq \max_{\pi \in \Pi} Q_\pi(s,a) \tag{2.12}$$

5. **Advantage Function**, which gives the "advantage" of taking action a instead of following policy π[51]:

$$A_\pi^{\text{adv}}(s,a) \doteq Q_\pi(s,a) - V_\pi(s) \tag{2.13}$$

Many popular RL[10] and DRL[11] algorithms leverage the recursive nature of the Q-function:

$$Q_\pi(s,a) = \mathbb{E}_\tau \left[\sum_{t'=t}^{T-1} \gamma^{t'-t} r_{t'} \mid s_t = s, a_t = a \right] \tag{2.14}$$

$$= \mathbb{E}_\tau \left[r_t + \gamma Q_\pi(s_{t+1}, a_{t+1}) \right] \tag{2.15}$$

This recurrence equation is usually referred to as the *Bellman equation*[52], and we can construct a loss to minimize the difference between the left and right terms of the equation[11] to learn the Q-values of the states.

2.4.1 Deep Value-based Methods

Empowered by the wave of deep learning approaches, Mnih et al.[11] introduced Deep Q-learning (DQN), a DRL algorithm that empowers computer agents to learn how to play 56 Atari games in the ALE from scratch. On a high level, DQN is a significant milestone in the field because it makes RL work with visual inputs, and the agents learn to play most games to a reasonable degree. In comparison, most previous methods only deal with low-level representations of the game states. Additionally, DQN has served as a foundational baseline that researchers continue to extend to improve sample efficiency. These extended algorithms are known as the algorithms in the DQN family: Double DQN[53], Prioritized DQN[54], Distributional DQN[55], Rainbow[23].

To compare across different algorithms and Atari games, researchers use the median *human-normalized score (HNS)* as a common metric. The HNS in a particular Atari game X is calculated as follows:

$$HNS_X = \frac{\text{Agent's Episodic Return in } X - \text{Random Play's Episodic Return in } X}{\text{Professional Game Tester's Episodic Return in } X - \text{Random Play's Episodic Return in } X} \tag{2.16}$$

100% HNS at game X is usually referred to as "human-level performance"[11], which means the agent can play as well as a human in X, and more than 100% HNS is known as "super-human performance"[56]. Then, the median HNS of 57 Atari games usually serves as a good measure of performance when compared to humans.

> *Remark* 2.3 (Evaluation Metric). The average HNS is often not a preferable metric. This is because in some games the agent tends to far exceed the humans (e.g., 2539% HNS in Video Pinball) whereas in other games the agent learns nothing (e.g., 0% HNS in Montezuma's Revenge)[11].

A comparison of these algorithms can be found in Figure 2.3, which showcases the median *human-normalized score (HNS)* over 200 million frames of training. Figure 2.3 is also known as a learning curve, where the agent obtains incrementally larger median HNS as they get more training.

Deep Q-learning

As an extension of the Q-learning[10], DQN's main technical contribution is the replay buffer and target network, both of which improve the algorithm's stability. Replay buffer works by keeping track of historical transitions (i.e., s_t, a_t, r_t, s_{t+1}), and the agents only train on samples of the replay buffer. Because the replay buffer tracks many historical transitions, it improves stability by ensuring the agents do not catastrophically forget how to play the game in the beginning. Another benefit of

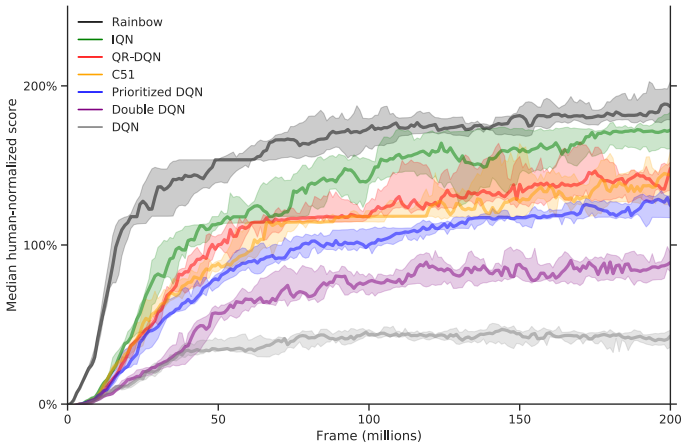


Figure 2.3: Median human-normalized performance across 57 Atari games of various learning methods, reproduced from the *deepmind/dqn_zoo* GitHub repository under the Apache2 License.

the replay buffer is a better utilization of the existing data, which further improves sample efficiency by doing more gradient updates. Furthermore, the target network intuitively makes sure the Q-target in the loss does not change drastically so that the loss would explode.

The rough idea of DQN is to approximate the optimal state-action value function Q_* by minimizing the mean square error between the Q-value at time step t and the next Q-value at time step $t+1$. Algorithm 1 shows a pseudocode for DQN. DQN stacks observations to help the agent infer the velocity and the direction of moving objects; let us use x_t to denote the raw observation, and we will stack multiple x_t as s_t to denote the stacked observations, which will be processed by some function φ

Double Deep Q-learning

One issue with DQN is that sometimes it learns unrealistically high action values. DQN has a maximization step over the estimated action values during training, which tends to overestimate instead of underestimating values[53]. To help mitigate this issue, van Hasselt et al.[53] propose Double DQN (DDQN). The main idea is in the Q-target (line 13 of Algorithm 1) $y_j = r_j + \gamma \max_{a'} Q_{\phi^-}(\varphi_{j+1}, a')$ the same Q-network is used for both the selection and evaluation of the action, so the authors suggest replacing the Q-target with

$$y_j = r_j + \gamma Q_{\phi^-}(\varphi_{j+1}, \arg\max_{a'} Q_\phi(\varphi_{j+1}, a')),$$

so that the action selection and uses the Q-network with weight ϕ, and the action evaluation uses the secondary Q-network with weights ϕ^-. van Hasselt et al.[53] empirically show that DDQN improves over DQN in value accuracy and agent's policy, achieving better sample efficiency.

Prioritized Experience Replay

Another improvement on DQN is the use of prioritized experience replay (PER)[54], which helps to sample and learn from more surprising experiences to the agent. DQN samples the experience uniformly as indicated by line 11 of Algorithm 1, which may be inefficient because some experience sampled might be redundant and unhelpful.

line 11: Sample random minibatch of transitions $(\varphi_j, a_j, r_j, \varphi_{j+1})$ from $\mathcal{D}$

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q_ϕ with random weights ϕ
Initialize target action-value function Q_{ϕ^-} with random weights ϕ^-
for episode $= 1, M$ **do**
    Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\varphi_1 = \varphi(s_1)$
    for $t = 1, T$ **do**
        With probability ϵ select a random action a_t
        otherwise select $a_t = \max_a Q_\phi(\varphi(s_t), a)$
        Execute action a_t in emulator and observe reward r_t and image x_{t+1}
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \varphi(s_{t+1})$
        Store transition $(\varphi_t, a_t, r_t, \varphi_{t+1})$ in $\mathcal{D}$
        Sample random minibatch of transitions $(\varphi_j, a_j, r_j, \varphi_{j+1})$ from $\mathcal{D}$
        Set $y_j = \begin{cases} r_j & \text{for terminal } \varphi_{j+1} \\ r_j + \gamma \max_{a'} Q_{\phi^-}(\varphi_{j+1}, a') & \text{for non-terminal } \varphi_{j+1} \end{cases}$
        Minimize $\left(y_j - Q_\phi(\varphi_j, a_j; \theta)\right)^2$ via gradient descent
        Every C steps reset $Q_{\phi^-} = Q_\phi$

Schaul et al.[54] propose PER to sample the experiences that have caused high TD-errors with priority to mitigate the issue. Focusing on these surprising experiences is shown to help the agents to learn more efficiently using existing data with DQN and Double DQN.

Distributional / Categorical DQN (C51)

Instead of predicting a *scalar value* for each action like in DQN, Bellemare et al.[55] propose predicting each action's *value distributions*. The authors approximate the value distribution via a discrete distribution whose support is a set of, by default, 51 atoms (hence the name C51). These 51 atoms are evenly spaced numbers over a defined interval from $V_{MIN} \in \mathbb{R}$ to $V_{MAX} \in \mathbb{R}$. The idea is to have the agent output the probability mass function of the discrete distribution and use a distributional bellman operator to construct a Q-target for the training. C51 better utilizes the data by predicting value distributions, achieving higher sample efficiency than DQN, DDQN, DDQN with PER, and Dueling DQN with PER[57].

Rainbow

Rainbow[23] is an algorithm that combines all the extensions of DQN presented above and two other extensions: Dueling DQN[57] and Noisy Net[58]. Because the extensions solve different problems, they are largely complementary to each other. Because of this, Rainbow can combine the benefits of all these extensions and produce state-of-the-art sample efficiency in the ALE.

2.5 Policy-based Methods

Alternatively, we could also parameterize a policy directly. Let us use π_θ to denote the policy π is parameterized by some vector θ. We can then obtain the *policy gradient* of the expected discounted return w.r.t. the policy parameter θ:

$$\nabla_\theta J(\pi_\theta) = \nabla_\theta \mathbb{E}_\tau \left[G(\tau)\right] = \nabla_\theta \mathbb{E}_\tau \left[\sum_{t=0}^{T-1} \gamma^t r_t\right] \tag{2.17}$$

Doing gradient ascent $\theta = \theta + \nabla_\theta J(\pi_\theta)$ therefore maximizes the expected discounted reward. We can derive $\nabla_\theta J(\pi_\theta)$ as follows[48]:

$$\begin{aligned}
\nabla_\theta J(\pi_\theta) &= \mathbb{E}_\tau \left[G(\tau) \right] \\
&= \nabla_\theta \int dx \, \mathbf{Pr}(\tau) G(\tau) \\
&= \int dx \, \nabla_\theta \mathbf{Pr}(\tau) G(\tau) \\
&= \int dx \, \nabla_\theta \mathbf{Pr}(\tau) \frac{\mathbf{Pr}(\tau)}{\mathbf{Pr}(\tau)} G(\tau) \\
&= \int dx \, \mathbf{Pr}(\tau) \frac{\nabla_\theta \mathbf{Pr}(\tau)}{\mathbf{Pr}(\tau)} G(\tau) \\
&= \int dx \, \mathbf{Pr}(\tau) \nabla_\theta \log \mathbf{Pr}(\tau) G(\tau) \quad \text{because} \frac{\partial}{\partial x} \log f(x) = \frac{f'(x)}{f(x)} \\
&= \mathbb{E}_\tau \left[\nabla_\theta \log \mathbf{Pr}(\tau) G(\tau) \right]
\end{aligned} \tag{2.18}$$

We can calculate $\nabla_\theta \log \mathbf{Pr}(\tau)$ as follows:

$$\begin{aligned}
\log \mathbf{Pr}(\tau) &= \log \left(\rho_0(s_0) \prod_{t=0}^{T-1} \pi_\theta \left(a_t \mid s_t \right) P \left(s_{t+1}, r_t \mid s_t, a_t \right) \right) \quad \text{Equation 2.7} \\
&= \log \rho_0(s_0) + \sum_{t=0}^{T-1} \log \left(\pi_\theta \left(a_t \mid s_t \right) + \log P \left(s_{t+1}, r_t \mid s_t, a_t \right) \right) \\
\nabla_\theta \log \mathbf{Pr}(\tau) &= \cancel{\nabla_\theta \log \rho_0(s_0)} + \sum_{t=0}^{T-1} \left(\nabla_\theta \log \pi_\theta \left(a_t \mid s_t \right) + \cancel{\nabla_\theta \log P \left(s_{t+1}, r_t \mid s_t, a_t \right)} \right) \\
&= \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta \left(a_t \mid s_t \right)
\end{aligned} \tag{2.19}$$

This means the policy gradient can be written as follows:

$$\nabla_\theta J(\pi_\theta) = \nabla_\theta \mathbb{E}_\tau \left[G(\tau) \right] = \mathbb{E}_\tau \left[\nabla_\theta \log \mathbf{Pr}(\tau) G(\tau) \right] = \mathbb{E}_\tau \left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta \left(a_t \mid s_t \right) G(\tau) \right] \tag{2.20}$$

It is perhaps surprising that the policy gradient *does not* depend on the distribution of the system dynamics p. Instead, it only depends on the probability of taking action a_t given the state s_t and the sample return. The intuition of the derived policy gradient is to *scale the likelihood of taking an action proportionate to the return of the trajectory* (e.g., the higher the return of the trajectory, the more likely the action will be taken after applying the gradient update).

We can further derive a version of the policy gradient to reduce variance[48]. Let us apply the previous policy gradient on a step of reward r_t:

$$\nabla_\theta \mathbb{E}_\tau \left[r_t \right] = \mathbb{E}_\tau \left[\sum_{t'=0}^{t} \nabla_\theta \log \pi_\theta \left(a_{t'} \mid s_{t'} \right) \gamma^t r_t \right]$$

Then we can sum all the steps of r_t to recover the policy gradient:

$$\nabla_\theta \mathbb{E}_\tau \left[G(\tau)\right] = \mathbb{E}_\tau \left[\sum_{t=0}^{T-1} \gamma^t r_t \sum_{t'=0}^{t} \nabla_\theta \log \pi_\theta \left(a_{t'} \mid s_{t'}\right)\right] \tag{2.21}$$

$$= \left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta \left(a_t \mid s_t\right) \gamma^t \sum_{t'=t}^{T-1} \gamma^{t'-t} r_{t'}\right] \tag{2.22}$$

$$= \left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta \left(a_t \mid s_t\right) \gamma^t G_t\right] \tag{2.23}$$

The derivation from the first line to the second line is not obvious but becomes clearer when considering an example. Let us assume $\tau = s_0, a_0, r_0, s_1, a_1, r_1, s_2, a_2, r_2, s_3$. Then we have:

$$\begin{aligned}
\nabla_\theta \mathbb{E}_\tau \left[G(\tau)\right] &= r_0 \nabla_\theta \log \pi_\theta \left(a_0 \mid s_0\right) \quad \text{(Equation 2.21)} \\
&+ \gamma r_1 (\nabla_\theta \log \pi_\theta \left(a_0 \mid s_0\right) + \nabla_\theta \log \pi_\theta \left(a_1 \mid s_1\right)) \\
&+ \gamma^2 r_2 (\nabla_\theta \log \pi_\theta \left(a_0 \mid s_0\right) + \nabla_\theta \log \pi_\theta \left(a_1 \mid s_1\right) + \nabla_\theta \log \pi_\theta \left(a_2 \mid s_2\right)) \\
&= r_0 \nabla_\theta \log \pi_\theta \left(a_0 \mid s_0\right) \\
&+ \gamma r_1 \nabla_\theta \log \pi_\theta \left(a_0 \mid s_0\right) + \gamma r_1 \nabla_\theta \log \pi_\theta \left(a_1 \mid s_1\right) \\
&+ \gamma^2 r_2 \nabla_\theta \log \pi_\theta \left(a_0 \mid s_0\right) + \gamma^2 r_2 \nabla_\theta \log \pi_\theta \left(a_1 \mid s_1\right) + \gamma^2 r_2 \nabla_\theta \log \pi_\theta \left(a_2 \mid s_2\right) \\
&= \nabla_\theta \log \pi_\theta \left(a_0 \mid s_0\right) \left(r_0 + \gamma r_1 + \gamma^2 r_2\right) \quad \text{(Equation 2.23)} \\
&+ \nabla_\theta \log \pi_\theta \left(a_1 \mid s_1\right) \gamma (r_1 + \gamma r_2) \\
&+ \nabla_\theta \log \pi_\theta \left(a_2 \mid s_2\right) \gamma^2 (r_2)
\end{aligned}$$

Notably, the policy gradient we just derived scales the likelihood of taking an action a_t proportionate to the *discounted return* a_t *gathered from the state* s_t. There is a subtle difference: Equation 2.20 scales $\log \pi_\theta \left(a_t \mid s_t\right)$ by the outcome of the *entire* trajectory, whereas Equation 2.23 scales $\log \pi_\theta \left(a_t \mid s_t\right)$ by the outcome of a_t. Intuitively, Equation 2.23 is better because we should let the agent understand the consequence G_t of its actions a_t and do not let the past consequence $r_0 + r_1 + \cdots + r_{t-1}$ impacts the agent's learning[59].

Remark 2.4 (Is policy gradient a gradient?). In practice, almost all the DRL algorithms, such as A3C[15], PPO[16], ACKTR[60], use the following gradient without the γ^t likely for implementation simplicity:

$$\nabla_? J(\pi_\theta) \left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta \left(a_t \mid s_t\right) \cancel{\gamma^t} \sum_{k=0}^{T-1} \gamma^k r_{t+k}\right] \tag{2.24}$$

The gradient above is only a policy gradient when $\gamma = 1$ and is *not* a valid policy gradient for $\gamma < 1$[61;62]. As a result, almost all deep policy gradient algorithms use the *wrong* gradient to optimize for the discounted objective. Interestingly, Zhang et al.[62] finds no sufficient evidence that the correct policy gradient always leads to better empirical performance. In this thesis, we point out this important issue, but we shall assume $\nabla_? J(\pi_\theta)$ is a good enough empirical approximation and denote $\nabla_? J(\pi_\theta)$ as $\nabla_\theta J(\pi_\theta)$ from here on.

We can leverage the $\nabla_\theta J(\pi_\theta)$ in Equation 2.24 to design a practical algorithm. Algorithm 2 shows the pseudocode of a variant of REINFORCE[63], a popular on-policy gradient algorithm. The general idea of the algorithm is to collect some trajectories based on the current policy and then update the policy according to Equation 2.24.

However, Algorithm 2 may be of high variance because the value of G_t could vary significantly depending on the reward's scale[2] (p. 329). To reduce the variance, a popular idea is to increase the likelihood of an action if it results in a *better-than-expected* return. We would use the advantage

Algorithm 2 A Variant of REINFORCE

Initialize a policy π_θ
Initialize a learning rate α
for *iteration* = 0,1,2,..., I **do**
    Collect a set of trajectories $\mathcal{D} = \{\tau_i\}$ following π_θ
    Calculate the returns $G_t^i = \sum_{t'=t}^{T-1} \gamma^{t'-t} r_{t'}^i$ for each τ_i
    Update the policy: $\theta = \theta + \alpha \frac{1}{|\mathcal{D}_k|(T-1)} \sum_{\tau_i \in \mathcal{D}} \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(s_t^i, a_t^i) G_t^i$
G_t^i, s_t^i, a_t^i, and r_t^i are the return, state, action, and reward at time step t and trajectory τ_i

Algorithm 3 A Variant of REINFORCE with advantage

Initialize a policy π_θ
Initialize a value V_ϕ
Initialize a learning rate α
for *iteration* = 0,1,2,..., I **do**
    Collect a set of trajectories $\mathcal{D} = \{\tau_i\}$ following π_θ
    Calculate the returns $G_t^i = \sum_{t'=t}^{T-1} \gamma^{t'-t} r_{t'}^i$ for each τ_i
    Update the policy: $\theta = \theta + \alpha \frac{1}{|\mathcal{D}_k|(T-1)} \sum_{\tau_i \in \mathcal{D}} \sum_{t=0}^{(T-1)} \nabla_\theta \log \pi_\theta(s_t^i, a_t^i)(G_t^i - V_\phi(S_t^i))$
    Update the value: $\phi = \phi + \alpha \frac{1}{|\mathcal{D}_k|(T-1)} \sum_{\tau_i \in \mathcal{D}} \sum_{t=0}^{(T-1)} (G_t^i - V_\phi(s_t^i)) \nabla_\phi V_\phi(s_t^i)$
G_t^i, s_t^i, a_t^i, and r_t^i are the return, state, action, and reward at time step t and trajectory τ_i

function A_π^{adv} in place of G_t, and the advantage function can be approximated via $G_t - V_\pi(S_t)$, which results in the following gradient

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_\tau \left[\nabla_\theta \log \pi_\theta(s_t, a_t)(G_t - V_\pi(s_t)) \right]$$

The value function V_π can be parameterized by $\phi \in \mathbb{R}^d$ (denoted as V_ϕ) and can be fitted with an MSE loss against the empirical returns. Algorithm 3 demonstrates the resulting algorithm.

2.5.1 Deep Actor-critic Methods

While the Deep Q-learning family obtains popularity, researchers also extend policy gradient methods to incorporate neural networks. The first such work is the Asynchronous Advantage Actor-Critic (A3C)[15], which continues to work on the ALE and extends to robotics environments with *continuous action spaces* using the MuJoCo simulator. Following that, researchers continue to improve sample efficiency with newer policy gradient methods such as Trust Region Policy Optimization (TRPO)[19], Proximal Policy Optimization (PPO)[16], Actor-Critic using Kronecker-Factored Trust Region (ACKTR)[60], Actor-Critic with Experience Replay(ACER)[64].

Asynchronous Advantage Actor-Critic.

Asynchronous Advantage Actor-Critic (A3C)[15] is basically Algorithm 3 but does rollouts and training asynchronously in separate workers. A3C is a foundational work that has two main contributions.

1. **Run faster while obtaining good performance.** One of the main benefits of DQN's experience replay is more diverse experiences during training, which help agents review skills they have already learned. Asynchronous Advantage Actor-Critic (A3C) provides a new paradigm to achieve the same purpose by having the agent collect experiences from multiple parallel environments that run on separate CPUs. Because of this, A3C achieves the same level of sample efficiency while taking significantly less wall time. As shown in the following Table, A3C achieves good performance taking as little as one day. In contrast, the DQN family has to take at least eight days to achieve the same level of performance (measured in mean

and median human-normalized scores on 57 Atari games using the human starts evaluation metric[15]).

2. **Work with continuous action spaces.** Unlike the DQN family, which only works with discrete action spaces, A3C also works with continuous action spaces that describe robotics control. A3C achieves this by modeling the discrete action spaces with a categorical distribution and continuous action spaces with a normal distribution.

Because of these two benefits over DQN, A3C and its synchronous variant A2C have become a great baseline that works with various games and tasks, gradually becoming a popular choice among researchers.

Trust Region Policy Optimization

Note that A3C only does a *single policy update* per rollouts collected, which can be wasteful. To further improve sample efficiency, Schulman et al.[19] propose Trust Region Policy Optimization (TRPO) to reuse the rollouts for *multiple policy updates* via importance sampling. To improve the stability of TRPO, the authors also propose constraining the amount of policy change (i.e., ensuring the trust region) via Kullback–Leibler divergence. TRPO shows robust performance in the MuJoCo robotics environments and in the ALE.

Proximal Policy Optimization

The practical algorithm to solve TRPO's constrained optimization problem involves natural gradient descent, which could be computationally expensive and challenging to implement[20]. To alleviate the complexity of TRPO, Schulman et al.[16] propose the Proximal Policy Optimization (PPO) algorithm with a simpler objective and implementation. Since its introduction in 2017, PPO has become arguably the most popular and essential DRL algorithm. It inherits the two main benefits of A3C of being reasonably fast and working with different action paces and further improves the algorithm's stability and performance. Researchers have used it as a go-to algorithm for relatively simple environments such as the ALE, MuJoCo robotics environment[16], and Google Football[65], all the way up to complicated multiplayer games such as Dota 2[66] and StarCraft II[67] and impactful real-life projects such as designing floor plans for microchips[68]. Here is its policy objective.

$$J^{CLIP}(\pi_\theta) = \mathbb{E}_\tau \left[\min \left(\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta'}(a_t \mid s_t)} \hat{A}^{\text{adv}}_{\pi}(s_t, a_t), \text{clip} \left(\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta'}(a_t \mid s_t)}, 1-\epsilon, 1+\epsilon \right) \hat{A}^{\text{adv}}_{\pi}(s_t, a_t) \right) \right] \tag{2.25}$$

where $\hat{A}^{\text{adv}}_{\pi}$ is an advantage estimator called Generalized Advantage Estimator[46]. During the optimization phase, the agent also learns the value function and maximizes the policy's entropy, therefore optimizing the following joint objective:

$$J^{JOINT}(\theta) = J^{CLIP}(\pi_\theta) - c_1 J^{VF}(\theta) + c_2 S[\pi_\theta], \tag{2.26}$$

where c_1, c_2 are coefficients, S is an entropy bonus, and J^{VF} is the squared error loss for the value function associated with π_θ.

Others

In addition to A3C and PPO, other notable mentions are Actor-Critic using Kronecker-Factored Trust Region (ACKTR)[60], Actor-Critic with Experience Replay(ACER)[64]. ACER improves the sample efficiency and stability by utilizing the replay buffer like in DQN, whereas ACKTR explores alternative trust-region optimization paradigms that are more scalable. Both methods show sizable improvement over the A3C baseline. However, PPO remains the most popular policy gradient algorithm due to its relative simplicity.

2.5.2 Deep Off-policy Actor-critic Methods

One of the drawbacks of DQN is the lack of support for continuous action spaces. To address this issue, Lillicrap et al.[69] modify DQN by breaking the Q-network into two separate networks, the actor and critic network, respectively, which results in the Deep Deterministic Policy Gradient algorithm that works with the continuous action spaces. Following this, Fujimoto et al.[70] propose improving the algorithm's stability and performance with some techniques borrowed from the DQN family.

Deep Deterministic Policy Gradient

The main problem with applying DQN to continuous action spaces is finding the action that maximizes the action value. The continuous action space requires iterative optimization and every step. Alternatively, it is also viable to discretize the continuous action space, but this approach has many drawbacks, mostly due to the dimensionality explosion. In DQN, its network takes the input of the state and outputs the action values of all the discrete actions. In Deep Deterministic Policy Gradient (DDPG)[69], its critic's network Q takes the input of a state-action pair and outputs the action value of the pair. Additionally, DDPG's actor's network μ takes the input of the state and outputs the action. During training, the critic's network can still be trained like in DQN:

$$\min \left(y_j - Q_\phi(\varphi_j, a_j)\right)^2 \text{ w.r.t } \phi, \quad y_j = r_j + \gamma Q_\phi(\varphi_{j+1}, \mu_\theta(\varphi_{j+1}))$$

where ϕ and θ are the weights of the critic's and actor's network, respectively. Moreover, the actor can be trained by simply maximizing the Q-value:

$$\max Q_\phi(\varphi_{j+1}, \mu_\theta(\varphi_{j+1})) \text{ w.r.t } \theta$$

DDPG have achieved good baseline performance in the MuJoCo environments.

Twin Delayed Deep Deterministic Policy Gradient

Twin Delayed Deep Deterministic Policy Gradient (TD3)[70] is an algorithm that improves upon DDPG. TD3 features three contributions:

1. **Clipped Double Q-learning.** Like DQN, DDPG suffers from over-estimation of the Q values. To mitigate this issue, the authors adopt ideas from Double Q-learning and propose Clipped Double Q-learning (CDQ) for training the critic's network. CDQ can significantly reduce over-estimation empirically. Although it is possible to underestimate Q values with CDQ's design, it is preferable to overestimate Q values since the errors will not be propagated through updates.

2. **Target Policy Smoothing Regularization.** When calculating the Q-target, Target Policy Smoothing Regularization (TPS) adds small amounts of clipped random noise to actions sampled from μ, which results in that similar actions should have similar values.

3. **Delayed Policy Updates.** Because there are two separate networks for the actor and critic, it is beneficial to update the actor's network only after the critic's network has been stabilized. To this end, TD3 delays the policy updates until the critic's network has been updated $k = 2$ times.

By combining these three contributions and tuning the architecture and hyperparameters, TD3 achieves state-of-the-art sample efficiency in the MuJoCo environments.

2.6 Acknolwedgement

This chapter has referenced the notations and preliminaries sections from the papers, book, or thesiss of Sutton and Barto[2]; Machado et al.[47]; Schulman[48]; Hoffman et al.[51]; Flet-Berliac[71].

Part I

Reproducibility in Deep Reinforcement Learning

Chapter 3: Demystifying PPO

As discussed in Section 1.4, the field of DRL suffers a major reproducibility problem partly because the implementation details are overlooked. In this chapter, we closely examine the implementation details of Proximal Policy Optimization (PPO), one of the most popular DRL algorithms. Despite its widespread popularity, PPO is *not* a peer-reviewed publication and has undergone several versions. As a result, understanding and reproducing PPO has been a recurrent issue for years. To address these issues, we take a deep dive into PPO's official implementation and enumerate 37 implementation details that are important to replicate PPO's performance, yet most of these details are not explicitly elaborated in the original PPO paper. By understanding these 37 details, we empirically show that we can reproduce PPO's performance with high fidelity in the ALE, MuJoCo Robotics Environment, Procgen, and other benchmark tasks. The work of this chapter is based on the following publication:

- Shengyi Huang, Rousslan Fernand Julien Dossa, Antonin Raffin, Anssi Kanervisto, and Weixun Wang. The 37 implementation details of proximal policy optimization. In *ICLR Blog Track*, 2022. URL https://iclr-blog-track.github.io/2022/03/25/ppo-implementation-details/

3.1 Authorship

Shengyi Huang created the codebase and designed experiments. Rousslan Fernand Julien Dossa helped with the codebase. Shengyi Huang led the writing, with contributions from Rousslan Fernand Julien Dossa, Antonin Raffin, Anssi Kanervisto, and Weixun Wang.

3.2 Motivation

PPO is a policy gradient algorithm proposed by Schulman et al.[16]. As a refinement to Trust Region Policy Optimization (TRPO)[19], PPO uses a simpler clipped surrogate objective, omitting the expensive second-order optimization presented in TRPO. Despite this simpler objective, Schulman et al.[16] show PPO has higher sample efficiency than TRPO in many control tasks. PPO also exhibits good empirical performance in the arcade learning environment (ALE). Since then, PPO has become one of the most popular DRL algorithms, accumulating over 8,000 citations at the time of this writing. Its applications are diverse — examples include video games[16;66], robotics control[72], Natural Language Processing (NLP)[73;74], database optimizations[75], and circuit and chip designs[76;77].

Despite PPO's widespread adoption, reproducing PPO's results can be surprisingly challenging. The original PPO codebase is written in Tensorflow 1.x and has arguably poor organization, resulting in hard-to-read code. As a result, researchers have often elected to homebrew their own PPO implementation given that the PPO paper[16] is relatively straightforward, or at least it seems so. To many RL practitioners' surprise, they found themselves struggling with the same problem: *"why does my PPO perform worse than the original PPO?"* When seeking help online, these practitioners usually find a series of mysterious words, such as "gradient clipping," which are never mentioned in the PPO paper.

However, there is no magic in the world: if the original PPO performs well and home-brewed PPOs do not, there must be some differences. Indeed, recent work has identified 9 code-level optimizations implemented in the original PPO codebase but barely mentioned in the paper[20]; researchers also conducted extensive ablation studies on 50+ design choices with PPO[21]. Despite their contribution, these works do not describe how all these implementation details come together

[0]See https://github.com/openai/baselines

as a whole picture. As a result, PPO implementation remains mysterious, especially to beginners. In this chapter, we describe our effort in demystifying PPO by explaining its implementation details in significant depth.

3.3 Background

PPO is a policy gradient algorithm proposed by Schulman et al.[16]. As a refinement to Trust Region Policy Optimization (TRPO)[19], PPO uses a simpler clipped surrogate objective, omitting the expensive second-order optimization presented in TRPO. Despite this simpler objective, Schulman et al.[16] demonstrates that PPO has higher sample efficiency than TRPO in many control tasks. PPO also has good empirical performance in the arcade learning environment (ALE) which contain Atari games.

To facilitate more transparent research, Schulman et al.[16] have made the source code of PPO available in the `openai/baselines` GitHub repository with the code name `pposgd` (commit da99706 on 7/20/2017). Later, the `openai/baselines` maintainers have introduced a series of revisions. The key events include:

1. 11/16/2017, commit 2dd7d30: the maintainers introduced a refactored version `ppo2` and renamed `pposgd` to `ppo1`. According to a GitHub issue, one maintainer suggests `ppo2` should offer better GPU utilization by batching observations from multiple simulation environments.
2. 8/10/2018, commit ea68f3b: after a few revisions, the maintainers evaluated `ppo2`, producing the MuJoCo benchmark
3. 10/4/2018, commit 7bfbcf1: after a few revisions, the maintainers evaluated `ppo2`, producing the Atari benchmark
4. 1/31/2020, commit ea25b9e: the maintainers have merged the last commit to `openai/baselines` to date. To our knowledge, `ppo2` (ea25b9e) is the base of many PPO-related resources:
   (a) RL libraries such Stable-Baselines3 (SB3), pytorch-a2c-ppo-acktr-gail, and CleanRL have built their PPO implementation to match implementation details in `ppo2` (ea25b9e) closely.
   (b) Recent papers[20;21] have examined implementation details concerning robotics tasks in `ppo2` (ea25b9e).

In recent years, reproducing PPO's results has become a challenging issue. Table 3.2 collects the best-reported performance of PPO in popular RL libraries in Atari and MuJoCo environments.

We offer several observations.

1. These revisions in `openai/baselines` are not without performance consequences. Reproducing PPO's results is challenging partly because *even the original implementation could produce inconsistent results*.
2. `ppo2` (ea25b9e) and libraries matching its implementation details have reported rather similar results. In comparison, other libraries have usually reported more diverse results.
3. Interestingly, we have found many libraries reported performance in MuJoCo tasks but not in Atari tasks.

Despite the complicated situation, we have found `ppo2` (ea25b9e) as an implementation worth studying. It obtains good performance in both Atari and MuJoCo tasks. More importantly, it also incorporates advanced features such as LSTM and treatment of the `MultiDiscrete` action space, unlocking application to more complicated games such as Real-time Strategy games. As such, we define `ppo2` (ea25b9e) as the **official PPO implementation** and base the remainder of this document on this implementation.

Table 3.2: The best-reported performance of PPO in popular RL libraries in Atari and MuJoCo environments.

RL Library	Benchmark Source	Breakout	Pong	BeamRider	Hopper	Walker2d	HalfCheetah
Baselines `pposgd / ppo1` (da99706)	paper[16] ($)	274.8	20.7	1590	~2250	~3000	~1750
Baselines `ppo2` (7bfbcf1 and ea68f3b)	docs (*)	114.26	13.68	1299.25	2316.16	3424.95	1668.58
Baselines `ppo2` (ea25b9e)	this work (*)	409.265 ± 30.98	20.59 ± 0.40	2627.96 ± 625.751	2448.73 ± 596.13	3142.24 ± 982.25	2148.77 ± 1166.023
Stable-Baselines3	docs (0) (^)	398.03 ± 33.28	20.98 ± 0.10	3397.00 ± 1662.36	2410.43 ± 10.02	3478.79 ± 821.70	5819.09 ± 663.53
CleanRL	docs (1) (*)	~402	~20.39	~2131	~2685	~3753	~1683
Ray/RLlib	repo (2) (*)	201	-	4480	-	-	9664
SpinningUp	docs (3) (^)	-	-	-	~2500	~2500	~3000
ChainerRL	paper (4) (*)	-	-	-	2719 ± 67	2994 ± 113	2404 ± 185
Tianshou	paper (5) (^)	-	-	-	7337.4 ± 1508.2	3127.7 ± 413.0	4895.6 ± 704.3
Tonic	paper (6) (^)	-	-	-	~2000	~4500	~5000

($) The experiments uses the v1 MuJoCo environments
(*) The experiments uses the v2 MuJoCo environments
(0) 1M steps for MuJoCo experiments, 10M steps for Atari games, 1 random seed
(^) The experiments uses the v3 MuJoCo environments
(1) 2M steps for MuJoCo experiments, 10M steps for Atari games, 2 random seeds
(2) 25M steps and 10 workers (5 envs per worker) for Atari experiments; 44M steps and 16 workers for MuJoCo experiments; 1 random seed
(3) 3M steps, PyTorch version, 10 random seeds
(4) 2M steps, 10 random seeds
(5) 3M steps, 10 random seeds
(6) 5M steps, 10 random seeds

3.4 38 Implementation Details

Instead of doing ablation studies and making recommendations on which details matter, this chapter takes a step back and focuses on reproductions of PPO's results in all accounts. Specifically, this work complements prior work in the following ways:

1. **Video Tutorials and Single-file Implementations:** we make video tutorials on re-implementing PPO in PyTorch from scratch, matching details in the official PPO implementation to handle classic control tasks, Atari games, and MuJoCo tasks[1]. Notably, we adopt single-file implementations in our code base, making the code quicker and easier to read.
2. **Implementation Checklist with References:** During our re-implementation, we have compiled an implementation checklist containing 37 details as follows. For each implementation detail, we display the permanent link to its code (which is not done in academic papers) and point out its literature connection.
    - 13 core implementation details
    - 9 Atari-specific implementation details
    - 9 implementation details for robotics tasks (with continuous action spaces)
    - 5 LSTM implementation details
    - 1 `MultiDiscrete` action spaces implementation detail
3. **High-fidelity Reproduction:** To validate our re-implementation, we show that the empirical results of our implementation match closely with those of the original, in classic control tasks, Atari games, MuJoCo tasks, LSTM, and Real-time Strategy (RTS) game tasks.
4. **Situational Implementation Details:** We also cover 4 implementation details not used in the official implementation but potentially useful on special occasions.

Our ultimate purpose is to help researchers understand the PPO implementation through and through, reproduce past results with high fidelity, and facilitate customization for new research. To make research reproducible, we have made source code available[2] and the tracked experiments available[3]

> *Remark* 3.1 (Full Content). The bulk of the document focuses on enumerating the 37 implementation details and making connections to existing literature sources. For this thesis, we do not include its full content and instead refer the readers to the document itself https://iclr-blog-track.github.io/2022/03/25/ppo-implementation-details/

[1] See the video tutorials at https://youtu.be/MEt6rrxH8W4, https://youtu.be/05RMTj-2K_Y, and https://youtu.be/BvZvx7ENZBw.
[2] See https://github.com/vwxyzjn/ppo-implementation-details.
[3] See https://wandb.ai/vwxyzjn/ppo-details.

3.5 Discussions

During our reproduction, we found several useful debugging techniques. They are as follows:

1. **Seed everything**: One debugging approach is to seed everything and then observe when things start to differ from the reference implementation. So you could use the same seed for your implementation and mine, check if the observation returned by the environment is the same, then check if the sample actions are the same. By following the steps, you would check everything to make sure they are aligned (e.g. print out `values.sum()` see if yours match the reference implementation). In the past, we have done this with the pytorch-a2c-ppo-acktr-gail repository and ultimately figured out a bug with our implementation.

2. **Check if `ratio=1`**: Check if the `ratio` are always 1s during the first epoch and first mini-batch update, when new and old policies are the same and therefore the `ratio` are 1s and has nothing to clip. If `ratio` are not 1s, it means there is a bug, and the program has not reconstructed the probability distributions used in rollouts.

3. **Check Kullback-Leibler (KL) divergence**: It is often useful to check if KL divergence goes too high. We have generally found the `approx_kl` stays below 0.02, and if `approx_kl` becomes too high, it usually means the policy is changing too quickly and there is a bug.

4. **Check other metrics**: As shown in the Results section, the other metrics, such as policy and value losses in our implementation, also closely match those in the original implementation. So if your policy loss curve looks very different than the reference implementation, there might be a bug.

5. **Rule of thumb: 400 episodic return in breakout**: Check if your PPO could obtain 400 episodic return in breakout. We have found this to be a practical rule of thumb to determine the fidelity of online PPO implementations in GitHub. Often we found PPO repositories not able to do this, and we know they probably do not match all implementation details of `openai/baselines`' PPO.

If you are doing research using PPO, consider adopting the following recommendations to help improve the reproducibility of your work:

1. **Enumerate implementation details used**: If you have implemented PPO as the baseline for your experiment, you should specify which implementation details you are using. Consider using bullet points to enumerate them like done in this work.

2. **Release locked source code**: Always open source your code whenever possible and make sure the code runs. We suggest adopting proper dependency managers such as poetry or pipenv to lock your dependencies. In the past, we have encountered numerous projects that are based on `pip install -e .`, which 80% of the time would fail to run due to some obscure errors. Having a pre-built `docker` image with all dependencies installed can also help in case the dependencies packages are not hosted by package managers after deprecation.

3. **Track experiments**: Consider using an experiment management software to track your metrics, hyperparameters, code, and others. They can boost your productivity by saving hundreds of hours spent on `matplotlib` and worrying about how to display data. Commercial solutions (usually more mature) include Weights and Biases and Neptune, and open-source solutions include Aim, ClearML, Polyaxon.

4. **Adopt single-file implementation**: If your research requires more tweaking, consider implementing your algorithms using single-file implementations. This document does this and creates standalone files for different environments. For example, our `ppo_atari.py` contains all relevant code to handle Atari games. Such a paradigm has the following benefits at the cost of duplicate and harder-to-refactor code:

- *Easier to see the whole picture*: Because each file is self-contained, people can easily spot all relevant implementation details of the algorithm. Such a paradigm also reduces the burden to understand how files like `env.py`, `agent.py`, `network.py` work together like in typical RL libraries.
- *Faster developing experience*: Usually, each file like `ppo.py` has significantly less LOC compared to RL libraries' PPO. As a result, it's often easier to prototype new features without having to do subclassing and refactoring.
- *Painless performance attribution*: If a new version of our algorithm has obtained higher performance, we know this single file is exactly responsible for the performance improvement. To attribute the performance improvement, we can simply do a `filediff` between the current and past versions, and every line of code change is made explicit to us.

3.5.1 Does modularity help RL libraries?

This document demonstrates reproducing PPO is a non-trivial effort, even though PPO's source code is readily available for reference. Why is it the case? We think one important reason might be that **modularity disperses implementation details**.

Almost all RL libraries have adopted modular design, featuring different modules / files like `env.py`, `agent.py`, `network.py`, `utils.py`, `runner.py`, etc. The nature of modularity necessarily puts implementation details into different files, which is usually great from a software engineering perspective. That is, we don't have to know how other components work when we just work on `env.py`. Being able to treat other components as black boxes has empowered us to work on large and complicated systems for the last decades.

However, this practice might clash hard with ML / RL: as the library grows, it becomes harder and harder to grasp all implementation details w.r.t an algorithm, whereas recognizing all implementation details has become increasingly important, as indicated by this document, Engstrom et al. [20], and Andrychowicz et al. [21]. So what can we do?

Modular design still offers numerous benefits such as 1) easy-to-use interface, 2) integrated test cases, 3) easy to plug different components and others. To this end, good RL libraries are valuable, and we recommend them to write good documentation and refactor libraries to adopt new features. For algorithmic researchers, however, we recommend considering single-file implementations because they are straightforward to read and extend.

3.5.2 Is asynchronous PPO better?

Not necessarily. The high-throughput variant Asynchronous PPO (APPO) [66] has obtained more attention in recent years. APPO eliminates the idle time in the original PPO implementation (e.g., have to wait for all N environments to return observations), resulting in much higher throughput, GPU and CPU utilization. However, APPO involves performance-reducing side-effects, namely stale experiences [34], and we have found insufficient evidence to ascertain its improvement. The biggest issue is:

Underbenchmarked APPO implementation: RLlib has an APPO implementation, yet its documentation contains no benchmark information and suggest "APPO is not always more efficient; it is often better to use standard PPO or IMPALA." Sample Factory [78] presents more benchmark results, but its support for Atari games is still a work in progress. To our knowledge, there is no APPO implementation that simultaneously works with Atari games, MuJoCo or Pybullet tasks, MultiDiscrete action spaces and with an LSTM.

While APPO is intuitively valuable for CPU-intensive tasks such as Dota 2, this document recommends an alternative approach to speed up PPO: **make the vectorized environments really fast**. Initially, the vectorized environments are implemented in python, which is slow. More recently, researchers have proposed to use accelerated vectorized environments. For example, 1. Procgen [79] uses C++ to implement native vectorized environments, resulting in much higher throughput when setting $N = 64$ (N is the number of environments), 1. Envpool uses C++ to offer native vectorized environments for Atari and classic control games, 1. NVIDIA's Isaac Gym [72] uses `torch` to write

hardware-accelerated vectorized environments, allowing the users to spin up $N = 4096$ environments easily, 1. Google's Brax uses jax to write hardware-accelerated vectorized environments, allowing the users to spin up $N = 2048$ environments easily and solve robotics tasks like `Ant` in minutes compared to hours of training in MuJoCo.

In the following section, we demonstrate accelerated training with PPO + envpool in the Atari game Pong.

3.5.3 Solving Pong in 5 minutes with PPO + Envpool

Envpool is a recent work that offers accelerated vectorized environments for Atari by leveraging C++ and thread pools. Our PPO gets a free and side-effects-free performance boost by simply adopting it. We make ~60 lines of code change to `ppo_atari.py` to incorporate this 1 detail, resulting in a self-contained `ppo_atari_envpool.py` (link) that has 365 lines of code.

As shown below, Envpool + PPO runs 3x faster without side effects (as in no loss of sample efficiency):

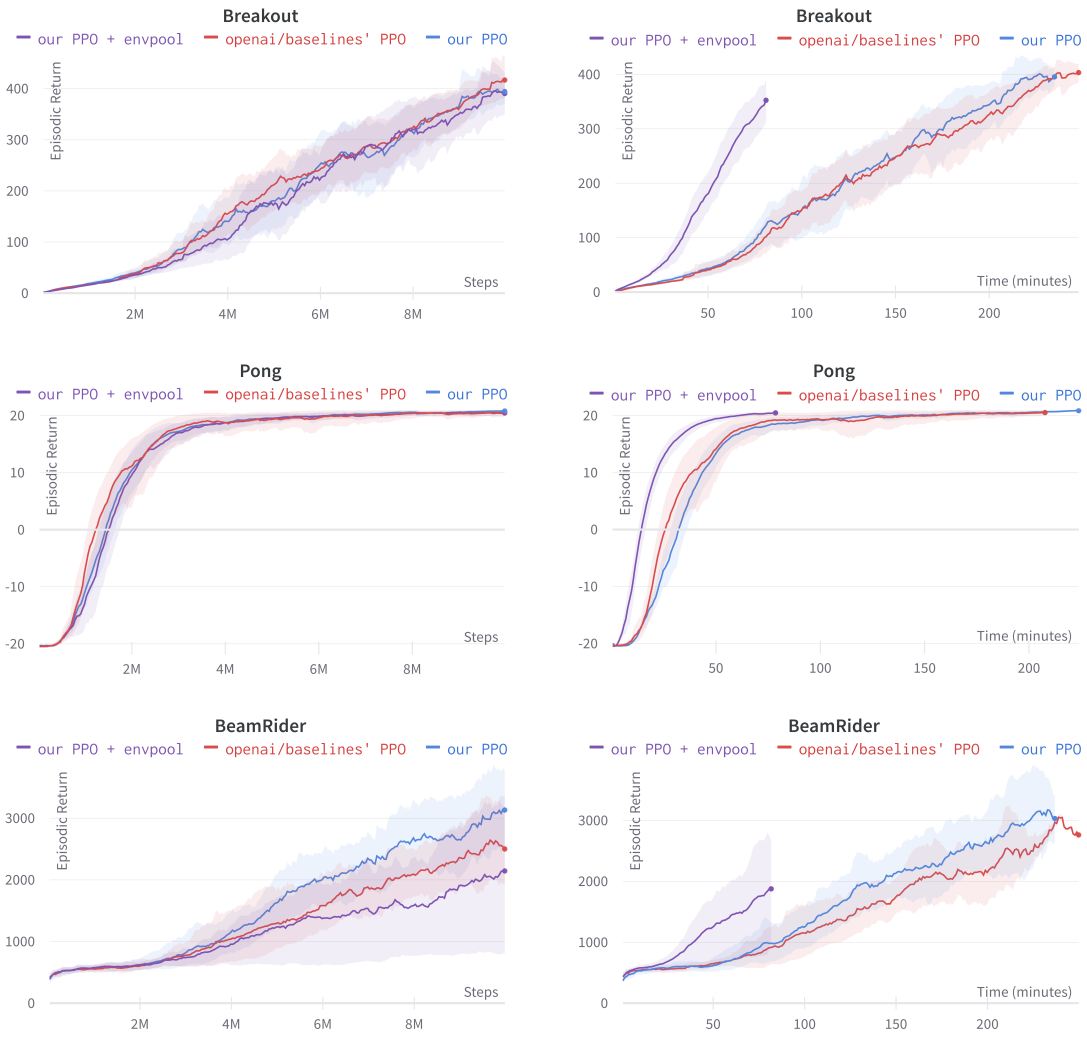


Figure 3.1: Benchmark of PPO + Envpol in Atari tasks.

Two quick notes: 1) the performance deterioration in BeamRider is largely due to a degenerate random seed, and 2) Envpool uses the v5 ALE environments but has processed them the same

way as the v4 ALE environments used in our previous experiments. Furthermore, by tuning the hyperparameters, we obtained a run that solves Pong in 5 mins. This performance is even comparable to IMPALA's[34] results:

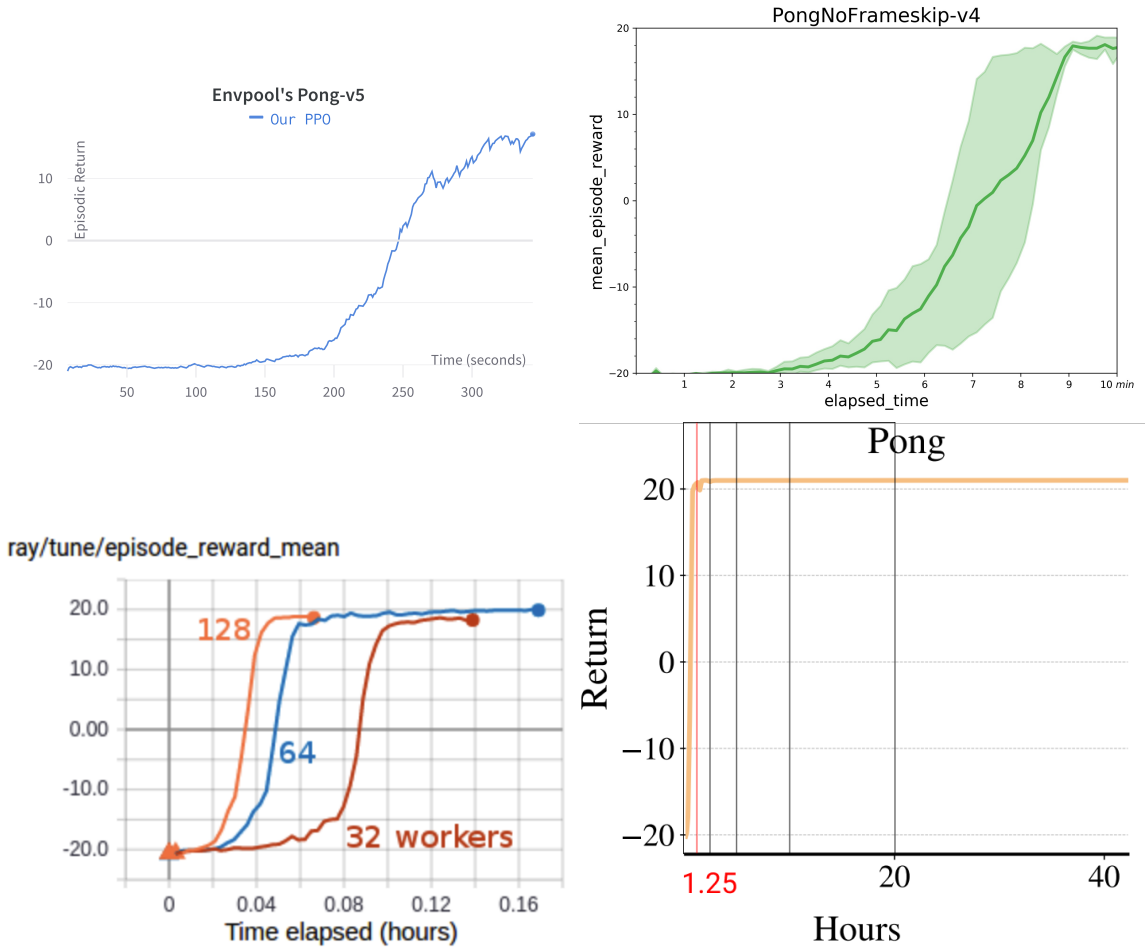


Figure 3.2: The performance of various high-throughput RL libraries in Pong. The top left figure is our tuned experiment that learns Pong in 5 minutes (24 CPU and a RTX 2060). The top right figure is the tuned experiment in PARL thatlearns Pong in 10 minutes (one learner (in a P40 GPU) and 32 actors (in 32 CPUs)). The bottom left figure is the tuned experiment from RLLib that learns Pong in 3 minutes (32, 64, 128 CPUs and presumably a GPU). The top right figure is the tuned experiment in SeedRL[25] that learns Pong in about 45 minutes (8 TPUv3 cores, 213 CPU cores).

We think this raises a practical consideration: adopting async RL such as IMPALA could be more difficult than just making your vectorized environments fast.

3.5.4 Request for Research

Given this document, we believe the community understands PPO better and would be in a much better place to make improvements. Here are a few suggested areas for research.

1. **Alternative choices**: As we have walked through the different details of PPO, it seems that some of them result from arbitrary choices. It would be interesting to investigate alternative choices and see how such change affects results. You can find below a non-exhaustive list of tracks to explore:

- use of a different Atari pre-processing (as partially explored by Machado et al., 2018))
- use of a different distribution for continuous actions (Beta distribution, squashed Gaussian, Gaussian with full covariance, ...), it will most probably require some tuning
- use of a state-dependent standard deviation when using continuous actions (with or without backpropagation of the gradient to the whole actor network)
- use of a different initialization for LSTM (ones instead of zeros, random noise, learnable parameter, ...), use of GRU cells instead of LSTM

2. **Vectorized architecture for experience-replay-based methods**: Experience-replay-based methods such as DQN, DDPG, and SAC are less popular than PPO due to a few reasons: 1) they generally have lower throughput due to a single simulation environment (also means lower GPU utilization), and 2) they usually have higher memory requirement (e.g., DQN requires the notorious 1M sample replay buffer which could take 32GB memory). Can we apply the vectorized architecture to experience-replay-based methods? The vectorized environments intuitively should replace replay buffer because the environments could also provide uncorrelated experience.

3. **Value function optimization**: In Phasic Policy Gradient (Cobbe et al., 2021), optimizing value functions separately turns out to be important. In DQN, the prioritized experience replay significantly boosts performance. Can we apply prioritized experience replay to PPO or just on PPO's value function?

3.6 Conclusion

Reproducing PPO's results has been difficult in the past few years. While recent works conducted ablation studies to provide insight on the implementation details, these works are not structured as tutorials and only focus on details concerning robotics tasks. As a result, reproducing PPO from scratch can become a daunting experience. Instead of introducing additional improvements or doing further ablation studies, this document takes a step back and focuses on delivering a thorough reproduction of PPO in all accounts, as well as aggregating, documenting, and cataloging its most salient implementation details. This document also points out software engineering challenges in PPO and further efficiency improvement via the accelerated vectorized environments. With these, we believe this document will help people understand PPO faster and better, facilitating customization and research upon this versatile RL algorithm.

Chapter 4: CleanRL - Understandable DRL Library

Chapter 3 highlights the importance of understanding the implementation details of PPO and advocates using single-file implementations for implementation transparency. Shared with the same spirit, this chapter introduced a DRL library called `CleanRL` that builds single-file implementations for other popular DRL algorithms such as DQN, DDPG, SAC, and others.

`CleanRL` is an open-source library that provides high-quality single-file implementations of Deep Reinforcement Learning (DRL) algorithms. These single-file implementations are self-contained algorithm variant files such as `dqn.py`, `ppo.py`, and `ppo_atari.py` that individually include all algorithm variant's implementation details. Such a paradigm significantly reduces the complexity and the lines of code (LOC) in each implemented variant, which makes them quicker and easier to understand. This paradigm gives the researchers the most fine-grained control over all aspects of the algorithm in a single file, allowing them to prototype novel features quickly. Despite having succinct implementations, CleanRL's codebase is thoroughly documented and benchmarked to ensure performance is on par with reputable sources. As a result, CleanRL produces a repository tailor-fit for two purposes: 1) understanding all implementation details of DRL algorithms and 2) quickly prototyping novel features. CleanRL's source code can be found at https://github.com/vwxyzjn/cleanrl. The work of this chapter is based on the following publication:

- Shengyi Huang, Rousslan Fernand Julien Dossa, Chang Ye, Jeff Braga, Dipam Chakraborty, Kinal Mehta, and João G.M. Araújo. Cleanrl: High-quality single-file implementations of deep reinforcement learning algorithms. *Journal of Machine Learning Research*, 23(274):1–18, 2022. URL http://jmlr.org/papers/v23/21-1342.html

4.1 Authorship

Shengyi Huang and Rousslan Fernand Julien Dossa co-founded CleanRL and has led its overall development. Chang Ye contributed a prototype with Random Network Distillation[80]. Shengyi Huang, Rousslan Fernand Julien Dossa, and Chang Ye are the main code reviewers and maintainers. Jeff Braga contributed hundreds of hours of tracked experiments in Weights and Biases and submitted various codebase improvements. Dipam Chakraborty contributed the Phasic Policy Gradient implementation. Kinal Mehta contributed the Deep Q-learning implementation with JAX. João G.M. Araújo contributed the Twin-Delayed Deep Deterministic Policy Gradient implementation with JAX. Shengyi Huang, Rousslan Fernand Julien Dossa, Chang Ye, Jeff Braga, Dipam Chakraborty, Kinal Mehta, João G.M. Araújo wrote the paper. Despite the joint authorship of the library, all the text in this chapter was written by Shengyi Huang.

4.2 Motivation

In recent years, Deep Reinforcement Learning (DRL) algorithms have achieved great success in training autonomous agents for tasks ranging from playing video games directly from pixels to robotic control[16;69;81]. At the same time, open-source DRL libraries also flourish in the community[82–86]. Many of them have adopted good modular designs and fostered vibrant development communities. Nevertheless, understanding all the implementation details of an algorithm remains difficult because these details are spread to different modules. However, understanding these implementation details is essential because they could significantly affect performance[20].

In this chapter, we introduce `CleanRL`, a DRL library based on single-file implementations to help researchers understand all the details of an algorithm, prototype new features, analyze experiments, and scale the experiments with ease. `CleanRL` is a *non-modular* library. Each algorithm variant in `CleanRL` is self-contained in a single file, in which the lines of code (LOC) have been trimmed to the

Figure 4.1: Filediff in Visual Studio Code: left click select `ppo_atari.py` then `cmd/ctrl` + left click select `ppo_continuous_action.py` to highlight neural network architecture differences of PPO when applying to Atari games and MuJoCo tasks.

bare minimum. Along with succinct implementations, CleanRL's codebase is thoroughly documented and benchmarked to ensure performance is on par with reputable sources. For example, our Proximal Policy Optimization (PPO)[16] implementation with Atari games is a single file `ppo_atari.py` using only 337 LOC, yet it closely matches `openai/baselines`' PPO performance in the game breakout (Appendix A.1), making it much easier to understand the algorithm in one go. In contrast, the researchers using modular DRL libraries often need to understand the modular design (usually 7 to 20 files) which can contain thousands of LOC. As a result, CleanRL is tailor-fit for two purposes: 1) understanding all implementation details of DRL algorithms and 2) quickly prototyping novel features.

4.3 Single-file Implementations

Despite the many features modular DRL libraries offer, understanding all the relevant code of an algorithm is a non-trivial effort. As an example, running the PPO model in Atari games using `Stable Baselines 3 (SB3)` with a debugger involves jumping back and forth between 20 python files that comprise 4000+ LOC[82] (Appendix A.5). This makes it difficult to understand how the algorithm works due to the sheer amount of code and its complex structure. This is a problem because even small implementation details can have a large impact on the performance of deep RL algorithms[20], and understanding them has become increasingly important.

`CleanRL` makes it much easier to understand implementation details with a simple idea — putting all implementation details of an algorithm variant into a single file. We call this practice "single-file implementations." Single-file implementations allow us to focus on implementing a specific variant without worrying about handling special cases. Also, for utilities that are not relevant to the algorithm itself, like logging and plotting, we import third-part libraries. As a result, CleanRL produces a codebase with an order of magnitude fewer LOC for each algorithm variant. For example, we have a:

1. `ppo.py` (321 LOC) for the classic control environments, such as `CartPole-v1`,
2. `ppo_atari.py` (337 LOC) for the Atari environments[12],
3. `ppo_continuous_action.py` (331 LOC) for the robotics environments (e.g., MuJoCo, PyBullet) with continuous action spaces[16].

The single-file implementations have the following benefits.

Transparent learning experience It becomes easier to recognize all aspects of the code in one place. By looking at `ppo.py`, it is straightforward to recognize the core implementation details of PPO. It also becomes easier to identify the difference between algorithm variants via `filediff`. For example, comparing `ppo.py` with `ppo_atari.py` shows a 30 LOC difference required to add environment prepossessing and modify neural networks. Meanwhile, another comparison with `ppo_continuous_action.py` shows a 25 LOC difference required to use normalization and account for continuous action space. See Figure 4.1 as an example. Being able to display the variant's differences explicitly has helped us explain 37 implementation details of PPO[22].

Better debug interactivity Everything is located in a single file, so when debugging, the user does not need to browse different modules like in modular libraries. Additionally, most variables in the files exist in the *global Python name scope*. This means the researchers can use `Ctrl+C` to stop the program execution and check most variables and their shapes in the interactive shell (Appendix A.2). This is more convenient than using the Python's debugger, which only shows the variables in a specific name scope like in a function.

Painless performance attribution If a new version of our algorithm has obtained a higher performance, we know the exact single file which is responsible for the performance improvement. To attribute the performance improvement, we can simply do a filediff between the current and past versions, and every line of code change is made explicit to us. In comparison, two different versions of modular RL libraries usually involve dozens of file changes, which are more difficult to compare.

Faster prototyping experience CleanRL gives researchers fine-grained control to everything related to the algorithm in a single file, hence making it efficient to develop prototypes without having to subclass like in other modular RL libraries. As an example, invalid action masking[36] is a common technique used in games with large, parameterized action spaces. With `CleanRL`, it takes about 40 LOC to implement[22] Sec. 4, whereas in other libraries it could take substantially more LOC (e.g., more than 600 LOC, excluding the test cases[1]) because of overhead such as re-factoring the functional arguments and making more general classes.

Because of these benefits, we have also implemented single-file implementations for Deep Q-learning[81], Categorical Deep Q-learning[55], Deep Deterministic Policy Gradient[69], Twin-delayed Deep Deterministic Policy Gradient[70], Soft Actor-cirtic[87], Phasic Policy Gradient[88], and Random Network Distillation[80].

Despite of these benefits of single-file implementations, one downside is the excessive amount of duplicate code. To help reduce the maintenance overhead, we have adopted a series of developmental tools to format code, pin dependencies automatically, scale experiments with cloud providers, etc (Appendix A.3).

4.4 Documentation and Benchmark

All CleanRL's single-file implementations are thoroughly documented and benchmarked in our main documentation site (`https://docs.cleanrl.dev/`). For each single-file implementation, we document the original paper and relevant information, usage, an explanation of logged metrics, noteworthy implementation details, and benchmark results which include learning curves, a table comparing performance against reputable sources when applicable, and links to the tracked experiments. In particular, the benchmark experiments are tracked with Weights and Biases[89], which allows the users to interactively explore other tracked data such as system metrics, hyperparameters, and the agents' gameplay videos. For convenience, we have included tables comparing the performance of CleanRL's single-file implementations against reputable sources when applicable (Appendix A.1).

4.5 When to Use CleanRL

CleanRL has its own set of pros and cons like other popular modular RL libraries. For example, modular DRL libraries, such as SB3, offer a friendly end user API — if an end user does not

[1] See `https://github.com/Stable-Baselines-Team/stable-baselines3-contrib/pull/25`.

know much about DRL but wants to apply PPO in their tasks, SB3 would be a great fit. Among many other benefits, SB3 makes it easy to configure different components. CleanRL does not have a friendly end user API like `agent.learn()`, but it exposes all implementation details and is easy to read, debug, modify for research, and study RL. Comparatively, CleanRL is well-suited for researchers who need to understand all implementation details of DRL algorithms, and prototype novel features quickly.

CleanRL complements the DRL research community with a unique developing experience. In fact, there is a win-win situation for CleanRL and SB3: "prototype with CleanRL and port to SB3 for wider adoption in the community." CleanRL's codebase often allows researchers to prototype specialized features much quicker. As shown above, the invalid action masking technique with PPO takes $\sim$40 LOC to implement. Once we have rigorously validated this technique, our results and analysis will provide concrete guidance for porting this technique to SB3, which enable our technique to reach a wider range of audience given SB3's friendly end user APIs.

Chapter 5: Cleanba: Reproducible Distributed DRL

One limitation of the works presented in previous chapters is that they focus exclusively on synchronous DRL that uses a single GPU. However, more recent DRL systems and algorithms are distributed to leverage more computational resources. Despite recent progress in the field, reproducibility issues have not been sufficiently explored.

In this chapter, we analyze the typical distributed actor-learner framework[34] and show that it can have reproducibility issues even if random seeds are controlled. We then introduce Cleanba, a new open-source platform for distributed DRL that proposes a highly reproducible architecture. Cleanba implements highly-optimized distributed variants of PPO[16] and IMPALA[34]. Our Atari experiments show that these variants can obtain equivalent or higher scores than `moolib`'s IMPALA, but with 1) 30% less training wall time under the 8 GPU setting and 2) more reproducible learning curves in different hardware settings.

5.1 Authorship

Shengyi Huang led the design and implementation of Cleanba. Jiayi Weng and Min Lin added numerous features to EnvPool (e.g., XLA interface, Procgen support, and bug fixes) in direct support of Cleanba. Rujikorn Charakorn Contributed to the implementation of PPO and ran parts of the reproducibility experiments. Zhongwen Xu helped advise and support this project. Shengyi Huang led the writing and incorporated feedback from Rujikorn Charakorn and Zhongwen Xu.

5.2 Motivation

Deep Reinforcement Learning (DRL) is a paradigm to train autonomous agents to perform tasks. In recent years, it has demonstrated remarkable success across various domains, including video games[11], robotics control[16], chip design[77], and large language model tuning[90]. Concurrent with the development of DRL is the rise of distributed DRL[25;34], a fast-growing field that leverages more computing resources to train agents. Despite recent progress, reproducibility issues in distributed DRL have not been sufficiently explored. This paper introduces Cleanba, a new platform for distributed DRL that addresses reproducibility issues under different hardware settings.

Reproducibility in DRL is a challenging issue. Not only are DRL algorithms brittle to hyperparameters and neural network architectures[17], implementation details are often crucial for successfully applying DRL but frequently omitted from publications[20–22]. Reproducibility issues in distributed DRL are under-studied and arguably even more challenging. In particular, most high-profile distributed DRL works, such as Apex-DQN[24], IMPALA[34], R2D2[91], and Podracer Sebulba[92] are not (fully) open-source. Furthermore, earlier work pointed out that more actor threads not only improve training speed but cause reproducibility issues – different hardware settings could impact the data efficiency in a non-linear fashion[15].

This need not be the case. In this paper, we seek a side-effects-free distributed DRL paradigm in which different hardware settings could make training speed slower or faster but do not impact data efficiency, thus making scaling results more reproducible and predictable. We first analyze the typical actor-learner architecture and show its parallelism paradigm makes itself not reproducible even if random seeds are controlled. We then propose a distributed architecture that is reproducible if random seeds are controlled, but it can also benefit from interleaving the actor and learner's computations. Based on this architecture, we introduce our Cleanba (meaning **Clean**RL-style[44] Podracer Sebul**ba**) distributed DRL platform, which aims to be an easy-to-understand distributed DRL infrastructure like CleanRL, but also be scalable as Podracer Sebulba. Cleanba implements a distributed variant of PPO[16] and IMPALA[34]. Next, we evaluate Cleanba's variants against `moolib`[93]'s IMPALA on 57 Atari games[12]. Our experiments show that Cleanba's variants can

obtain equivalent or higher scores than `moolib`'s IMPALA, but with 1) 30% less training wall time under the 8 GPU setting and 2) more reproducible learning curves in different hardware settings.

To facilitate more transparency and reproducibility, we have made available our source code at https://github.com/vwxyzjn/cleanba, trained models at https://huggingface.co/cleanrl, and tracked experiments at https://wandb.ai/openrlbenchmark/cleanba.

5.3 Background

Distributed DRL Systems Utilizing more computational power has been an attractive topic for researchers. Earlier DRL methods like DQN[11] were synchronous and typically used a single simulation environment, which made them slow and inefficient in using hardware resources. A3C spawns multiple actor threads; each interacts with its own copy of the environment and asynchronously accumulates gradient. To make distributed DRL more scalable, IMPALA decouples the actors and the learners[25;34]. The actors produce training data asynchronously, while the learners produce new agent parameters, which are transferred asynchronously to the actor. Actor-learner systems can achieve higher throughput and shorter training wall time than A3C. Additional distributed actor-learner systems include GA3C[94], IMPALA[34], Apex-DQN[24], R2D2[91], and Podracer Sebulba[92].

Reproducibility Issues with Different Hardware Settings Empirical evidence suggests that increasing the number of actor threads can enhance the training speed in distributed DRL (Mnih et al.[15], Fig. 4). However, this augmentation is not without its complications. It also impacts data efficiency and final Atari scores (Mnih et al.[15], Fig. 3), and these effects could manifest in a non-linear manner. While the authors found the side effects of value-based asynchronous methods to be positive and improve data efficiency, the side effects of contemporary distributed DRL systems, such as IMPALA, Apex-DQN, and R2D2, across various hardware configurations, have not been sufficiently explored.

Open-source Distributed DRL Infrastructure Most distributed deep reinforcement learning (DRL) algorithms are not open-source. There have been many notable distributed DRL replications in the open-source software (OSS) community. These efforts include SEED RL[25], `rlplyt`[95], Decentralized Distributed PPO[96], Sample Factory[78], HTS-RL[97], `torchbeast`[98], and `moolib`[93]. Many of them have shown high throughput and good empirical performance in select domains. Nevertheless, most of them either do not have evaluations on 57 Atari games or have various hardware restrictions, leading to reproducibility concerns. `moolib` is the only OSS infrastructure that has both evaluations on 57 Atari games in the standard 200M frames setting and can scale beyond a single GPU setting[1].

5.4 Preliminaries

Let us consider the RL problem in a *Markov Decision Process (MDP)*[45], where $\mathcal{S}$ is the state space and $\mathcal{A}$ is the action space. The agent performs some actions to the environment, and the environment transitions to another state according to its *dynamics* $P(s' \mid s, a) : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$. The environment also provides a scalar reward according to the reward function $R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, and the agent attempts to maximize the expected discounted return following a policy π:

$$
\begin{gathered}
J(\pi) = \mathbb{E}_\tau \left[G(\tau) \right] \\
\text{where } \tau \text{ is the trajectory } \left(s_0, a_0, r_0, \ldots, s_{T-1}, a_{T-1}, r_{T-1}, s_T \right) \\
\text{and } s_0 \sim \rho_0, s_t \sim P(\cdot|s_{t-1}, a_{t-1}), a_t \sim \pi_\theta(\cdot|s_t), r_t = r\left(s_t, a_t\right)
\end{gathered}
\tag{5.1}
$$

PPO[16] is a popular algorithm that proposes a clipped policy gradient objective to help avoid

[1] While SEED RL also has evaluations on 57 Atari games and scale beyond 1 GPU, SEED RL trained the agents for 40 billion frames 40 hours per game.

Algorithm 4 Proximal Policy Optimization

1: **Initialize** environment E containing `local_num_envs` parallel sub-environments
2: **Initialize** policy parameters θ_π, value parameters θ_v, optimizer O
3: **Initialize** observation s_{next}, done flag d_{next}
4: **for** $i = 0,1,2,...,$ I **do**
5: **Set** $\mathcal{D} = (s, a, \log \pi(a|s), r, d, v)$ as tuple of 2D arrays
6: **for** $t = 0,1,2,...,$ `num_steps` **do** ▷ Rollout Phase
7: **Cache** $o_t = s_{next}$ and $d_t = d_{next}$
8: **Get** $a_t \sim \pi(\cdot|s_t; \theta_\pi)$ and $v_t = v(s_t; \theta_v)$
9: **Step** simulator: $s_{next}, r_t, d_{next} = E.step(a_t)$
10: **Store** $s_t, d_t, v_t, a_t, \log \pi(a_t|s_t; \theta_\pi), r_t$ in $\mathcal{D}$
11: **Estimate** next value $v_{next} = v(s_{next})$ ▷ Learning Phase
12: **Compute** advantage $\hat{A}^{\text{adv}}_\pi$ and return R using $\mathcal{D}$ and v_{next}
13: **Prepare** the batch $\mathcal{B} = \mathcal{D}, \hat{A}^{\text{adv}}_\pi, R$ and flatten $\mathcal{B}$
14: **for** *epoch* $= 0,1,2,...,$ `update_epochs` **do**
15: **for** mini-batch $\mathcal{M}$ of size m in $\mathcal{B}$ **do**
16: **Normalize** advantage $\mathcal{M}.\hat{A}^{\text{adv}}_\pi$
17: **Compute** policy loss L^π, value loss L^V, and entropy loss L^S using $\mathcal{M}$
18: **Back-propagate** joint loss $L = -L^\pi + c_1 L^V - c_2 L^S$
19: **Clip** maximum gradient norm of θ_π and θ_v to 0.5
20: **Step** optimizer O w.r.t. θ_π and θ_v

unstable updates[16;19]:

$$J^{\text{CLIP}}(\pi_\theta) = \mathbb{E}_\tau \left[\sum_{t=0}^{T-1} \min \left(r_t(\theta) \hat{A}^{\text{adv}}_\pi(s_t, a_t), \text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right) \hat{A}^{\text{adv}}_\pi(s_t, a_t) \right) \right] \tag{5.2}$$

where $\pi_{\theta_{\text{old}}}$ is the policy parameter before the update, $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$, $\hat{A}^{\text{adv}}_\pi$ is an advantage estimator called Generalized Advantage Estimator[46], and ϵ is PPO's clipped coefficient. During the optimization phase, the agent also learns the value function and maximizes the policy's entropy, therefore optimizing the following joint objective:

$$J^{\text{JOINT}}(\theta) = J^{\text{CLIP}}(\pi_\theta) - c_1 J^{\text{VF}}(\theta) + c_2 S[\pi_\theta], \tag{5.3}$$

where c_1, c_2 are coefficients, S is an entropy bonus, and J^{VF} is the squared error loss for the value function associated with π_θ. Algorithm 4 shows the pseudocode of PPO that more accurately reflects how PPO is implemented in the original codebase[2]. For more detail on PPO's implementation, see[22]. Given this pseudocode, the following list unifies the nomenclature/terminology of PPO's key hyperparameters.

- `world_size` is the number of instances of training processes; typically this is 1 (e.g., you have a single GPU).
- `local_num_envs` is the number of parallel environments PPO interacts within an instance of the training process (see line 1). `num_envs` = `world_size` $*$ `local_num_envs` is the total number of environments across all training instances.
- `num_steps` is the number of steps in which the agent samples a batch of `local_num_envs` actions and receives a batch of `local_num_envs` next observations, rewards, and done flags from the simulator (see line 6), where the done flags signal if the episodes are terminated or truncated. `num_steps` has many names, such as the "sampling horizon"[95] and "unroll length"[99].

[2] https://github.com/openai/baselines

- `local_batch_size` is the batch size calculated as `local_num_envs` $*$ `num_steps` within an instance of the training process (`local_batch_size` is the size of the $\mathcal{B}$ in line 13).
- `batch_size` = `world_size` $*$ `local_batch_size` is the aggregated batch size across all training instances.
- `update_epochs` is the number of update epochs that the agent goes through the training data in $\mathcal{B}$ (see line 14).
- `num_minibatches` is the number of mini-batches that PPO splits $\mathcal{B}$ into (see line 15).
- `local_minibatch_size` is m = `local_batch_size` / `num_minibatches`, the size of each mini-batch $\mathcal{M}$ (see line 15). `minibatch_size` = `world_size` $*$ `local_minibatch_size` is the aggregated batch size across all training instances.

To make understanding more concrete, let us consider an example of Atari training. Typically, PPO uses a single training instance (i.e., `world_size` = 1), `local_num_envs` = `num_envs` = 8, and `num_steps` = 128. In the rollout phase (line 6-10), the agent collects a batch of $8 * 128 = 1024$ data points in $\mathcal{D}$. Then, suppose `num_minibatches` = 4, $\mathcal{D}$ is evenly split to 4 mini-batches of size $m = 1024/4 = 256$. Next, if $K = 4$, the agent would perform K * `num_minibatches` = 16 gradient updates in the learning phase (line 11-20).

We consider two options to scale to larger training data. **Option 1** is to increment `local_num_envs` – the agent interacts with more environments, and as a result, the training data is larger. The second option is to increment `world_size` – have two or more copies of Algorithm 4 running in parallel and average the gradient of the copies in line 20. **Option 2** is especially desirable when the users want to leverage more computational resources, such as GPUs.

Note that both options can be equivalent *in terms of hyperparameters*. For example, when setting `world_size` = 2, the agent effectively interacts with two distinct sets of `local_num_envs` environments, making its `num_envs` doubled. To make option 1 achieve the same hyperparameters, we just need to double its `local_num_envs`. Below is a table summarizing the resulting hyperparameters of both options.

Hyperparameter	**Option 1**: Increment `local_num_envs`	**Option 2**: Increment `world_size`
`world_size`	1	2
`local_num_envs`	120	60
`num_envs`	**120**	**120**
`num_steps`	128	128
`local_batch_size`	15360	7680
`batch_size`	**15360**	**15360**
`num_minibatches`	4	4
`local_minibatch_size`	3840	1920
`minibatch_size`	**3840**	**3840**

Importantly, we can get the same hyperparameter configuration for PPO by adjusting `local_num_envs` and `world_size` accordingly. That is, we can obtain the same `num_envs`, `batch_size`, and `minibatch_size` core hyperparameters.

5.5 Reproducibility Issues in IMPALA

To advance scientific progress, it is important to understand how to attain reliable reproducibility with distributed DRL. Generally, there are three components to reproducing identical model pa-

IMPALA Actor-Learner Architecture

```
batch_size = 32
agent = Agent()
data_Q = queue()

def actor():
  while True:
    data = rollout(agent.param, 1)


    data_Q.put(data)
def learner():
  for _ in range(1, ITER):
    data = data_Q.get_many(batch_size)
    agent.learn(data)
    broadcast_to_actors(agent.param)
for _ in range(num_actors):
  thread(actor).start()
thread(learner).start()
```

Cleanba's architecture

```
batch_size = 32
agent = Agent()
data_Q = queue(len=1)
param_Q = queue(len=1)
def actor():
  for i in range(1, ITER):
    if i != 2:
      params = param_Q.get()
    data = rollout(params, batch_size)
    data_Q.put(data)
def learner():
  for _ in range(1, ITER):
    data = data_Q.get()
    agent.learn(data)
    param_Q.put(agent.param)
param_Q.put(agent.param)
thread(actor).start()
thread(learner).start()
```

Figure 5.1: The pseudocode for IMPALA architecture (left) and Cleanba's architecture (right). Colors are used to highlight the code differences between the two architectures. The `rollout(params, num_envs)` function collects rollout data on `num_envs` independent environments for M (`num_steps`) steps.

rameters across runs[3]: 1) seeding the neural networks, 2) seeding the environments, and 3) using the same hyperparameters. When achieving completing these components, PPO in Algorithm 4 becomes a fully deterministic algorithm and can reproduce identical models, learning curves, and losses. In this section, we show that IMPALA has additional non-determinism by nature, which arises from the concurrent scheduling of different actor threads. This non-determinism could further cause subtle learning results and reproducibility issues.

5.5.1 Non-determinisim of IMPALA's Architecture

Figure 5.1 lists a typical pseudocode of IMPALA, which spawns many actor CPU threads (lines 16-17) to collect rollout data, which are then transferred into a queue, from which the learner gets data and performs an optimization step. The new policy is then asynchronously broadcasted to all the actor threads[4].

It is important to note that IMPALA struggles to consistently reproduce identical model parameters across multiple runs when the actor count exceeds one. This difficulty persists even when the neural networks and environments are seeded. This is primarily due to the actors running at varying rates to populate the rollout data queue, which results in differing rollout data across runs. Consequently, the reproduction of identical model parameters becomes an unlikely outcome.

However, non-determinism can be desirable in parallel programming because they make programs faster without making outputs significantly different. For example, some of NVIDIA's CuDNN operations are inherently non-determinisitic[5]. As a result, it is more important to investigate if this non-determinism could cause a performance difference, in terms of learning curves.

5.5.2 Algorithmic Reproducibility Issues

A natural question arises: what happens when the learner produces a new policy while the actor is in the middle of producing a trajectory? It turns out multiple policy versions could contribute

[3] Achieving perfect reproducibility is difficult in the context of deep learning because non-determinism is everywhere, such as in GPU and operating systems [1].

[4] Sharing a single set of parameters is the default setting for original IMPALA [34]

[5] https://docs.nvidia.com/deeplearning/cudnn/developer-guide/index.html#reproducibility

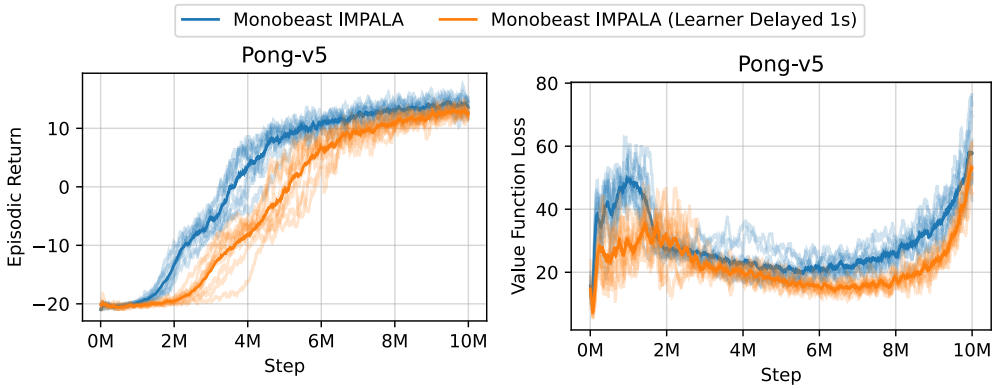


Figure 5.2: Episodic return and value function loss of two sets of `monobeast` experiments that use the *exact same hyperparameters*, but the orange set of experiments has its learner update manually delayed for 1 second.

to a single trajectory of length `num_steps`, especially if the learner updates are fast and frequent. However, this impacts the rollout data construction in a non-trivial way. From a reproducibility point of view, it is important to realize the frequency at which the policies are updated is a source of non-determinism.

To demonstrate this effect, we manufacture a specific experiment that magnifies this non-determinism in `monobeast`'s IMPALA. For the controlled group, we decreased the number of trajectories in the batch from 32 to 8 to reduce training time, thus making the actor's policy updates more frequent. We further used 80 actor threads and increased `monobeast`'s default unroll length (`num_steps`) from 20 to 240 to increase the chance of observing the actor's policy updates in the middle of a trajectory. For the experimental group, we *manually slowed down the policy broadcasting* by sleeping the learner for 1 second after the policy updates in order to simulate a case where the learner is significantly slower (such as when running the learner on CPU).

We found that in the control group, the actors, on average, changed their policy versions 12-13 times in the middle of the 240-length trajectory (Appendix C.5). In the experimental group, because of the manual slowdown in broadcasting the learner's policy, the actors, on average, changed the policy one time. We note that the results vary on different hardware settings as well. For example, the control group changed their policy versions, on average, eight times when using 40 actor threads. We noted that in `moolib`, the actor's policy could also change mid-rollout.

Figure 5.2 demonstrates the empirical effect of the experiments. Note that the learning and loss curves looked notably different across ten random seeds, even though the control and experimental group have the *exact same hyperparameters*. This experiment shows that IMPALA algorithmically could be susceptible to reproducibility issues across different hardware settings. While Figure 5.2 only shows the experimental results on one environment, the primary purpose of it is to show that this issue exists and is barely predictable. Furthermore, this type of issue can be much more subtle and difficult to diagnose at a much larger scale, so it is important that we investigate them.

5.6 Towards Reproducible Distributed DRL

Despite these reproducibility issues, the actor-learner architecture is useful because it allows us to interleave the computations of the actors and learners. In this work, we address the reproducibility issues mentioned above by 1) decoupling hyperparameters and hardware settings and 2) proposing a synchronization mechanism that makes distributed DRL reproducible.

Table 5.1: The different architectures and their rollout data compositions.

Iteration	1	2	3
Cleanba's Architecture, Actor	$\pi_1 \to \mathcal{D}_{\pi_1}$	$\pi_1 \to \mathcal{D}_{\pi_1}$	$\pi_2 \to \mathcal{D}_{\pi_2}$
Cleanba's Architecture, Learner		$\mathcal{D}_{\pi_1} \to \pi_2$	$\mathcal{D}_{\pi_1} \to \pi_3$
Synchronous Architecture	$\pi_1 \to \mathcal{D}_{\pi_1}$, $\mathcal{D}_{\pi_1} \to \pi_2$	$\pi_2 \to \mathcal{D}_{\pi_2}$ $\mathcal{D}_{\pi_2} \to \pi_3$	$\pi_3 \to \mathcal{D}_{\pi_3}$ $\mathcal{D}_{\pi_3} \to \pi_4$

5.6.1 Decoupling hyperparameters and hardware settings

As mentioned in the previous section, different numbers of actor threads could make policy updates more or less frequent in the middle of a trajectory generation. This is unpredictable and need not be the case. A different number of actors also creates a different number of simulation environments and thus should be recognized as a hyperparameter setting.

To make a more clarified setting, we advocate decoupling the number of actor threads into two separate hyperparameters: 1) the number of environments, corresponding to `local_num_envs` in Algorithm 4, and 2) the number of CPUs. In this case, we can use a different number of CPUs to simulate a given number of environments. This decoupled interface is readily provided by EnvPool[37], which we use in our proposed architecture.

5.6.2 Deterministic Rollout Data Composition

To address the non-determinism in rollout data composition, we propose our *Cleanba's architecture*, which retains the benefit of interleaving actor-learner computations but can produce deterministic rollout data composition.

At its core, Cleanba's architecture is a simple mechanism for synchronizing the actor and learner, ensuring the actor's policy version is **always exactly one step** behind the learner's policy version. Figure 5.1 is the pseudocode of the architecture, where the `rollout` function corresponds to the rollout phase and `agent.learn` corresponds to the learning phase in PPO. While we used PPO as an example, Cleanba's architecture can be similarly used in IMPALA.

Let us use π_i to denote the policy of version i and $\mathcal{D}_{\pi_i}$ the rollout data created by π_i. In the second iteration of Figure 5.1, we skipped the `param_Q.get()` call, so $\pi_1 \to \mathcal{D}_{\pi_1}$ happens concurrently with $\mathcal{D}_{\pi_1} \to \pi_2$. Because `Queue.get` is blocking when the queue is empty and `Queue.put` is blocking when the queue is full (we set the maximum size to be 1), we made sure the actor's policy version is exactly 1 version preceding the learner's policy version, as demonstrated in Table 5.1. As a result, Cleanba's architecture can interleave the actor and learner's computation.

Cleanba's architecture above has several benefits. First, it is easy to reason and reproduce. As highlighted in the table above, we can ascertain the specific policy used for collecting the rollout data, and we know with certainty that the size of the data is `local_batch_size`. This knowledge about which policy generates the rollout data enhances the transparency and reproducibility of distributed RL. Second, Cleanba's architecture is easy to debug for throughput. For diagnosing throughput, we can evaluate the time taken for `rollout_Q.get()` and `param_Q.get()`. If, on average, `rollout_Q.get()` consumes less time than `param_Q.get()`, it becomes evident that learning is the bottleneck, and vice versa. This further means we can scale architecture to a distributed setting while maintaining good reproducibility principles.

Based on Cleanba's architecture, this work introduces Cleanba as a reproducible distributed DRL platform. Cleanba is inspired by DeepMind's Sebulba Podracer architecture[92] and is also highly efficient. Its implementation uses JAX[100] and EnvPool[37], both of which are designed to be efficient. To improve the learner's throughput, we allow using multiple learner devices via `pmap`. To improve the system's scalability, we use `jax.distibuted` to distribute to more nodes.

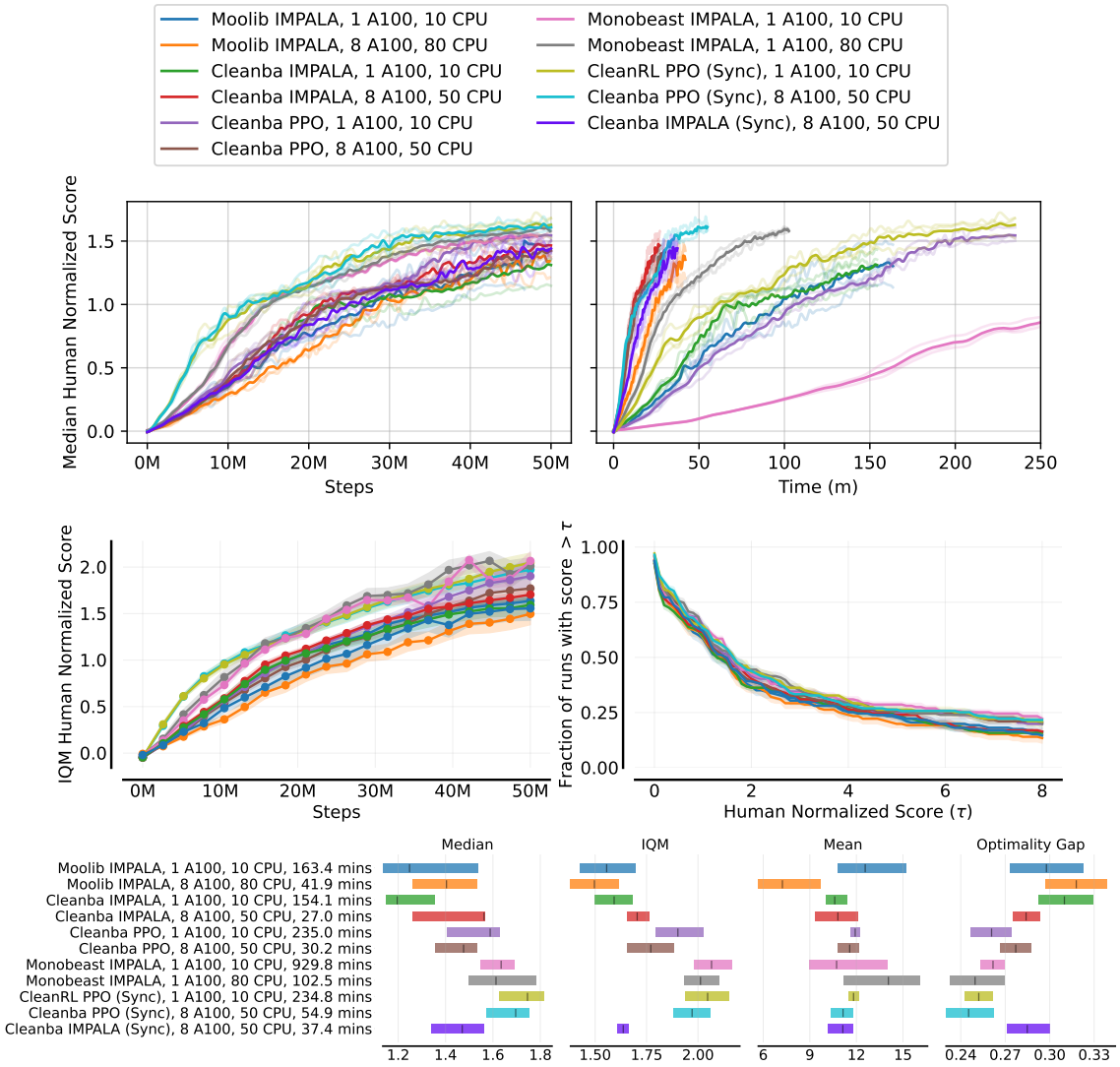


Figure 5.3: Top figure: the median human-normalized scores of Cleanba variants compared with `moolib`. Middle figure: the IQM human-normalized scores and performance profile[1]. Bottom figure: the average runtime in minutes and aggregate human normalized score metrics with 95% stratified bootstrap CIs.

5.7 Experiments

We tested the Cleanba variants on Atari games[12]. To make a more direct and fair comparison, we controlled the hardware and environment simulation setups and compared only the reference IMPALA implementations in `moolib` and `monobeast`[6] and the reference PPO implementation in CleanRL[44]. As a base comparison, we used an A100 GPU and 10 CPU setting since 10 actor CPU is the default `moolib`'s setting. We also conducted experiments under the 8 A100 setting, where Cleanba's variants use 50 CPUs. The `moolib` and `monobeast` experiments use 80 CPUs[7].

[6]We wanted to test out IMPALA's official source code released in `deepmind/scalable_agent`, but it was built with `tensorflow 1.x` which does not support the A100 GPU tested in this paper.

[7]By default, `moolib` uses 256 environments, 10 actor CPUs, and a single GPU. We followed the recommended scaling instructions to add 8 training GPU-powered peers, which in total used 2048 environments, 80 actor CPUs, and 8 GPUs. While the training time was reduced to about 27 minutes, sample efficiency dropped, and it obtained a

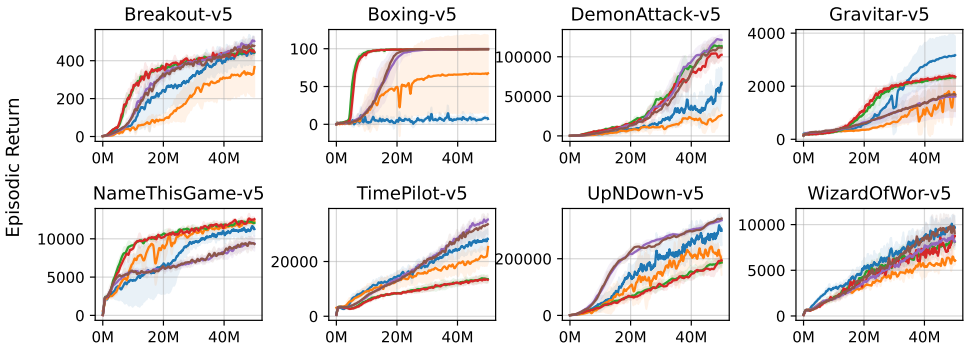


Figure 5.4: The episodic returns of Cleanba variants compared with `moolib`.

All experiments used 84×84 images with greyscale, an action repeat of 4, 4 stacked frames, and a maximum of 108,000 frames. We followed the recommended Atari evaluation protocol by Machado et al.[47], which used sticky action with a probability of 25%, no loss of life signal, and the full action space. Throughout all experiments, the agents used the IMPALA's Resnet architecture[34], ran for 200M frames with three random seeds[8]. The hyperparameters and the learning curves can be found in Appendix C.1.

Figure 5.3 shows the experiment results, such as median HNS learning curves, interquartile mean (IQM) learning curves, performance profile, and 95% stratified bootstrap confidence intervals for the mean, median, IQM, and optimality gap (the amount by which the algorithm fails to meet a minimum normalized score of 1)[1].

5.7.1 Comparison with `moolib`'s IMPALA

Under the 10 CPU and 1 A100 setting, Cleanba's IMPALA obtains a similar level of median HNS as `moolib` and is slightly faster. Cleanba's PPO obtains a higher median HNS but takes longer training time, likely due to the longer training step time spent on reusing rollout data 4 times.

Under the 8 GPU setting, we found Cleanba's IMPALA and PPO can match or exceed `moolib`'s IMPALA's scores in 30% less wall time. The CIs for the median are large, especially for Cleanba's IMPALA and `moolib`'s IMPALA. However, we found the Cleanba's variants to obtain higher interquartile mean (IQM) and mean human-normalized scores. Additionally, the performance profile reveals that Cleanba's PPO has better performance variability across 57 Atari games than Cleanba's IMPALA and `moolib`'s IMPALA.

Cleanba's variants are also more reproducible. We examined the individual learning curves and produced Figure 5.4. We found that Cleanba's variants are much more likely to produce identical learning curves, whereas `moolib`'s learning curves can be much more unpredictable in two hardware settings. Nevertheless, we noticed the learning curves were not smooth in around 10 environments, which probably had more difficult reward surfaces[101]. However, they ultimately caused a high variance in Cleanba's IMPALA's final median HNS.

5.7.2 Comparison with `monobeast`'s IMPALA

The `monobeast` experiments are interesting in several ways. First, it produces a higher median HNS than `moolib`'s IMPALA, which is the opposite of what was shown in Mella et al.[93]. This is probably because Mella et al.[93] used "comparable environment settings" instead of the same environment

catastrophic 28.51% median HNS after 200M frames. We suspected the drop was due to the 2048 environments used, so we set the total number of environments back to 256. Furthermore, we did not restrict `moolib` to use 50 CPUs because we worried it might change the learning behaviors due to the issues mentioned in Section 5.5, so we kept the default scaling to 80 CPUs. For comparison with `moolib`, `monobeast` experiments also use 80 CPUs.

[8]For the `moolib` experiment, we conducted two sets of 3 random seeds. We reported the results with higher IQM and lower median. See Appendix C.2.

settings used in our experiments. Second, the `monobeast` experiments appear robust in two different hardware settings in practice, despite the reproducibility issues we showed in Section 5.5.2.

While `monobeast` obtained high scores, it is significantly slower in the 1 A100 and 10 CPU settings due to poor GPU utilization. It also does not work with more than a single-GPU setting and should scale less efficiently with larger networks because actor threads only run on CPUs when compared to `moolib` and Cleanba's variants.

5.7.3 Comparison with CleanRL's PPO

CleanRL's PPO showed significantly better data efficiency, likely due to the fact that it always learns from the most recent policy, unlike Cleanba's architecture which learns from the second most recent policy. To further study this, we can simulate the synchronous architecture with Cleanba's variants by commenting out line 7 (`if i != 2:`) in Figure 5.1. When done so, we noticed Cleanba's PPO (Sync) achieved similar data efficiency as CleanRL's PPO at the cost of being around 50% slower, showing that Cleanba's PPO can act as a multi-GPU alternative to CleanRL's PPO. However, we noticed Cleanba's IMPALA (Sync) did not benefit from increased data efficiency when using the synchronous architecture.

5.8 Conclusion

In this paper, we have presented Cleanba, a distributed deep reinforcement learning platform that prioritizes reproducibility. Our analysis shows that the existing actor-learner framework can cause reproducibility issues due to the non-determinism arising from the concurrent scheduling of threads and algorithmic issues. To address those issues, we propose a more principled architecture that clarifies which policies are used to create rollout data, providing a solid foundation for reproducibility.

Our Atari experiments demonstrate that Cleanba's PPO and IMPALA achieve equivalent or better data efficiency comparable to `moolib`'s IMPALA but with 30% less wall time under the 8 GPU setting. Furthermore, Cleanba's variants are more likely to produce identical learning curves across different hardware configurations. We also showed that Cleanba's variants can scale to more accelerators and train agents faster than `monobeast`. We believe that Cleanba will be a valuable platform for the research community to conduct further open-source distributed RL research that has a more principled approach to reproducibility.

Part II

Efficient Deep Reinforcement Learning Testbeds and Techniques

Chapter 6: Game Representation in RTS Games

Part II is about designing efficient DRL testbeds and techniques. In this chapter, we begin our journey by creating a DRL interface for the classic real-time strategy (RTS) game simulator μRTS, which will serve as a testbed for researching efficient DRL techniques in Chapter 7-9. While we study these efficient techniques in the context of μRTS, the concepts can be applied to other domains.

Specifically, this chapter presents an initial study comparing different observation and action space representations for Deep Reinforcement Learning (DRL) in the context of Real-time Strategy (RTS) games. Specifically, we compare two representations: (1) a *global* representation where the observation represents the whole game state, and the RL agent needs to choose which unit to issue actions to, and which actions to execute; and (2) a *local* representation where the observation is represented from the point of view of an individual unit, and the RL agent picks actions for each unit independently. We evaluate these representations in μRTS showing that the local representation seems to outperform the global representation when training agents with the task of harvesting resources. The work of this chapter is based on the following workshop paper:

- Shengyi Huang and Santiago Ontañón. Comparing observation and action representations for deep reinforcement learning in μrts. *AIIDE Workshop on Artificial Intelligence for Strategy Games*, 2019

While in this chapter we concluded that the agent using the global representation performs poorly, in hindsight, this conclusion no longer holds significant weight in the light of the invalid action masking technique presented in Chapter 7. The agent that learns under the global representation struggles to collect any resources because the search space is made artificially too large. Once we incorporate invalid action masking, the agent can learn much more efficiently under the global representation, which has other benefits such as more complete information for the agent and hence posing a higher learning potential.

6.1 Motivation

Real-time strategy (RTS) games pose a significant challenge for artificial intelligence (AI) [29;30]. They are complex due to a variety of reasons: (1) players need to issue actions in real-time, which means agents have a very limited time to produce what is the next action to execute, (2) most RTS games are partially observable, i.e., a player might not always able to observe the opponents' strategies and actions, and (3) RTS games have very large action spaces.

Recent application of Deep Reinforcement Learning (DRL) to RTS games introduces additional challenges such as (1) dealing with extremely sparse rewards and (2) designing efficient observation and action space representations. In this document, we attack the latter challenge by comparing two intuitive representations. The first one is a *global* representation that enables the DRL agent to observe the entire game state as a series of features, and issue global commands to choose both which unit to issue actions to and which actions to execute. The second one is a *local* representation where the DRL agent sequentially selects actions for individual units independently. We used μRTS as our testbed to evaluate the performance of these representations and our experiments show that: (1) the local representation seems to outperform global representation, probably because the agent using local representation does not have to learn how to select a unit; (2) both local and global representation significantly outperform a random baseline agent.

6.2 Background

Real-time Strategy (RTS) games are complex adversarial domains, typically simulating battles between many military units, that pose a significant challenge to both human and artificial intelli-

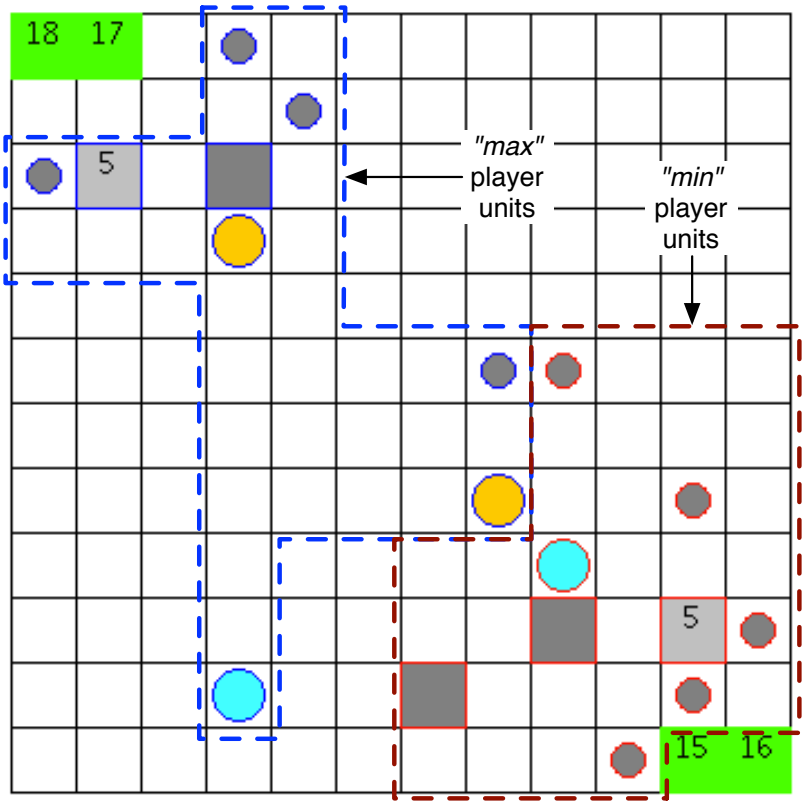


Figure 6.1: A screenshot of μRTS. Square units are "bases" (light grey, that can produce workers), "barracks" (dark grey, that can produce military units), and "resources mines" (green, from where workers can extract resources to produce more units), the circular units are "workers" (small, dark grey) and military units (large, yellow or light blue).

gence[29]. Designing AI techniques for RTS games is challenging because (1) they have *huge decision spaces*: the branching factor of a typical RTS game, StarCraft, has been estimated to be on the order of 10^{50} or higher[30]; (2) they are *real-time*, which means that these games typically execute at 10 to 50 decision cycles per second, leaving players with just a fraction of a second to decide the next action, players can issue actions simultaneously, and actions are durative, and (3) most of them are partially observable and players must sent units to scout the map.

In the experiments reported in this document, we employed μRTS[1], a simple RTS game maintaining the essential features that make RTS games challenging from an AI point of view: simultaneous and durative actions, large branching factors and real-time decision making. Although the game can be configured to be partially observably and non-deterministic, those settings are turned off for all the experiments presented in this document.

A screenshot of the game is shown in Figure 6.1. The squared units in green are Minerals with numbers on them indicating the remaining resources. The units with blue outline belong to player 1 and those with red outline belong to player 2. The light grey squared units are Bases with numbers indicating the amount of resources owned by the player, while the darker grey squared units are the Barracks. Movable units have round shapes with grey units being Workers, orange units being Lights, yellow being Heavy units (not shown in the figure) and blue units being Ranged.

Moreover, to ease reinforcement learning research in μRTS, we have prepared an OpenAI gym wrapper for the game, which we have made available to the research community[2].

One of the earliest applications of reinforcement learning to RTS games is the work of Concurrent

[1] https://github.com/santiontanon/microrts
[2] https://github.com/vwxyzjn/gym-microrts

Hierarchical Reinforcement Learning (CHRL) with Q-learning — Marthi et al. experimented with the Wargus game and demonstrated the use of Alisp language to specify a list of desired tasks and use Q-learning to tune the parameters[102]. They showed that the CHRL could significantly improve the agents' performance compared to flat concurrent hierarchical Q-learning.

In addition, Jaidee and Muñoz-Avila have leveraged Q-learning to learn a policy for each class of units (peasants, knights, barracks, etc), which drastically reduced the memory requirement of Q-learning[103]. The trade-off is the lack of coordination between agents since each agent only learns to control a certain type of units or buildings. That being said, they were able to demonstrate good performance by defeating the built-in AI 80% of the time.

Another idea explored in RTS games is that of *options*, which are temporally extended actions[104]. The agent first needs to evaluate which option to choose, and then needs to determine when to terminate the options chosen. The options framework also exhibits a sense of Hierarchical Reinforcement Learning (HRL) since raw actions are primitives compared to options. Researchers have tried to combine options and heuristic algorithms to simplify the game space[105].

In recent years, *Deep Reinforcement Learning (DRL)* approaches have received significant attention in a number of games such as classic Atari games[11] and Go[106]. Over the years, researchers have introduced different type of DRL algorithms. Most notably, Asynchronous Advantage Actor-Critic (A3C), Deep Q-Network (DQN), and Proximal Policy Gradient (PPO) have gained the state-of-the-art results in a variety of games[11;15;16].

In 2017, researchers at Deepmind started tackling the challenge of training agents for StarCraft II, one of the most complex and popular RTS games in history[107]. They conducted a series of experiments using A3C[15] on the full game as well as a collection of mini-games such as training the "marines" units to defeat the "roaches" units in StarCraft II. They were able to show good converging performance for the mini-games but had no material success in the full game setting. However, they recently demonstrated a bot "AlphaStar" that defeats a professional StarCraft II Protoss player in the full game setting.

Researchers at Facebook also joined this line of research by publishing an open-sourced RTS game research platform ELF[108]. For simplicity, they conducted experiments utilizing hard-coded actions such as BUILD_BARRACKS, which "automatically picks a worker to build barracks at an empty location, if the player can afford". Through these high-level actions, they trained agents using A3C, curriculum training, MCTS, and show the trained agent is able to defeat the built-in scripted bots in the full-game setting. Notably, researchers at Tencent used similar approach to hard-code high-level actions in StarCraft II and defeated the cheating AI in the full-game setting by commencing early attacks[67]. Compared to this work, we are interested in RL settings that do not require hand-crafted macro actions.

Moreover, Liu et al. studied the application of HRL to incorporate the generation of macro actions through expert replays[109]. Lee et al. also hard-coded a collection of macro actions for zergs and trained agents combining DQN and LSTM, obtaining a 83% winning rate in the AIIDE 2017 StarCraft AI competition[110].

To address the action coordination problem, Samvelyan, Rashid et al. built the StarCraft Multi-Agent Challenge (SMAC) for Multi-agent Reinforcement Learning (MARL), where each unit is controlled by a separate agent[111]. They compared popular MARL algorithms such as QMIX, COMA, VDN, and IQL in mini maps that usually features battle between different agents[112–115].

Though these previous research show important results, there has not been any published study comparing the relative merits of different types of observation and action representations in RTS games, which is the key issue that our paper starts to address.

6.3 Gym-µRTS: Comparing Game Representations

We designed two pairs of observation and action representations for comparing performance. The first one is a global representation that enables the RL agent to observe the entire game and issue global commands to choose both which unit to issue actions to and which actions to execute. The second one is a local representation where the agent is externally given a specific unit to control and

Table 6.1: The list of feature maps and their descriptions.

Global Representation	
Features	Possible values
Hit Points	0, 1, 2, 3, 4, 5, ≥ 6
Resources	0, 1, 2, 3, 4, 5, ≥ 6
Owner	player 1, -, player 2
Unit Types	-, resource, base, barrack, worker, light, heavy
Action	-, move, harvest, return, produce, attack

Local Representation	
Features	Possible values
Hit Points	wall, 0, 1, 2, 3, 4, 5, ≥ 6
Resources	wall, 0, 1, 2, 3, 4, 5, ≥ 6
Owners	wall, player 1, -, player 2
Unit Types	wall, -, resource, base, barrack, worker, light, heavy
Action	wall, -, move, harvest, return, produce, attack

just has to pick actions for that unit. Both representations are designed to train agents to harvest resources as fast as possible.

6.3.1 Global Representation

We utilized *flattened one-hot* feature maps as our observation representation similar to PySC2[107] and a couple pre-existing works on observation evaluation functions on μRTS[116]. Assuming the game map is represented as a grid, a feature map is a matrix of the same dimensions as the game map, where in each cell of the matrix we have a value that represents some aspect of the corresponding game map grid cell. For simplicity, we assume that each feature map takes discrete values. The exhaustive list of feature maps and their description is presented at Table 6.1.

Let h and w be the height and width of the map respectively, we construct the *observation* as a set of $n_f = 5$ feature maps and that all feature maps can take the same number of values n_c. Thus, each feature map is a vector of length $h * w$, and the observation can be represented as a $n_f \times (h * w)$ matrix. Finally, in order to facilitate learning, we use a one-hot-encoding representation, and thus the observation is represented as a $n_f \times (h * w) \times n_c$ tensor (n_c is the number of values of the feature plane that can take the larger number of values). The dimensions of these tensors are usually referred to as their *shape* in common deep learning libraries, and thus we will use this term from now on. Moreover, in the experiments reported in this document $n_f = 5$ and $n_c = 7$.

Then, to execute an action in μRTS, the RL agent needs to specify the unit, the action types, and the parameters of the action to execute. For our experiments that only focus on the task of harvesting resources, the set of target action types are NOOP (no operation, represented simply as a - in Table 6.1), Move, Harvest, Return. Some of these action types requires an action parameter that specifies the direction at which the action is issued. The list of available action parameter is Up, Right, Down, Left. So if the agent wants to command a worker to harvest the resources located left to it, the agent has to specify the worker, set action type to Harvest, and action parameter to Left.

Since unit IDs change from game to game, in order to learn generalizable policies, the global representation does not require the agent to specify the unit ID. Instead, the RL agent predicts the coordinates of the unit it wants to control.

Therefore, we construct the *action* at time step t as a vector with 4 elements: $\left[a_t^{\mathrm{x}}, a_t^{\mathrm{y}}, a_t^{\text{action type}}, a_t^{\text{action param}}\right]$, where $a_t^{\mathrm{x}}, a_t^{\mathrm{y}}, a_t^{\text{action type}}, a_t^{\text{action param}}$ are integers signifying the selected x-coordinate, y-coordinate, action type, and action parameters, respectively, predicted using one-hot encoding representations. Moreover:

1. If the action produced is invalid in μRTS, the action will be replaced by a NOOP action. Thus,

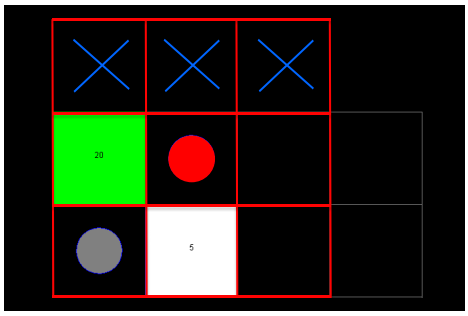


Figure 6.2: The local feature maps of shape $(2w+1) \times (2w+1) = 3 \times 3$ outlined in red of the an unit (the red circle) when $w = 1$. The blue crosses indicate the cells are the walls of the game map.

it is up to the RL agent to learn which are the valid actions.

2. This representation varies from PYSC2's action representation that uses the actions themselves to select units and perform actions on them.

3. The agent can only issue one action to one unit at each time step.

In summary, the global representation encodes the entire game state for the RL agent, who needs to learn to choose which unit to control (by outputting the unit's coordinates) and which action type and parameter to be issued at each time step.

6.3.2 Local Representation

In this representation, the agent perceives the game from the point of view of a particular unit and only needs to issue action to that unit. In other words, the agent does *not* have to learn to pick which unit to control.

We utilized flattened one-hot *local* feature maps as our observation representation. Consider a parameter of window size w that specifies "how far can an unit see". Specifically, a window size $w = 1$ means each unit is able to observe the cell it is located and all the cells that are 1 distance away from it in either axis, as shown in Figure 6.2. Note that if the unit sees a cell beyond the boundary of the map, we consider that cell a "Wall" (Note that the global representation does not need such "Wall" unit type because the agent always sees the entire map and does *not* see beyond the boundary of the map). A local feature map is thus a matrix of dimension $(2w+1) \times (2w+1)$. A comprehensive list of local feature maps and their descriptions can be found in Table 6.1.

At each time step, the agent is given a target unit (we rotate which unit is the focus at consecutive game frames). For instance, assuming there are 3 units u_1, u_2, u_3, then $u_1, u_2, u_3, u_1, u_2, ...$ will be selected at time step $1, 2, 3, 4, 5, ...$, respectively. Given the selected unit at each time step, we construct *observation* to be a set of n_f local feature maps from the point of view of the said unit according to w, with all of them can take the same number of discrete values n_c. Similarly, we use one-hot-encoding representation and the observation is represented as a $n_f \times (2w+1)^2 \times n_c$ tensor, where $n_f = 5, n_c = 8$.

For action execution, since the unit is already selected, the agent only needs to predict the action type and parameter. Therefore, we construct the *action* at time step t as a vector with 2 element: $\left[a_t^{\text{action type}}, a_t^{\text{action param}}\right]$, where $a_t^{\text{action type}}, a_t^{\text{action param}}$ are integers that signifies the selected action type, and action parameters, respectively (predicted as one-hot vectors). As before, if the action issued is not valid in μRTS, it will be replaced by a NOOP action.

In summary, the local representation contains the local game state features from the point of view of a selected unit for the RL agent, and it needs to learn which action type and parameter to be issued at each time step. Note that it does *not* have to learn how to select a unit to control, but only how to interact with its surrounding observations.

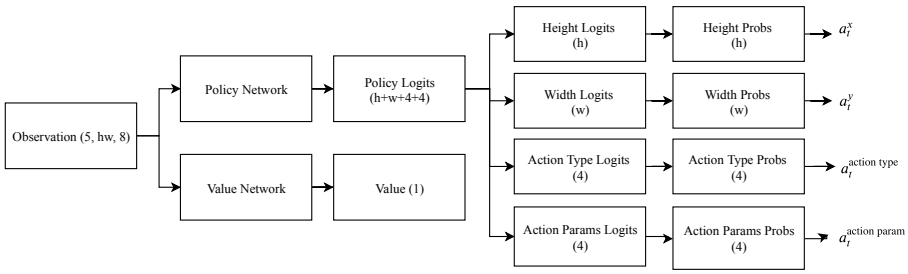


Figure 6.3: The neural network architecture that demonstrates the flow from the observation vector to action probabilities. The number at each box suggests the input or output shapes

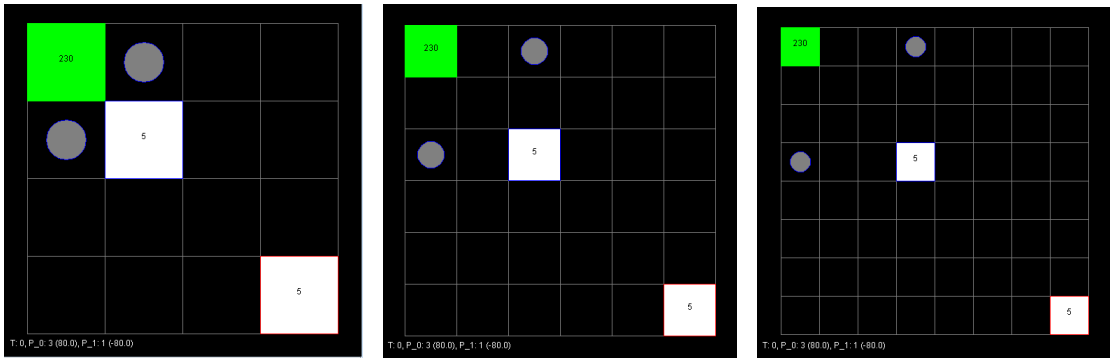


Figure 6.4: The mini-games that focuses on harvesting resources with different map sizes of 4×4, 6×6, and 8×8.

6.3.3 Reward Function

The game environment will give the agent a reward of 10 when a worker has successfully harvested resources and another 10 when it returns the harvested resources back to the base. Otherwise the game environment gives a reward of 0.

It's worth pointing out that, like in many other reinforcement learning problems, rewards can be very sparse as maps grow and workers get further away from the resources and the base, requiring thousands of exploration steps before getting a non-zero reward.

6.4 Experimental Study

For the global representation, we selected Advantage Actor-Critic (A2C), one of the most popular RL algorithms, to evaluate the agents' performance[15]. We created the Policy Network and Value Networks. As shown in Figure 6.3 that demonstrates the forward pass, the observation tensor is fed into the policy network that produces a dense vector of shape $(h \cdot w \cdot n_a \cdot n_p)$, which is split into 4 different subvectors of logits. Each of those logits is passed into a Categorical Distribution to gain the probability distribution for sampling values to produce the a combined action vector $a = \begin{bmatrix} a_t^{\text{x}} & a_t^{\text{y}} & a_t^{\text{action type}} & a_t^{\text{action param}} \end{bmatrix}$. In addition, the observation vector is also used by the value network to create a valuation of the current observation.

Given the forward pass, we run the game for a full episode, record the reward r_t at each time

Table 6.2: The list of experiment parameters and their values.

Parameter Names	Parameter Values
Maps	4×4, 6×6, 8×8
Learning Rate of Adam Optimizer	0.0007
Random Seed	1, 2, 3
Episode Length	2,000 time steps
Total Time steps	2,000,000 time steps
w (Local Representation Window Size)	1, 2
γ (Discount Factor)	0.99
c_1 (Value Function Coefficient)	0.25
c_2 (Entropy Regularization Coefficient)	0.01
ω (Gradient Norm Threshold)	0.5

step t and compute

$$\begin{aligned}\log \pi_\theta(a_t|s_t) = &\log \pi_\theta(a_t^{\text{x}}|s_t) \\ &+ \log \pi_\theta(a_t^{\text{y}}|s_t) \\ &+ \log \pi_\theta(a_t^{\text{action type}}|s_t) \\ &+ \log \pi_\theta(a_t^{\text{action param}}|s_t)\end{aligned}$$

where s_t is the observation tensor at time step t. In addition, analogously compute the entropy $\sum_a \pi_\theta(a|s_t) \log \pi_\theta(a|s_t)$ as the sum of entropy for $a_t^{\text{x}}, a_t^{\text{y}}, a_t^{\text{action type}}$ and $a_t^{\text{action param}}$. After each episode finishes, we could train the agent based on the following gradient

$$\underbrace{(G_t - v_{\theta'}(s_t))\nabla_\theta \log \pi_\theta(a_t|s_t)}_{\text{policy gradient}} +$$

$$\underbrace{c_1(G_t - v_{\theta'}(s_t))\nabla_{\theta'} v_{\theta'}(s_t)}_{\text{value estimation gradient}} +$$

$$\underbrace{c_2 \sum_a \pi_\theta(a|s_t) \log \pi_\theta(a|s_t)}_{\text{entropy regularization}}$$

where θ' and θ are weights for the value network and the policy network, respectively, c_1 and c_2 are hyperparameters for controlling the contribution of value gradients and entropy regularization gradients, and G_t is the discounted rewards from step t. A comprehensive list of training parameters and their values can be found at Table 6.2. For more details on the algorithm, please refer to the original paper[15].

Regarding the local representation, we used a similar Policy Network architecture that produces a dense vector of shape $(n_a \cdot n_p)$. We similarly calculate

$$\log \pi_\theta(a_t|s_t) = \log \pi_\theta(a_t^{\text{action type}}|s_t) + \log \pi_\theta(a_t^{\text{action param}}|s_t)$$

and the entropy $\sum_a \pi_\theta(a|s_t) \log \pi_\theta(a|s_t)$ as the sum of entropy for $a_t^{\text{action type}}$ and $a_t^{\text{action param}}$; then use A2C to train the agent.

In this section, we compare the aforementioned approaches on maps of different sizes.

6.4.1 Experimental Setup

We created three maps of size 4×4, 6×6, and 8×8, all of which are shown in the Figure 6.4. They all have two workers, a block containing 230 resources, a base for the workers to return the harvested resources, and a dummy enemy base that is useless. Note that it takes a worker 10 time

Table 6.3: The list of representations and their performance according to our metrics. The "-" in $t_{\text{first return}}$ indicates the agent never returned any resources.

	map	$t_{\text{first harvest}}$	$t_{\text{first return}}$	r
RandomAI	4×4	51.33	142.67	11.87
Global	4×4	99.00	167.73	13.13
Local ($w = 1$)	4×4	29.87	172.47	67.20
Local ($w = 2$)	4×4	45.00	73.73	33.40
RandomAI	6×6	421.33	797.33	2.00
Global	6×6	533.33	1931.20	0.07
Local ($w = 1$)	6×6	59.20	567.40	3.53
Local ($w = 2$)	6×6	62.33	408.73	3.93
RandomAI	8×8	878.67	1480.67	0.87
Global	8×8	1464.53	-	0.00
Local ($w = 1$)	8×8	167.20	1844.20	0.20
Local ($w = 2$)	8×8	89.87	-	0.00

steps to harvest a resource and another 10 time steps to return the resources to the base, excluding the time steps to move around the map. So in the most efficient harvesting setup as shown in the 4×4 map, where each worker is right next to the base and the resources, the most resources that the two workers can gather is 2 per 20 frames. In fact, since both representations only accept one action at each time step, the two workers have to wait until 21 frames for the resources to be returned. Simple math shows that $2 \cdot 2000/20 = 200$ is the most that the two workers can harvest within the 2,000 time steps per episode. A comprehensive list of experimental parameters and their values are presented in Table 6.2. For values of hyperparameters, we simply used the default values from the OpenAI's A2C implementation [33].

6.4.2 Experimental Results

Figure 6.5 shows the average reward of runs with different random seeds per episode as a function of training steps. As shown in the plots, local representation with $w = 1$ yields the best episode rewards in all three maps. As mentioned previously, we suspect local representation is performing better just because the agent doesn't have to learn to pick a unit as the agent does under the global representation, i.e., the units are pre-selected under the local representation.

The reason that local representation with $w = 2$ is performing worse than local representation with $w = 1$ seems purely computational since the observation tensor when $w = 2$ is almost twice as the size when $w = 1$. The other notable characteristic is that local representation with $w = 2$ seems to exhibit more stable performance. Given more computational resources, we think local representation with $w = 2$ will eventually become more efficient than local representation with $w = 1$ because larger window size means the units are less likely to move to a cell where the base or resources are outside of its local feature maps (or in simpler terms, the unit moved to a cell that could not observe the base or resources). In our experiments, the map sizes are generally small and the position of the base and resources are stationary, so the local representation with $w = 1$ is probably enough for the unit to observe sufficient information (and for learning to reach the resources, they just need to learn to move to the top-left coordinate of the map).

Notably, the performance of both local and global representation suffer when we scale the game with larger maps. As shown in the Figure 6.4, the larger maps presents even challenging moving trajectories for the agents since the distance between resources and the base is increased, which means sparser rewards and larger exploration needed for the agents. Global representation is especially worse off because the agent still has to learn to select units, navigate, harvest, and return to base in the larger maps.

As a baseline, we created three metrics to evaluate the performance of agents against the built-in RandomAI μRTS agent that uniformly selects a valid action, either NOOP, Move, Harvest, or Return, and a valid parameter depending on the game state. Notice that all actions produced by

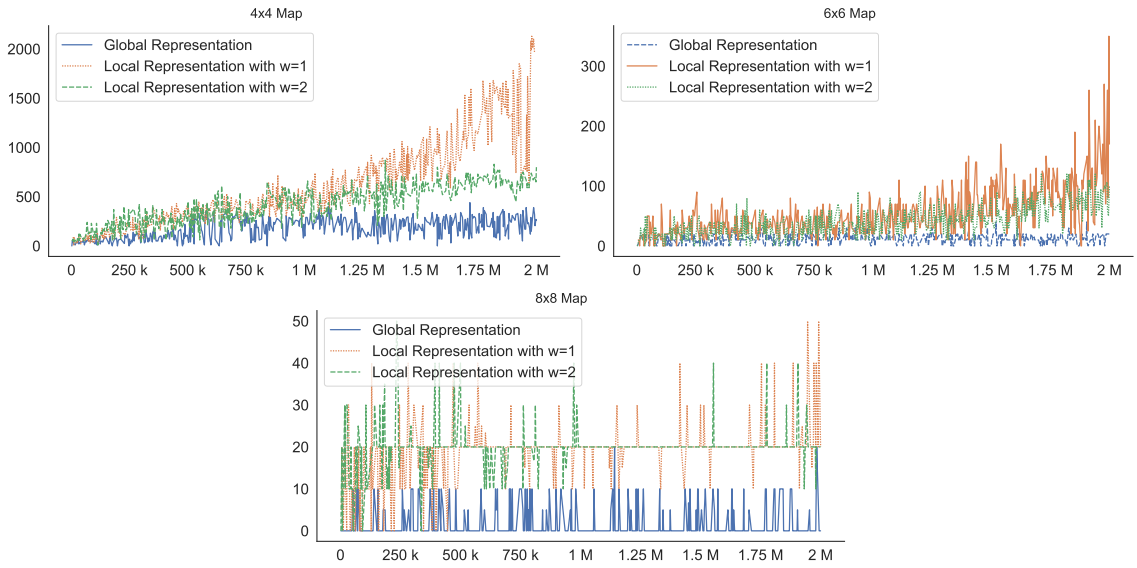


Figure 6.5: Episode rewards (y axis) as a function of training time steps (x-axis) for the 3 map sizes.

the RandomAI agent are valid since it has the game rules hard-coded, whereas our reinforcement learning agents have to learn which actions are valid. Thus, the RandomAI agent has an unfair advantage. A way to level the playing field would be to mask out invalid agents in the output of the reinforcement learning, and consider the action with highest likelihood only among the valid ones. However, we decided against giving our agent any domain knowledge beyond the observation and action representation in this first set of experiments, just to set a baseline.

Let $t_{\text{first harvest}}$ and $t_{\text{first return}}$ be the time steps that the agent successfully harvested and return resources for the first time, respectively, which will give us an idea of how much "wandering" did the agent did before harvesting and returning. Then, let r be the number of resources gathered in total, which will evaluate the agents' performance as a whole. We ran the trained agents with different seeds for 5 episodes (10,000 time steps) and calculate the average $t_{\text{first harvest}}$, $t_{\text{first return}}$, and r across runs with different seeds. The evaluation result is listed at Table 6.3. As shown, the trained agents perform significantly better than the RandomAI agent except in the 8×8 map where all agents perform equally bad.

6.4.3 Visual Behavior of Agents

When using RL to train agents to play games, it is easy to be distracted by the numerical rewards and various metrics. However, watching the agent play in the game remains the ultimate measure for agents' ability. The following paragraphs highlight some interesting behaviors of agents.

In 4×4 maps, the agent under the local representation almost exhibits optimal behavior where the two Workers harvest the resources and return them restlessly. The agent under the global representation, however, seems to only learn to control *one* Worker. It seems the agent gets stuck in a local minimum where it believes the maneuver of only one Worker is best for harvesting resources.

In 6×6 and 8×8 maps, the agent under the local representation struggles with returning the resources to the base. In fact, the agent tries to harvest another unit of resources even though it already carries one. The unit moves closer to the resources than to the base, but unfortunately it has to return the resources before harvesting new ones. It almost seems that the agent learned only to harvest resources, but has not learned how to return them.

6.5 Discussion

In this document, we compared two different observation and action representations in the context of RTS games: (1) a global representation that feeds the agent an observation of feature maps that are as large as the game map itself, and require the agent to learn to locate unit it is wants to issue actions to and predict the intended action type and parameter; and (2) a local representation that feeds the agent an observation of local feature maps that are the feature maps of some distance away from the point of view of a selected unit, and require the agent to learn which action type and parameter to issue for the said unit. We train agents on maps that focus on harvesting resources, establish some objective metrics to evaluate the agents' performance and show that the local representation generally outperforms the global representation. This advantage, however, does not necessarily hold in larger maps, where the exploration and sparse rewards become a huge problem.

Chapter 7: Invalid Action Masking

In the last chapter, we built the Gym-μRTS testbed for us to experiment and study efficient DRL techniques. We soon realized the agent in the last chapter performed poorly because the agent's action space is too large. For example, in an 8×8 map with 2 worker units, the chance of the agent choosing a valid worker unit is $2/(8 * 8) = 3.12\%$, so for 97% of the time the agent will be issuing invalid actions. Upon reviewing related work, we found recent work to employ a technique called invalid action masking[107], which can help avoid sampling invalid actions. However, invalid action masking was an anecdotal technique mentioned in previous literature with no theoretical analysis or empirical ablation study. In this chapter, we take a deep dive into this technique and help the research community understand its mechanism and effect better. The work of this chapter is based on the following publication:

- Shengyi Huang and Santiago Ontañón. A closer look at invalid action masking in policy gradient algorithms. volume 35, May 2022. doi: 10.32473/flairs.v35i.130584. URL https://journals.flvc.org/FLAIRS/article/view/130584

7.1 Motivation

Deep Reinforcement Learning (DRL) algorithms have yielded state-of-the-art game-playing agents in challenging domains such as Real-time Strategy (RTS) games[27;107] and Multiplayer Online Battle Arena (MOBA) games[66;117]. Because these games have complicated rules, the valid discrete action spaces of different states usually have different sizes. That is, one state might have 5 valid actions and another state might have 7 valid actions. To formulate these games as a standard reinforcement learning problem with a singular action set, previous work combines these discrete action spaces to a *full discrete action space* that contains available actions of all states[66;107;117]. Although such a full discrete action space makes it easier to apply DRL algorithms, one issue is that an action sampled from this full discrete action space could be invalid for some game states, and this action will have to be discarded.

To make matters worse, some games have extremely large full discrete action spaces and an action sampled will typically be invalid. As an example, the full discrete action space of Dota 2 has a size of 1,837,080[66], and an action sampled might be to buy an item, which can be *valid* in some game states but will become *invalid* when there is not enough gold. To avoid repeatedly sampling invalid actions in full discrete action spaces, recent work applies policy gradient algorithms in conjunction with a technique known as invalid action masking, which "masks out" invalid actions and then just samples from those actions that are valid[66;107;117]. To the best of our knowledge, however, the theoretical foundations of invalid action masking have not been studied and its empirical effect is under-investigated. In this paper, we take a closer look at invalid action masking in the context of games, pointing out the gradient produced by invalid action masking corresponds to a valid policy gradient. More interestingly, we show that in fact, invalid action masking can be seen as applying a *state-dependent differentiable function* during the calculation of the action probability distribution, to produce a behavior policy. Next, we design experiments to compare the performance of *invalid action masking* versus *invalid action penalty*, which is a common approach that gives negative rewards for invalid actions so that the agent learns to maximize reward by not executing any invalid actions. We empirically show that, when the space of invalid actions grows, invalid action masking scales well and the agent solves our desired task while invalid action penalty struggles to explore even the very first reward. Then, we design experiments to answer two questions: (1) What happens if we remove the invalid action mask once the agent was trained with the mask? (2) What is the agent's performance when we implement the invalid action masking naively by sampling the action from the masked action probability distribution but updating the policy gradient using the unmasked action

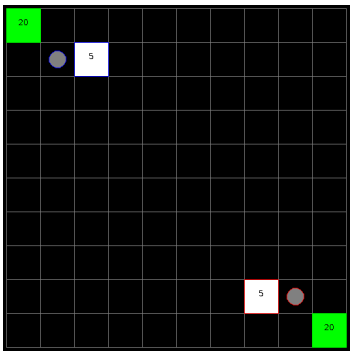


Figure 7.1: A screenshot of μRTS. Square units are "bases" (light grey, that can produce workers), "barracks" (dark grey, that can produce military units), and "resources mines" (green, from where workers can extract resources to produce more units), the circular units are "workers" (small, dark grey) and military units (large, yellow or light blue), and on the right is the 10×10 map we used to train agents to harvest resources. The agents could control units at the top left, and the units in the bottom left will remain stationary.

probability distribution? Finally, we made our source code available at GitHub for the purpose of reproducibility[1].

7.2 Background

There have been other approaches to deal with invalid actions. Dulac-Arnold, Evans, et al.[118] suggest embedding discrete action spaces into a continuous action space, using nearest-neighbor methods to locate the nearest valid actions. In the field of games with natural language, others propose to train an Action Elimination Network (AEN)[119] to reduce the action set.

The purpose of avoiding executing invalid actions arguably is to boost exploration efficiency. Some less related work achieves this purpose by reducing the full discrete action space to a simpler action space. Kanervisto, et al.[120] describes this kind of work as "action space shaping", which typically involves 1) action removals (e.g. Minecraft RL environment removes non-useful actions such as "sneak"[121]), and 2) discretization of continuous action space (e.g. the famous CartPole-v0 environment discretize the continuous forces to be applied to the cart[50]). Although a well-shaped action space could help the agent efficiently explore and learn a useful policy, action space shaping is shown to be potentially difficult to tune and sometimes detrimental in helping the agent solve the desired tasks[118].

Lastly, Kanervisto, et al.[120] and Ye, et al.[117] provide ablation studies to show invalid action masking could be important to the performance of agents, but they do not study the empirical effect of invalid action masking as the space of invalid action grows, which is addressed in this paper.

7.3 Invalid Action Masking

Invalid action masking is a common technique implemented to avoid repeatedly generating invalid actions in large discrete action spaces[66;107;117]. To the best of our knowledge, there is no literature providing detailed descriptions of the implementation of invalid action masking. Existing work[66;107] seems to treat invalid action masking as an auxiliary detail, usually describing it using only a few sentences. Additionally, there is no literature providing theoretical justification to explain why it works with policy gradient algorithms. In this section, we examine how invalid action masking is implemented and prove it indeed corresponds to valid policy gradient updates[14]. More interestingly,

[1] https://github.com/vwxyzjn/invalid-action-masking

we show it works by applying a *state-dependent differentiable function* during the calculation of action probability distribution.

First, let us see how a discrete action is typically generated through policy gradient algorithms. Most policy gradient algorithms employ a neural network to represent the policy, which usually outputs unnormalized scores (logits) and then converts them into an action probability distribution using a softmax operation or equivalent, which is the framework we will assume in the rest of the paper. For illustration purposes, consider an MDP with the action set $\mathcal{A} = \{a_0, a_1, a_2, a_3\}$ and $\mathcal{S} = \{s_0, s_1\}$, where the MDP reaches the terminal state s_1 immediately after an action is taken in the initial state s_0 and the reward is always +1. Further, consider a policy π_θ parameterized by $\theta = [l_0, l_1, l_2, l_3] = [1.0, 1.0, 1.0, 1.0]$ that, for the sake of this example, directly produces θ as the output logits. Then in s_0 we have:

$$\begin{aligned}\pi_\theta(\cdot|s_0) &= [\pi_\theta(a_0|s_0), \pi_\theta(a_1|s_0), \pi_\theta(a_2|s_0), \pi_\theta(a_3|s_0)] \\ &= \text{softmax}([l_0, l_1, l_2, l_3]) \\ &= [0.25, 0.25, 0.25, 0.25], \\ &\text{where } \pi_\theta(a_i|s_0) = \frac{\exp(l_i)}{\sum_j \exp(l_j)}\end{aligned} \tag{7.1}$$

At this point, regular policy gradient algorithms will sample an action from $\pi_\theta(\cdot|s_0)$. Suppose a_0 is sampled from $\pi_\theta(\cdot|s_0)$, and the policy gradient can be calculated as follows:

$$\begin{aligned}g_{\text{policy}} &= \mathbb{E}_\tau \left[\nabla_\theta \sum_{t=0}^{T-1} \log \pi_\theta(a_t|s_t) G_t \right] \\ &= \nabla_\theta \log \pi_\theta(a_0|s_0) G_0 \\ &= [0.75, -0.25, -0.25, -0.25]\end{aligned}$$

$$(\nabla_\theta \log \text{softmax}(\theta)_j)_i = \begin{cases} (1 - \frac{\exp(l_j)}{\sum_j \exp(l_j)}) & \text{if } i = j \\ \frac{-\exp(l_j)}{\sum_j \exp(l_j)} & \text{otherwise} \end{cases}$$

Now suppose a_2 is invalid for state s_0, and the only valid actions are a_0, a_1, a_3. Invalid action masking helps to avoid sampling invalid actions by "masking out" the logits corresponding to the invalid actions. This is usually accomplished by replacing the logits of the actions to be masked by a large negative number M (e.g. $M = -1 \times 10^8$). Let us use $mask : \mathbb{R} \to \mathbb{R}$ to denote this masking process, and we can calculate the re-normalized probability distribution $\pi'_\theta(\cdot|s_0)$ as the following:

$$\pi'_\theta(\cdot|s_0) = \text{softmax}(mask([l_0, l_1, l_2, l_3])) \tag{7.2}$$

$$= \text{softmax}([l_0, l_1, M, l_3]) \tag{7.3}$$

$$= [\pi'_\theta(a_0|s_0), \pi'_\theta(a_1|s_0), \epsilon, \pi'_\theta(a_3|s_0)] \tag{7.4}$$

$$= [0.33, 0.33, 0.0000, 0.33]$$

where ϵ is the resulting probability of the masked invalid action, which should be a small number. If M is chosen to be sufficiently negative, the probability of choosing the masked invalid action a_2 will be virtually zero. After finishing the episode, the policy is updated according to the following gradient, which we refer to as the *invalid action policy gradient*.

$$g_{\text{invalid action policy}} = \mathbb{E}_\tau \left[\nabla_\theta \sum_{t=0}^{T-1} \log \pi'_\theta(a_t|s_t) G_t \right] \tag{7.5}$$

$$= \nabla_\theta \log \pi'_\theta(a_0|s_0) G_0 \tag{7.6}$$

$$= [0.67, -0.33, 0.0000, -0.33]$$

This example highlights that invalid action masking appears to do more than just "renormalizing

Table 7.1: Observation features and action components.

Observation Features	Planes	Description
Hit Points	5	0, 1, 2, 3, ≥ 4
Resources	5	0, 1, 2, 3, ≥ 4
Owner	3	player 1, -, player 2
Unit Types	8	-, resource, base, barrack, worker, light, heavy, ranged
Current Action	6	-, move, harvest, return, produce, attack
Action Components	**Range**	**Description**
Source Unit	$[0, h \times w - 1]$	the location of the unit selected to perform an action
Action Type	$[0, 5]$	NOOP, move, harvest, return, produce, attack
Move Parameter	$[0, 3]$	north, east, south, west
Harvest Parameter	$[0, 3]$	north, east, south, west
Return Parameter	$[0, 3]$	north, east, south, west
Produce Direction Parameter	$[0, 3]$	north, east, south, west
Produce Type Parameter	$[0, 6]$	resource, base, barrack, worker, light, heavy, ranged
Attack Target Unit	$[0, h \times w - 1]$	the location of unit that will be attacked

the probability distribution"; it in fact makes the gradient corresponding to the logits of the invalid action to zero.

7.3.1 Masking Still Produces a Valid Policy Gradient

The action selection process is affected by a process that seems external to π_θ that calculates the mask. It is therefore natural to wonder how does the policy gradient theorem [14] apply. As a matter of fact, our analysis shows that the process of invalid action masking can be considered as a state-dependent differentiable function applied for the calculation of π'_θ, and therefore $g_{\text{invalid action policy}}$ can be considered as a policy gradient update for π'_θ.

Proposition 1. $g_{\text{invalid action policy}}$ is the policy gradient of policy π'_θ.

Proof. Let $s \in S$ to be arbitrary and consider the process of invalid action masking as a differentiable function $mask$ to be applied to the logits $l(s)$ outputted by policy π_θ given state s. Then we have:

$$\pi'_\theta(\cdot|s) = \text{softmax}(mask(l(s)))$$

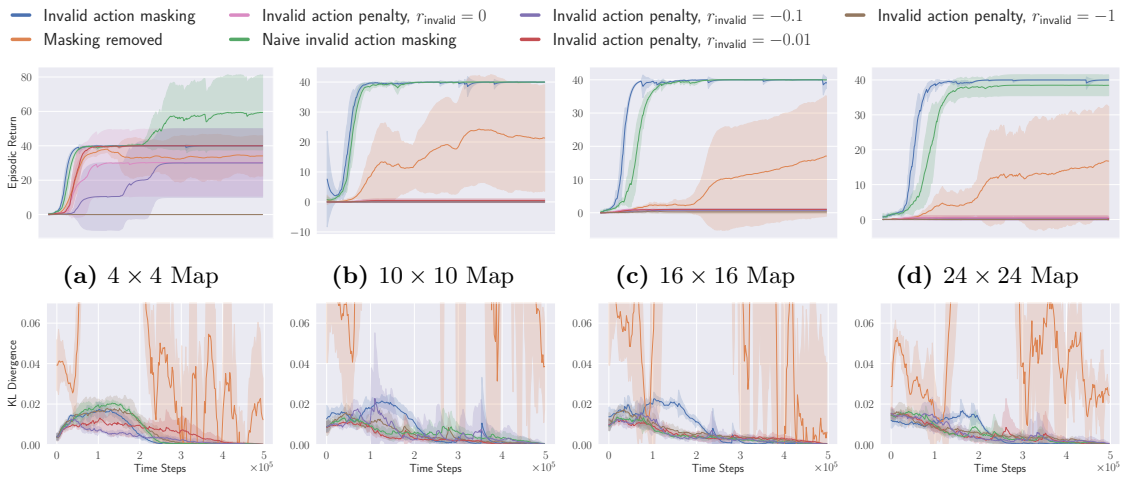
$$mask(l(s))_i = \begin{cases} l_i & \text{if } a_i \text{ is valid in } s \\ M & \text{otherwise} \end{cases}$$

Clearly, $mask$ is either an identity function or a constant function for elements in the logits. Since these two kinds of functions are differentiable, π'_θ is differentiable to its parameters θ. That is, $\frac{\partial \pi'_\theta(a|s)}{\partial \theta}$ exists for all $a \in A, s \in S$, which satisfies the assumption of policy gradient theorem [14]. Hence, $g_{\text{invalid action policy}}$ is the policy gradient of policy π'_θ. □

Note that $mask$ is *not* a piece-wise linear function. If we plot $mask$, it is either an identity function or a constant function, depending on the state s, going from $-\infty$ to $+\infty$. We therefore call $mask$ a state-dependent differentiable function. That is, given a vector x and two states s, s' with different number of invalid actions available in these states, $mask(s, x) \neq mask(s', x)$.

7.4 Experimental Study

In the remainder of this paper, we provide a series of empirical results showing the practical implications of invalid action masking.



(a) 4×4 Map **(b)** 10×10 Map **(c)** 16×16 Map **(d)** 24×24 Map

(e) 4×4 Map **(f)** 10×10 Map **(g)** 16×16 Map **(h)** 24×24 Map

Figure 7.2: The first row shows the episodic return over the time steps, and the second row shows the Kullback–Leibler (KL) divergence between the target and current policy of PPO over the time steps. The shaded area represents one standard deviation of the data over 4 random seeds. Curves are exponentially smoothed with a weight of 0.9 for readability.

7.4.1 Evaluation Environment

We use μRTS[2] as our testbed, which is a minimalistic RTS game maintaining the core features that make RTS games challenging from an AI point of view: simultaneous and durative actions, large branching factors, and real-time decision-making. A screenshot of the game can be found in Figure 7.1. It is the perfect testbed for our experiments because the action space in μRTS grows combinatorially and so does the number of invalid actions that could be generated by the DRL agent. We now present the technical details of the environment for our experiments.

- **Observation Space.** Given a map of size $h \times w$, the observation is a tensor of shape (h, w, n_f), where n_f is a number of feature planes that have binary values. The observation space used in this paper uses 27 feature planes as shown in Table 7.1, similar to previous work in μRTS[31;116;122]. A feature plane can be thought of as a concatenation of multiple one-hot encoded features. As an example, if there is a worker with hit points equal to 1, not carrying any resources, the owner being Player 1, and currently not executing any actions, then the one-hot encoding features will look like the following:

$$[0,1,0,0,0],[1,0,0,0,0],[1,0,0],$$
$$[0,0,0,0,1,0,0,0],[1,0,0,0,0,0]$$

  The 27 values of each feature plane for the position in the map of such worker will thus be the concatenation of the arrays above.

- **Action Space.** Given a map of size $h \times w$, the action is an 8-dimensional vector of discrete values as specified in Table 7.1. The action space is designed similarly to the action space formulation by Hausknecht, et al.,[123]. The first component of the action vector represents the unit in the map to issue actions to, the second is the action type, and the rest of the components represent the different parameters different action types can take. Depending on which action type is selected, the game engine will use the corresponding parameters to execute the action.

[2] https://github.com/santiontanon/microrts

Table 7.2: Results averaged over 4 random seeds. The symbol "-" means "not applicable". Higher is better for r_{episode} and lower is better for a_{null}, a_{busy}, a_{owner}, t_{solve}, and t_{first}.

Strategies	Map size	r_{invalid}	r_{episode}	a_{null}	a_{busy}	a_{owner}	t_{solve}	t_{first}
Invalid action penalty	4×4	-1.00	0.00	0.00	0.00	0.00	-	0.53%
		-0.10	30.00	0.02	0.00	0.00	50.94%	0.52%
		-0.01	**40.00**	0.02	0.00	0.00	14.32%	0.51%
		0.00	30.25	2.17	0.22	2.70	36.00%	0.60%
	10×10	-1.00	0.00	0.00	0.00	0.00	-	3.43%
		-0.10	0.00	0.00	0.00	0.00	-	2.18%
		-0.01	0.50	0.00	0.00	0.00	-	1.57%
		0.00	0.25	90.10	0.00	102.95	-	3.41%
	16×16	-1.00	0.25	0.00	0.00	0.00	-	0.44%
		-0.10	0.75	0.00	0.00	0.00	-	0.44%
		-0.01	1.00	0.02	0.00	0.00	-	0.44%
		0.00	1.00	184.68	0.00	2.53	-	0.40%
	24×24	-1.00	0.00	49.72	0.00	0.02	-	1.40%
		-0.10	0.25	0.00	0.00	0.00	-	1.40%
		-0.01	0.50	0.00	0.00	0.00	-	1.92%
		0.00	0.50	197.68	0.00	0.90	-	1.83%
Invalid action masking	04x04	-	**40.00**	-	-	-	8.67%	**0.07%**
	10x10	-	**40.00**	-	-	-	11.13%	**0.05%**
	16x16	-	**40.00**	-	-	-	11.47%	**0.08%**
	24x24	-	**40.00**	-	-	-	18.38%	**0.07%**
Masking removed	04x04	-	33.53	63.57	0.00	17.57	76.42%	-
	10x10	-	25.93	128.76	0.00	7.75	94.15%	-
	16x16	-	17.32	165.12	0.00	0.52	-	-
	24x24	-	17.37	150.06	0.00	0.94	-	-
Naive invalid action masking	4×4	-	**59.61**	-	-	-	11.74%	**0.07%**
	10×10	-	**40.00**	-	-	-	13.97%	**0.05%**
	16×16	-	**40.00**	-	-	-	30.59%	**0.10%**
	24×24	-	**38.50**	-	-	-	49.14%	**0.07%**

- **Rewards.** We are evaluating our agents on the simple task of harvesting resources as fast as they can for Player 1 who controls units at the top left of the map. A +1 reward is given when a worker harvests a resource, and another +1 is received once the worker returns the resource to a base.
- **Termination Condition.** We set the maximum game length to be 200 time steps, but the game could be terminated earlier if all the resources in the map are harvested first.

Notice that the space of invalid actions becomes significantly larger in larger maps. This is because the range of the first and last discrete values in the action space, corresponding to *Source Unit* and *Attack Target Unit* selection, grows linearly with the size of the map. To illustrate, in our experiments, there are usually only two units that can be selected as the *Source Unit* (the base and the worker). Although it is possible to produce more units or buildings to be selected, the production behavior has no contribution to reward and therefore is generally not learned by the agent. Note the range of *Source Unit* is $4 \times 4 = 16$ and $24 \times 24 = 576$, in maps of size 4×4 and 24×24, respectively. Selecting a valid *Source Unit* at random has a probability of $2/16 = 0.125$ in the 4×4 map and $2/576 = 0.0034$ in the 24×24 map. With such action space, we can examine the scalability of invalid action masking.

7.4.2 Training Algorithm

We use Proximal Policy Optimization[16] as the DRL algorithm to train our agents.

7.4.3 Strategies to Handle Invalid Actions

To examine the empirical importance of invalid action masking, we compare the following four strategies to handle invalid actions.

1. **Invalid action penalty.** Every time the agent issues an invalid action, the game environment adds a non-positive reward $r_{\text{invalid}} \leq 0$ to the reward produced by the current time step. This technique is standard in previous work[124]. We experiment with $r_{\text{invalid}} \in \{0, -0.01, -0.1, -1\}$, respectively, to study the effect of the different scales on the negative reward.

2. **Invalid action masking.** At each time step t, the agent receives a mask on the *Source Unit* and *Attack Target Unit* features such that only valid units can be selected and targeted. Note that in our experiments, invalid actions still could be sampled because the agent could still select incorrect parameters for the current action type. We didn't provide a feature-complete invalid action mask for simplicity, as the mask on *Source Unit* and *Attack Target Unit* already significantly reduce the action space.

3. **Naive invalid action masking.** At each time step t, the agent receives the same mask on the *Source Unit* and *Attack Target Unit* as described for invalid action masking. The action shall still be sampled according to the re-normalized probability calculated in Equation 7.4, which ensures no invalid actions could be sampled, but the gradient is updated according to the probability calculated in Equation 7.1. We call this implementation *naive invalid action masking* because its gradient does not replace the gradient of the logits corresponds to invalid actions with zero.

4. **Masking removed.** At each time step t, the agent receives the same mask on the *Source Unit* and *Attack Target Unit* as described for invalid action masking, and trains in the same way as the agent trained under invalid action masking. However, we then evaluate the agent without providing the mask. In other words, in this scenario, we evaluate what happens if we train with a mask, but then perform without it.

We evaluate the agent's performance in maps of sizes 4×4, 10×10, 16×16, and 24×24. All maps have one base and one worker for each player, and each worker is located near the resources.

7.4.4 Evaluation Metrics

We used the following metrics to measure the performance of the agents in our experiments: r_{episode} is the average episodic return over the last 10 episodes. a_{null} is the average number of actions that select a *Source Unit* that is not valid over the last 10 episodes. $a_{\text{busy}}$is the average number of actions that select a *Source Unit* that is already busy executing other actions over the last 10 episodes. a_{owner} is the average number of actions that select a *Source Unit* that does not belong to Player 1 over the last 10 episodes. t_{solve} is the percentage of total training time steps that it takes for the agents' moving average episodic return of the last 10 episodes to exceed 40. t_{first} is the percentage of the total training time step that it takes for the agent to receive the first positive reward.

7.4.5 Evaluation Results

We report the results in Figure 7.2 and in Table 7.2. Here we present a list of important observations:

Invalid action masking scales well. Invalid action masking is shown to scale well as the number of invalid actions increases; t_{solve} is roughly 12% and very similar across different map sizes. In addition, the t_{first} for invalid action masking is not only the lowest across all experiments (only taking about $0.05 - 0.08\%$ of the total time steps), but also consistent against different map sizes. This would mean the agent was able to find the first reward very quickly regardless of the map sizes.

Invalid action penalty does not scale. Invalid action penalty is able to achieve good results in 4×4 maps, but it does not scale to larger maps. As the space of invalid action gets larger, sometimes it struggles to even find the very first reward. E.g. in the 10×10 map, agents trained with invalid action penalty with $r_{\text{invalid}} = -0.01$ spent 3.43% of the entire training time just discovering the first

reward, while agents trained with invalid action masking take roughly 0.06% of the time in all maps. In addition, the hyper-parameter r_{invalid} can be difficult to tune. Although having a negative r_{invalid} did encourage the agents not to execute any invalid actions (e.g. a_{null}, a_{busy}, a_{owner} are usually very close to zero for these agents), setting $r_{\text{invalid}} = -1$ seems to have an adverse effect of discouraging exploration by the agent, therefore achieving consistently the worst performance across maps.

KL divergence explodes for naive invalid action masking. According to Table 7.2, the r_{episode} of naive invalid action masking is the best across almost all maps. In the 4×4 map, the agent trained with naive invalid action masking even learns to travel to the other side of the map to harvest additional resources. However, naive invalid action masking has two main issues: 1) As shown in the second row of Figure 7.2, the average Kullback–Leibler (KL) divergence[125] between the target and current policy of PPO for naive invalid action masking is significantly higher than that of any other experiments. Since the policy changes so drastically between policy updates, the performance of naive invalid action masking might suffer when dealing with more challenging tasks. 2) As shown in Table 7.2, the t_{solve} of naive invalid action masking is more volatile and sensitive to the map sizes. In the 24×24 map, for example, the agents trained with naive invalid action masking take 49.14% of the entire training time to converge. In comparison, agents trained with invalid action masking exhibit a consistent $t_{\text{solve}} \approx 12\%$ in all maps.

Masking removed still behaves to some extent. As shown in Figures 7.2b, masking removed is still able to perform well to a certain degree. As the map size gets larger, its performance degrades and starts to execute more invalid actions by, most prominently, selecting an invalid *Source Unit*. Nevertheless, its performance is significantly better than that of the agents trained with invalid action penalty even though they are evaluated without the use of invalid action masking. This shows that the agents trained with invalid action masking can, to some extent, still produce useful behavior when the invalid action masking can no longer be provided.

7.5 Conclusions

In this paper, we examined the technique of invalid action masking, which is a technique commonly implemented in policy gradient algorithms to avoid executing invalid actions. Our work shows that: 1) the gradient produced by invalid action masking is a valid policy gradient, 2) it works by applying a *state-dependent differentiable function* during the calculation of action probability distribution, 3) invalid action masking empirically scales well as the space of invalid action gets larger; in comparison, the common technique of giving a negative reward when an invalid action is issued fails to scale, sometimes struggling to find even the first reward in our environment, 4) the agent trained with invalid action masking was still able to produce useful behaviors with masking removed.

Given the clear effectiveness of invalid action masking demonstrated in this paper, we believe the community would benefit from wider adoption of this technique in practice. Invalid action masking empowers the agents to learn more efficiently, and we ultimately hope that it will accelerate research in applying DRL to games with large and complex discrete action spaces.

Chapter 8: Action Guidance

The last chapter explores the technique that helps reduce the search space of RTS games, which is generally useful. In this chapter, we investigate techniques that help us circumvent the downsides of using shaped rewards in Gym-μRTS. Namely, the agent may learn to overfit the shaped rewards instead of the true objective — this means the agent may get a lot of shaped rewards by harvesting resources or producing workers instead of trying to win the game as soon as possible. We propose a technique that we call *action guidance* that successfully trains agents to eventually optimize the true objective in games with sparse rewards while maintaining most of the sample efficiency that comes with reward shaping. We evaluate our approach in Gym-μRTS to demonstrate its effectiveness. The work of this chapter is based on the following workshop article:

- Shengyi Huang and Santiago Ontañón. Action guidance: Getting the best of sparse rewards and shaped rewards for real-time strategy games. *AIIDE Workshop on Artificial Intelligence for Strategy Games*, abs/2010.03956, 2020. URL https://arxiv.org/abs/2010.03956

8.1 Motivation

Training agents using Reinforcement Learning with sparse rewards is often difficult [126]. First, due to the sparsity of the reward, the agent often spends the majority of the training time doing inefficient exploration and sometimes not even reaching the first sparse reward during the entirety of its training. Second, even if the agents have successfully retrieved some sparse rewards, performing proper credit assignment is challenging among complex sequences of actions that have led to theses sparse rewards. Reward shaping [127] is a widely-used technique designed to mitigate this problem. It works by providing intermediate rewards that lead the agent towards the sparse rewards, which are the true objective. For example, the sparse reward for a game of Chess is naturally +1 for winning, -1 for losing, and 0 for drawing, while a possible shaped reward might be +1 for every enemy piece the agent takes. One of the critical drawbacks for reward shaping is that the agent sometimes learns to optimize for the shaped reward instead of the real objective. Using the Chess example, the agent might learn to take as many enemy pieces as possible while still losing the game. A good shaped reward achieves a nice balance between letting the agent find the sparse reward and being too shaped (so the agent learns to just maximize the shaped reward), but this balance can be difficult to find.

In this paper, we present a novel technique called *action guidance* that successfully trains the agent to eventually optimize over sparse rewards while maintaining most of the sample efficiency that comes with reward shaping. It works by constructing a *main policy* that only learns from the sparse reward function $R_{\mathcal{M}}$ and some *auxiliary policies* that learn from the shaped reward function $R_{\mathcal{A}_1}, R_{\mathcal{A}_2}, \ldots, R_{\mathcal{A}_n}$. During training, we use the same rollouts to train the main and auxiliary policies and initially set a high-probability of the main policy to take *action guidance* from the auxiliary policies, that is, *the main policy will execute actions sampled from the auxiliary policies*. Then the main policy and auxiliary policies are updated via off-policy policy gradient. As the training goes on, the main policy will get more independent and execute more actions sampled from its own policy. Auxiliary policies learn from shaped rewards and therefore make the training sample-efficient, while the main policy learns from the original sparse reward and therefore makes sure that the agents will eventually optimize over the true objective. We can see action guidance as combining reward shaping to train auxiliary policies interlieaved with a sort of imitation learning to guide the main policy from these auxiliary policies.

We examine action guidance in the context of a real-time strategy (RTS) game simulator called μRTS for three sparse rewards tasks of varying difficulty. For each task, we compare the performance of training agents with the sparse reward function $R_{\mathcal{M}}$, a shaped reward function $R_{\mathcal{A}_1}$, and action guidance with a singular auxiliary policy learning from $R_{\mathcal{A}_1}$. The main highlights are:

Action guidance is sample-efficient. Since the auxiliary policy learns from $R_{\mathcal{A}_1}$ and the main policy takes action guidance from the auxiliary policy during the initial stage of training, the main policy is more likely to discover the first sparse reward more quickly and learn more efficiently. Empirically, action guidance reaches almost the same level of sample efficiency as reward shaping in all of the three tasks tested.

The true objective is being optimized. During the course of training, the main policy has never seen the shaped rewards. This ensures that the main policy, which is the agent we are really interested in, is always optimizing against the true objective and is less biased by the shaped rewards. As an example, Figure 8.1 shows that the main policy trained with action guidance eventually learns to win the game as fast as possible, even though it has only learned from the match outcome reward (+1 for winning, -1 for losing, and 0 for drawing). In contrast, the agents trained with reward shaping learn more diverse sets of behaviors which result in high shaped reward.

To support further research in this field, we make our source code available at GitHub[1], as well as all the metrics, logs, and recorded videos[2].

8.2 Background

In this section, we briefly summarize the popular techniques proposed to address the challenge of sparse rewards.

Reward Shaping. Reward shaping is a common technique where the human designer uses domain knowledge to define additional intermediate rewards for the agents. Ng et al. [127] show that a slightly more restricted form of state-based reward shaping has better theoretical properties for preserving the optimal policy.

Transfer and Curriculum Learning. Sometimes learning the target tasks with sparse rewards is too challenging, and it is more preferable to learn some easier tasks first. *Transfer learning* leverages this idea and trains agents with some easier source tasks and then later transfer the knowledge through value function [128] or reward shaping [129]. *Curriculum learning* further extends transfer learning by automatically designing and choosing a full sequences of source tasks (i.e. a curriculum) [130].

Imitation Learning. Alternatively, it is possible to directly provide examples of human demonstration or expert replay for the agents to mimic via Behavior Cloning (BC) [131], which uses supervised learning to learn a policy given the state-action pairs from expert replays. Alternatively, Inverse Reinforcement Learning (IRL) [132] recovers a reward function from expert demonstrations to be used to train agents.

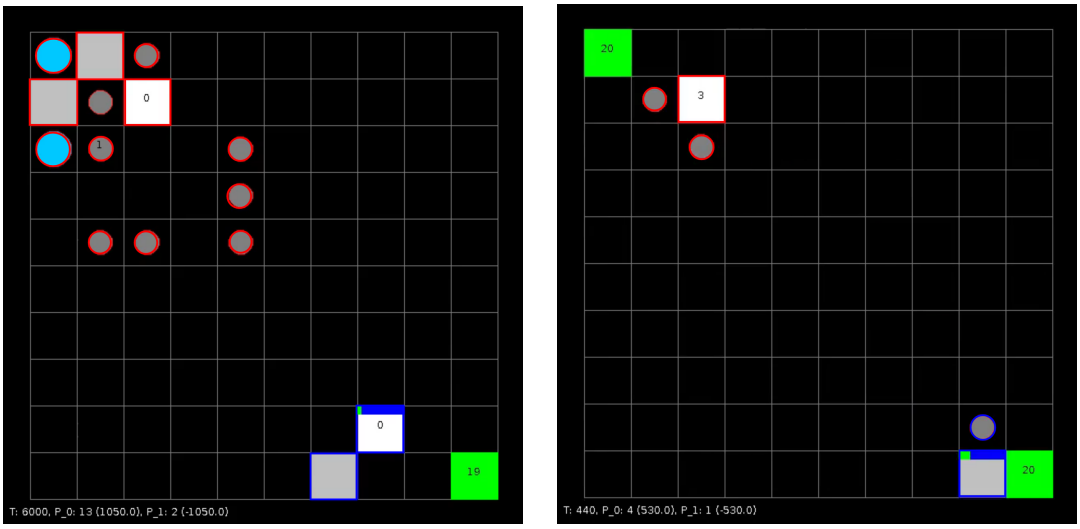
Curiosity-driven Learning. Curiosity driven learning seeks to design *intrinsic* reward functions [133] using metrics such as prediction errors [134] and "visit counts" [135;136]. These intrinsic rewards encourage the agents to explore unseen states.

Goal-oriented Learning. In certain tasks, it is possible to describe a goal state and use it in conjunction with the current state as input [137]. Hindsight experience replay (HER) [138] develops better utilization of existing data in experience replay by replaying each episode with different goals. HER is shown to be an effective technique in sparse rewards tasks.

Hierarchical Reinforcement Learning (HRL). If the target task is difficult to learn directly, it is also possible to hierarchically structure the task using experts' knowledge and train hierarchical agents, which generally involves a main policy that learns abstract goals, time, and actions, as well as auxiliary policies that learn primitive actions and specific goals [124]. HRL is especially popular in RTS games with combinatorial action spaces [139;140].

The most closely related work is perhaps Scheduled Auxiliary Control (SAC-X) [141], which is an HRL algorithm that trains auxiliary policies to perform primitive actions with shaped rewards and a main policy to schedule the use of auxiliary policies with sparse rewards. However, our approach differs in the treatment of the main policy. Instead of learning to *schedule* auxiliary policies, our

[1] https://github.com/anonymous-research-code/action-guidance
[2] Blinded for peer review



(a) shaped reward
(https://youtu.be/UM88KyBLQzM)

(b) action guidance
(https://youtu.be/arsDaIq4B38)

Figure 8.1: The screenshot shows the typical learned behavior of agents in the task of DefeatRandomEnemy. (a) shows that an agent trained with some shaped reward function $R_{\mathcal{A}_1}$ learns many helpful behaviors such as building workers (grey circles), combat units (blue circles), and barracks (grey square) or using owned units (with red boarder) to attack enemy units (with blue border), but does not learn to win as fast as possible (i.e. it still does not win at internal time step $t = 6000$). In contrast, (b) shows an agent trained with action guidance optimizes over the match outcome and learns to win as fast as possible (i.e. about to win the game at $t = 440$), with its main policy learning from the match outcome reward function $R_{\mathcal{M}}$ and a singular auxiliary policy learning from the same shaped reward function $R_{\mathcal{A}_1}$. Click on the link below figures to see the full videos of trained agents.

main policy learns to act in the entire action space by *taking action guidance* from the auxiliary policies. There are two intuitive benefits to our approach since our main policy learns in the full action space. First, during policy evaluation our main policy does not have to commit to a particular auxiliary policy to perform actions for a fixed number of time steps like it is usually done in SAC-X. Second, learning in the full action space means the main policy will less likely suffer from the definition of hand-crafted sub-tasks, which could be incomplete or biased.

8.3 Action Guidance

The key idea behind *action guidance* is to create a main policy that trains on the sparse rewards, and creating some auxiliary policies that are trained on shaped rewards. During the initial stages of training, the main policy has a high probability to take *action guidance* from the auxiliary policies, that is, the main policy can execute actions sampled from the auxiliary policies, rather than from its own policy. As the training goes on, this probability decreases, and the main policy executes more actions sampled from its own policy. During training, the main and auxiliary policies are updated via off-policy policy gradient. Our use of auxiliary policies makes the training sample-efficient, and our use of the main policy, who only sees its own sparse reward, makes sure that the agent will eventually optimize over the true objective of sparse rewards. In a way, *action guidance* can be seen as training agents using shaped rewards, while having the main policy learn by imitating from them.

Specifically, let us define $\mathcal{M}$ as the MDP that the main policy learns from and $\mathcal{A} = \{\mathcal{A}_1, \mathcal{A}_2, ..., \mathcal{A}_k\}$ be a set of auxiliary MDPs that the auxiliary policies learn from. In our constructions, $\mathcal{M}$ and $\mathcal{A}$ share the same state, observation, and action space. However, the reward function for $\mathcal{M}$ is $R_{\mathcal{M}}$,

which is the sparse reward function, and reward functions for $\mathcal{A}$ are $R_{\mathcal{A}_1}, ..., R_{\mathcal{A}_k}$, which are the shaped reward functions. For each of these MDPs $\mathcal{E} \in \mathcal{S} = \{\mathcal{M}\} \cup \mathcal{A}$ above, let us initialize a policy $\pi_{\theta_\mathcal{E}}$ parameterized by parameters $\theta_\mathcal{E}$, respectively. Furthermore, let us use $\pi_\mathcal{S} = \{\pi_{\theta_\mathcal{E}} | \mathcal{E} \in \mathcal{S}\}$ to denote the set of these initialized policies.

At each timestep t, let us use some exploration strategy S that selects a policy $\pi_b \in \pi_\mathcal{S}$ to sample an action a_t given s_t. At the end of the episode, each policy $\pi_\theta \in \pi_\mathcal{S}$ can be updated via its off-policy policy gradient [142;143]:

$$\mathbb{E}_{\tau \sim \pi_{\theta_b}} \left[\left(\prod_{t=0}^{T-1} \frac{\pi_\theta\left(a_t|s_t\right)}{\pi_{\theta_b}\left(a_t|s_t\right)} \right) \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta\left(a_t|s_t\right) A(\tau, V, t) \right] \tag{8.1}$$

When $\pi_\theta = \pi_{\theta_b}$, the gradient in Equation 8.1 means on-policy policy gradient update for π_θ. Otherwise, the objective means off-policy policy gradient update for π_θ.

8.3.1 Practical Algorithm

The gradient in Equation 8.1 is unbiased, but its product of importance sampling ratio $\left(\prod_{t=0}^{T-1} \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_b}(a_t|s_t)}\right)$ is known to cause high variance [64]. In practice, we clip the gradient the same way as Proximal Policy Gradient (PPO) [16]:

$$L^{CLIP}(\theta) = \mathbb{E}_{\tau \sim \pi_{\theta_b}} \left[\sum_{t=0}^{T-1} \left[\nabla_\theta \min\left(\rho_t(\theta) A(\tau, V, t), \text{clip}\left(\rho_t(\theta), \varepsilon\right) A(\tau, V, t)\right)\right] \right] \tag{8.2}$$

$$\rho_t(\theta) = \frac{\pi_\theta\left(a_t|s_t\right)}{\pi_{\theta_b}\left(a_t|s_t\right)}, \quad \text{clip}\left(\rho_t(\theta), \varepsilon\right) = \begin{cases} 1-\varepsilon & \text{if } \rho_t(\theta) < 1-\varepsilon \\ 1+\varepsilon & \text{if } \rho_t(\theta) > 1+\varepsilon \\ \rho_t(\theta) & \text{otherwise} \end{cases}$$

During the optimization phase, the agent also learns the value function and maximizes the policy's entropy. We therefore optimize the following joint objective for each $\pi_\theta \in \pi_\mathcal{S}$:

$$L^{CLIP}(\theta) = L^{CLIP}(\theta) - c_1 L^{VF}(\theta) + c_2 S[\pi_{\theta_b}], \tag{8.3}$$

where c_1, c_2 are coefficients, S is an entropy bonus, and L^{VF} is the squared error loss for the value function associated with π_θ as done by [16]. Although action guidance can be configured to leverage multiple auxiliary policies that learn diversified reward functions, we only use one auxiliary policy for the simplicity of experiments. In addition, we use ϵ-greedy as the exploration strategy S for determining the behavior policy. That is, at each timestep t, the behavior policy is selected to be $\pi_{\theta_\mathcal{M}}$ with probability $1-\epsilon$ and $\pi_{\theta_\mathcal{D}}$ for $\mathcal{D} \in \mathcal{A}$ with probability ϵ (note that is ϵ is different from the clipping coefficient ε of PPO). Additionally, ϵ is set to be a constant 0.95 at start for some period of time steps (e.g. 800,000), which we refer to as the *shift* period (the time it takes to start "shifting" focus away from the auxiliary policies), then it is set to linearly decay to ϵ_{end} for some period of time steps (e.g. 1,000,000), which we refer to as the *adaptation* period (the time it takes for the main policy to fully "adapt" and become more independent).

8.3.2 Positive Learning Optimization

During our initial experiments, we found the main policy sometimes did not learn useful policies. Our hypothesis is that this was because the main policy is updated with too many trajectories with zero reward. Doing a large quantities of updates of these zero-reward trajectories actually causes the policy to converge prematurely, which is manifested by having low entropy in the action probability distribution.

To mitigate this issue of having too many zero-reward trajectories, we use a preliminary code-level optimization called Positive Learning Optimization (PLO). After collecting the rollouts, PLO works by skipping the gradient update for $\pi_{\theta_\mathcal{E}} \in \pi_\mathcal{S}$ and its value function if the rollouts contains no

reward according to $R_{\mathcal{E}}$. Intuitively, PLO makes sure that the main policy learns from meaningful experience that is associated with positive rewards. To confirm its effectiveness, we provide an ablation study of PLO in the experiment section.

8.4 Experimental Study

We use μRTS[3] as our testbed, which is a minimalistic RTS game maintaining the core features that make RTS games challenging from an AI point of view: simultaneous and durative actions, large branching factors and real-time decision making. To interface with μRTS, we use gym-microrts[4] [36] to conduct our experiments. The details of gym-microrts as a RL interface can be found at Appendix 9.4.

8.4.1 Tasks Description

We examine the three following sparse reward tasks with a range of difficulties. For each task, we compare the performance of training agents with the sparse reward function $R_{\mathcal{M}}$, a shaped reward function $R_{\mathcal{A}_1}$, and action guidance with a single auxiliary policy learning from $R_{\mathcal{A}_1}$. Here are the descriptions of these environments and their reward functions.

1. LearnToAttack: In this task, the agent's objective is to learn move to the other side of the map where the enemy units live and start attacking them. Its $R_{\mathcal{M}}$ gives a +1 reward for each valid attack action the agent issues. This is of sparse reward because the action space is so large: the agent could have build a barracks or produce a unit; it is unlikely that the agents will by chance issue lots of moving actions (out of 6 action types) with correct directions (out of 4 directions) and then start attacking. Its $R_{\mathcal{A}_1}$ gives the difference between previous and current Euclidean distance between the enemy base and its closet unit owned by the agent as the shaped reward in addition to $R_{\mathcal{M}}$.

2. ProduceCombatUnits: In this task, the agent's objective is to learn to build as many combat units as possible. Its $R_{\mathcal{M}}$ gives a +1 reward for each combat unit the agent produces. This is a more challenging task because the agent needs to learn 1) harvest resources, 2) produce barracks, 3) produce combat units once enough resources are gathered, 4) move produced combat units out of the way so as to not block the production of new combat units. Its $R_{\mathcal{A}_1}$ gives +1 for constructing every building (e.g. barracks), +1 for harvesting resources, +1 for returning resources, and +7 for each combat unit it produces.

3. DefeatRandomEnemy: In this task, the agent's objective is to defeat a biased random bot of which the attack, harvest and return actions have 5 times the probability of other actions. Its $R_{\mathcal{M}}$ gives the match outcome as the reward (-1 on a loss, 0 on a draw and +1 on a win). This is the most difficult task we examined because the agent is subject to the full complexity of the game, being required to make both macro-decisions (e.g. deciding the high-level strategies to win the game) and micro-decisions (e.g. deciding which enemy units to attack. In comparison, its $R_{\mathcal{A}_1}$ gives +5 for winning, +1 for harvesting one resource, +1 for returning resources, +1 for producing one worker, +0.2 for constructing every building, +1 for each valid attack action it issues, +7 for each combat unit it produces, and $+(0.2 * d)$ where d is difference between previous and current Euclidean distance between the enemy base and its closet unit owned by the agent.

8.4.2 Agent Setup

We use PPO[16] as the base DRL algorithm to incorporate action guidance. We compared the following strategies:

[3] https://github.com/santiontanon/microrts
[4] https://github.com/vwxyzjn/gym-microrts

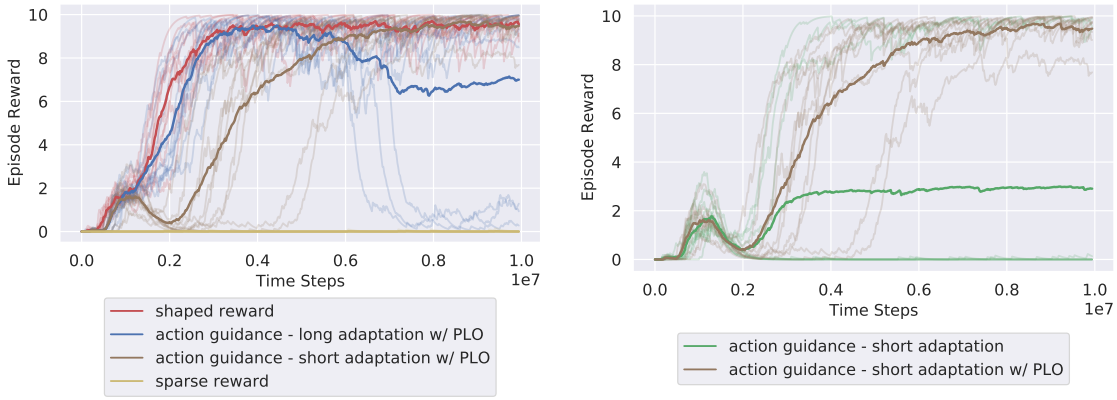


Figure 8.2: The faint lines are the actual sparse return of each seed for selected strategies in ProduceCombatUnits; solid lines are their means. The left figure showcase the sample-efficiency of action guidance; the right figure is a motivating example for PLO.

1. **Sparse reward (first baseline).** This agent is trained with PPO on $R_{\mathcal{M}}$ for each task.
2. **Shaped reward (second baseline).** This agent is trained with PPO on $R_{\mathcal{A}_1}$ for each task.
3. **Action guidance - long adaptation.** The agent is trained with PPO + action guidance with *shift* $= 2,000,000$ time steps, *adaptation* $= 7,000,000$ time steps, and $\epsilon_{end} = 0.0$
4. **Action guidance - short adaptation.** The agent is trained with PPO + action guidance with *shift* $= 800,000$ time steps, *adaptation* $= 1,000,000$ time steps, and $\epsilon_{end} = 0.0$
5. **Action guidance - mixed policy.** The agent is trained with PPO + action guidance with *shift* $= 2,000,000$ time steps and *adaptation* $= 2,000,000$ time steps, and $\epsilon_{end} = 0.5$. We call this agent "mixed policy" because it will eventually have 50% chance to sample actions from the main policy and 50% chance to sample actions from the auxiliary policy. It is effectively having mixed agent making decisions jointly.

Although it is desirable to add SAC-X to the list of strategies compared, it was not designed to handle domains with large discrete action spaces. Lastly, we also toggle the PLO option for action guidance - long adaptation, action guidance - short adaptation, action guidance - mixed policy, and sparse reward training strategies for a preliminary ablation study.

8.4.3 Experimental Results

Each of the 6 strategies is evaluated in 3 tasks with 10 random seeds. We report the results in Table 8.1. From here on, we use the term "sparse return" to denote the episodic return according to $R_{\mathcal{M}}$, and "shaped return" the episodic return according to $R_{\mathcal{A}_1}$. Below are our observations.

Action guidance is almost as sample-efficient as reward shaping. Since the auxiliary policy learns from $R_{\mathcal{A}_1}$ and the main policy takes a lot of action guidance from the auxiliary policy during the shift period, the main policy is more likely to discover the first sparse reward more quickly and learn more efficiently. As an example, Figure 8.2 demonstrates such sample-efficiency in ProduceCombatUnits, where the agents trained with sparse reward struggle to obtain the very first reward. In comparison, most action guidance related agents are able to learn almost as fast as the agents trained with shaped reward.

Action guidance eventually optimizes the sparse reward. This is perhaps the most important contribution of our paper. Action guidance eventually optimizes the main policy over the true objective, rather than optimizing shaped rewards. Using the ProduceCombatUnits task as an example, the agent trained with shaped reward would only start producing combat units once all

Table 8.1: The average sparse return achieved by each training strategy in each task over 10 random seeds.

	LearnToAttack	ProduceCombatUnit	DefeatRandomEnemy
sparse reward (first baseline)	3.30 ± 5.04	0.00 ± 0.01	-0.07 ± 0.03
sparse reward w/ PLO	0.00 ± 0.00	0.00 ± 0.01	-0.05 ± 0.03
shaped reward (second baseline)	10.00 ± 0.00	**9.57 ± 0.30**	0.08 ± 0.17
action guidance - long adaptation	**11.00 ± 0.00**	8.31 ± 2.62	0.11 ± 0.35
action guidance - long adaptation w/ PLO	**11.00 ± 0.01**	6.96 ± 4.04	**0.52 ± 0.35**
action guidance - mixed policy	**11.00 ± 0.00**	**9.67 ± 0.17**	**0.40 ± 0.37**
action guidance - mixed policy w/ PLO	**10.67 ± 0.12**	**9.36 ± 0.35**	**0.30 ± 0.42**
action guidance - short adaptation	**11.00 ± 0.01**	2.95 ± 4.48	-0.06 ± 0.04
action guidance - short adaptation w/ PLO	**11.00 ± 0.00**	**9.48 ± 0.51**	-0.05 ± 0.03

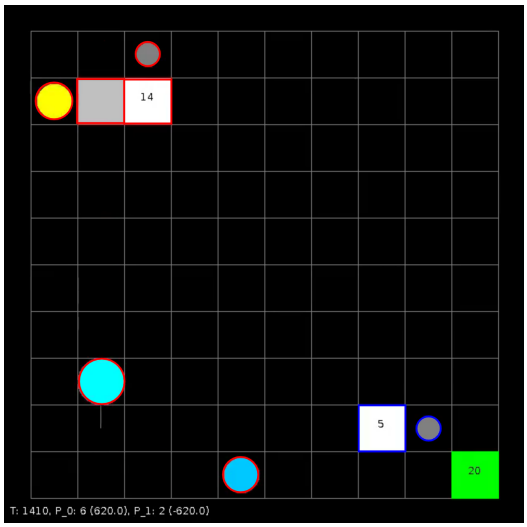
the resources have been harvested, probably because the +1 reward for harvesting and returning resources are easy to retrieve and therefore the agents exploit them first. Only after these resources are exhausted would the agents start searching for other sources of rewards then learn producing combat units.

In contrast, the main policy of action guidance - short adaptation w/ PLO are initially guided by the shaped reward agent during the shift period. During the adaptation period, we find the main policy starts to optimize against the real objective by producing the first combat unit as soon as possible. This disrupts the behavior learned from the auxiliary policy and thus cause a visible degrade in the main policy's performance during 1M and 2M timesteps as shown in Figure 8.2. As the adaption period comes to an end, the main policy becomes fully independent and learn to produce combat units and harvest resources concurrently. This behavior matches the common pattern observed in professional RTS game players and is obviously more desirable because should the enemy attack early, the agent will have enough combat units to defend.

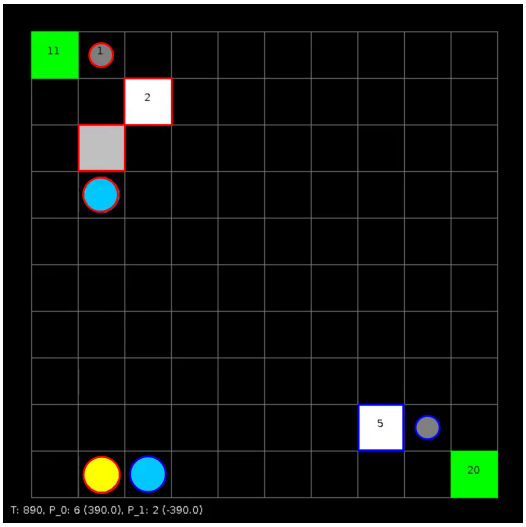
In the DefeatRandomEnemy task, the agents trained with shaped rewards learn a variety of behaviors; some of them learn to do a worker rush while others learn to focus heavily on harvesting resources and producing units. This is likely because the agents could get similar level of shaped rewards despite having diverse set of behaviors. In comparison, the main policy of action guidance - long adaptation w/ PLO would start optimizing the sparse reward after the shift period ends; it almost always learns to do a worker rush, which an efficient way to win against a random enemy as shown in Figure 8.1.

The hyper-parameters adaptation and shift matter. Although the agents trained with action guidance - short adaptation w/ PLO learns the more desirable behavior, they perform considerably worse in the harder task of DefeatRandomEnemy. It suggests the harder that task is perhaps the longer adaptation should be set. However, in ProduceCombatUnits, agents trained with action guidance - long adaptation w/ PLO exhibits the same category of behavior as agents trained with shaped reward, where the agent would only start producing combat units once all the resources have been harvested. A reasonable explanation is that higher adaptation gives more guidance to the main policy to consistently find the sparse reward, but it also inflicts more bias on how the task should be accomplished; lower adaption gives less guidance but increase the likelihood for the main policy to find better ways to optimize the sparse rewards.

Positive Learning Optimization results show large variance. We found PLO to be an interesting yet sometimes effective optimization in stabilizing the performance for agents trained with action guidance. However, the results show large variance: PLO either significantly helps the agents or make them much worse. As a motivating example, Figure 8.2 showcases the actual sparse return of 10 seeds in ProduceCombatUnits, where agents trained with action guidance - short adaptation and PLO seem to always converge while agents trained without PLO would only sometimes converge. However, PLO actually hurt the performance of action guidance - long adaptation in ProduceCombatUnits by having a few degenerate runs as shown in Figure 8.2. It is also worth noting the PLO does not help the sparse reward agent at all, suggesting PLO is a an optimization somewhat unique to action guidance.



(a) shaped reward
(https://youtu.be/MB0FjW-3Ktc)



(b) action guidance
(https://youtu.be/r5Nsda_YTNE)

Figure 8.3: The screenshot shows the typical learned behavior of agents in the task of ProduceCombatUnits. (a) shows an agent trained with shaped reward function $R_{\mathcal{A}_1}$ learn to only produce combat units once the resources are exhausted (i.e. it produces three combat units at $t = 1410$). In contrary, (b) shows an agent trained with action guidance learn to produce units and harvest resources concurrently (i.e. it produces three combat units at $t = 890$). Click on the link below figures to see the full videos of trained agents.

Action guidance - mixed policy is viable. According to Table 8.1, agents trained with action guidance - mixed policy with or without PLO seem to perform relatively well in all three tasks examined. This is a interesting discovery because it suggests action guidance could go both ways: the auxiliary policies could also benefit from the learned policies of the main policy. An alternative perspective is to consider the main policy and the auxiliary policies as a whole entity that mixes different reward functions, somehow making joint decision and collaborating to accomplish common goals.

8.5 Discussion

In this paper, we present a novel technique called *action guidance* that successfully trains the agent to eventually optimize over sparse rewards yet does not lose the sample efficiency that comes with reward shaping, effectively getting the best of both worlds. Our experiments with DefeatRandomEnemy in particular show it is possible to train a main policy on the full game of μRTS using only the match outcome reward, which suggests action guidance could serve as a promising alternative to the training paradigm of AlphaStar[27] that uses supervised learning with human replay data to bootstrap an agent. As part of our future work, we would like to scale up the approach to defeat stronger opponents.

Chapter 9: Unit-level Control

The work presented in the previous chapters (6, 7, and 8) all work with a restricted action space of μRTS, which only issues a single action to a single unit at each time step. In comparison, the previous μRTS bots that leverage other ML methods (e.g., Monte-Carlo Tree Search) can issue actions to all the player-owned units simultaneously. This limitation makes it difficult to evaluate our RL agent against previous μRTS bots. In this chapter, we study techniques that will allow us to scale the RL agent to the full game of μRTS. In particular, we provide action masks on the action parameters, significantly improving the RL agent's training efficiency.

This chapter has two main contributions: 1) we introduce a new version of Gym-μRTS for **full-game** RTS research, and 2) we present a collection of techniques to scale DRL to play full-game μRTS as well as ablation studies to demonstrate their empirical importance. Our best-trained bot can defeat every μRTS bot we tested from the past μRTS competitions when working in a single-map setting, resulting in a state-of-the-art DRL agent while only taking about 60 hours of training using a single machine (one GPU, three vCPU, 16GB RAM). Being able to train RL agents under full-game settings makes this line of research more accessible to researchers — previous work under the full-game settings leverages StarCraft II and has high computational costs[27], usually requiring the use of thousands of GPUs and CPUs for weeks. The work of this chapter is based on the following publication:

- Shengyi Huang, Santiago Ontañón, Chris Bamford, and Lukasz Grela. Gym-μrts: Toward affordable full game real-time strategy games research with deep reinforcement learning. In *2021 IEEE Conference on Games (CoG)*, pages 1–8. IEEE, 2021

9.1 Authorship

Shengyi Huang led the testbed design, experiment design, and paper writing. Chris Bamford led the experiments with the encoder-decoder network. Lukasz Grela helped with project planning and prototyping.

9.2 Motivation

In recent years, researchers have achieved great success in applying Deep Reinforcement Learning (DRL) algorithms to Real-time Strategy (RTS) games. Most notably, DeepMind trained a grandmaster-level AI called AlphaStar with DRL for the popular RTS game StarCraft II[27]. AlphaStar demonstrates impressive strategy and game control, presenting many human-like behaviors, and is able to defeat professional players consistently. Given most previously designed bots fail to perform well in the full-game against humans[30], AlphaStar clearly represents a significant milestone in the field. While this accomplishment is impressive, it comes with high computational costs. In particular, AlphaStar and even further attempts by other teams to lower the computational costs[144] still require thousands of CPUs and GPUs/TPUs to train the agents for an extended period of time, which is outside of the computational budget of most researchers.

This paper has two main contributions to address this issue. The first main contribution is to introduce Gym-μRTS as an RL testbed for affordable full-game RTS research, which focuses on all aspects of the game such as harvesting resources, defending units, and attack enemies (this is in contrast to *mini-games* that only focus on one aspect of the game). Gym-μRTS is a reinforcement learning interface for the RTS game μRTS[145], which has been a popular platform to test out a variety of AI techniques for RTS games. Despite its simple visuals, μRTS captures the core challenges of RTS games. Although Gym-μRTS shares many similarities to the StarCraft II Learning Environment (PySC2)[107], there are also many key differences (e.g., PySC2 uses a human-like action space whereas

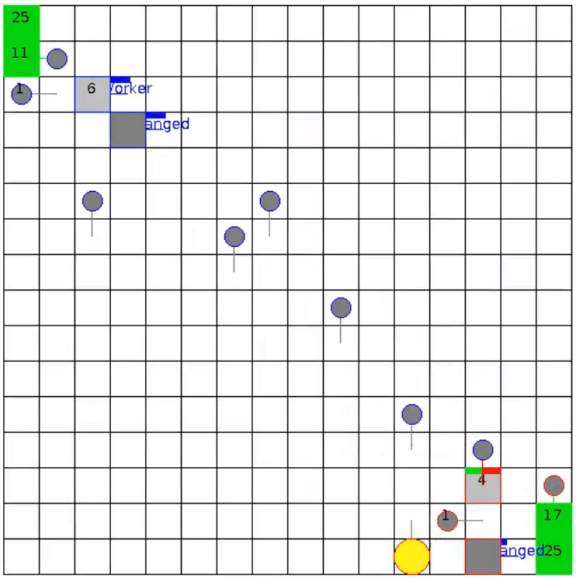


Figure 9.1: Screenshot of our best-trained agent (top-left) playing against CoacAI (bottom-right), the 2020 μRTS AI competition champion. Strategy-wise, our agent usually defeats CoacAI by harvesting resources (green squares) efficiently using two workers (dark gray circles), doing a highly optimized worker rush that takes out the enemy base in the bottom right (shown with 50% damage), followed by a transition to the mid and late game by producing combat units (colored circles) from the barracks (dark gray squares). The blue and red border suggest the unit is owned by player 1 and 2, respectively. See additional combat videos here: https://bit.ly/3llOhex

Gym-μRTS uses a lower-level action space). Through Gym-μRTS, we are able to conduct full-game RTS research using DRL without extensive technical resources such as high-performance compute clusters.

Despite the simplifications done in μRTS, playing 1v1 competitive matches via DRL is still a daunting task. Thus, our second main contribution is a collection of techniques to scale DRL to play μRTS. We start with a Proximal Policy Optimization (PPO)[16] implementation that matches implementation details of PPO in *openai/baselines*[33], and incrementally stack augmentations to account for Gym-μRTS's combinatorial action space (all units must be controlled simultaneously) and improve training efficiency and performance. Among these augmentations, two are *essential*: 1) action composition and 2) invalid action masking. These two augmentations combined allowed us to bootstrap an initial agent that could compete on the 16×16 map to a reasonable standard. Additionally, we experimented with 3) diversified training opponents, and 4) different neural network architectures. We provide ablation studies to shed insights on the importance of each of these augmentations. Our best-trained agent can defeat every μRTS bot we tested against, from the past μRTS competitions[1] in a single-map setting, establishing a new state-of-the-art for DRL bots in μRTS while only taking about 60 hours of training using a single machine (one GPU, three vCPU, 16GB RAM). We make source code and trained models [2], as well as all the metrics, logs, and recorded videos[3] available for comparison.

[1] https://sites.google.com/site/micrortsaicompetition/home
[2] https://github.com/vwxyzjn/gym-microrts-paper
[3] https://wandb.ai/vwxyzjn/gym-microrts-paper

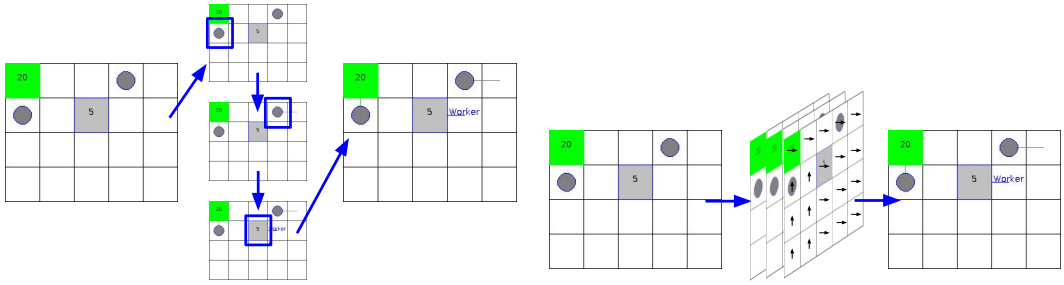
9.3 Background

Real-time Strategy (RTS) games are complex adversarial domains, typically simulating battles between a large number of combat units, that pose a significant challenge to both human and artificial intelligence[29]. Designing AI techniques for RTS games is challenging due to a variety of reasons: 1) players need to issue actions in real-time, leaving little time computational budget, 2) the action spaces grows combinatorially with the number of units in the game, 3) the rewards are very sparse (win/loss at the end of the game), 4) generalizing against diverse set of opponents and maps is difficult, and 5) stochasticity of game mechanics and partial observability (these last two are not considered in this paper).

StarCraft I & II are very popular RTS games and, among other games, have attracted much research attention. Past work in this area includes reinforcement learning[146], case-based reasoning[147;148], or game tree search[149–152] among many other techniques designed to tackle different sub-problems in the game, such as micromanagement, or build-order generation. In the full-game settings, however, most techniques have had limited success in creating viable agents to play competitively against professional StarCraft players until recently. In particular, DeepMind introduced AlphaStar[27], an agent trained with DRL and self-play, that sets a new state-of-the-art bot for StarCraft II, defeating professional players in the full-game. In Dota 2, a popular collaborative online-player game that shares many similar challenges as StarCraft, Open AI Five[66] is able to create agents that can achieve super-human performance. Although these two systems achieve great performance, they come with large computational costs. AlphaStar used 3072 TPU cores and 50,400 preemptible CPU cores for a duration of 44 days[27;144]. This makes it difficult for those with less computational resources to do full-game RTS research using DRL.

There are usually three ways to circumvent this computational costs. The first way is to focus on sub problems such as combat scenarios[111]. The second way is to reduce the full-game complexity by either considering hierarchical actions spaces or incorporating scripted actions[67;110]. The third way is to use alternative game simulators that run faster such as Mini-RTS[108], Deep RTS[153], and CodeCraft[4].

We show that Gym-μRTS as an alternative that could be used for full-game RTS research with the full action space while using affordable computational resources.



(a) Unit Action Simulation (UAS) calls the policy iteratively until all units have actions (at each step, it).

(b) Gridnet predicts an action for each cell in the map (it predicts all the action component planes the policy chooses a unit and issue a unit action to in one step), which then define which actions each unit will perform.

Figure 9.2: Demonstration of how actions are assigned under UAS and Gridnet.

[4] https://github.com/cswinter/DeepCodeCraft

9.4 Gym-μRTS: Unit-level Control

Gym-μRTS[5] is a reinforcement learning interface for the RTS games simulator μRTS[6]. Despite having a simplified implementation, μRTS captures the core challenges of RTS games, such as combinatorial action space, real-time decision-making, optionally partial observability and stochasticity. Gym-μRTS's observation space provides a series of feature maps similar to PySC2 (the StarCraft II Learning environment[107]). Its action space design, however, is more low-level due to its lack of AI-assisted actions. In this section, we introduce their technical details.

9.4.1 Observation Space.

Given a map of size $h \times w$, the observation is a tensor of shape (h, w, n_f), where n_f is a number of feature planes that have binary values. The observation space used in this paper uses 27 feature planes as shown in Table 7.1. The different feature planes result as the concatenation of multiple one-hot encoded features. As an example, if there is a worker player 1 with hit points equal to 1, not carrying any resources, and currently not executing any actions, then the one-hot encoding features will look like this (see Table 7.1):

$$[0,1,0,0,0],[1,0,0,0,0],[1,0,0],$$
$$[0,0,0,0,1,0,0,0],[1,0,0,0,0,0,0]$$

Each feature plane contains one value for each coordinate in the map. The values for the 27 feature planes for the position in the map of such worker will thus be:

$$[0,1,0,0,0,1,0,0,0,0,1,0,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0]$$

9.4.2 Action Space.

Compared to traditional reinforcement learning environments, the design of the action space of RTS games is more difficult because, depending on the game state, there is a different number of units to control, and each unit might have different number of actions available. This poses a challenge for directly applying off-the-shelf DRL algorithm such as PPO that generally assume a fixed output size for the actions. Early work on RL in RTS games simply learned policies for individual units, rather than having the policy control all the units at once[146]. To address this issue, we decompose the action space into two parts: the *unit action space* (the space of possibilities for issuing actions to only one unit) and the *player action space* (the space of unit actions for all the units a player owns).

In the unit action space, given a map of size $h \times w$, the unit action is an 8-dimensional vector of discrete values as specified in Table 9.1. The first component of the unit action vector represents the unit in the map to issue commands to, the second is the unit action type, and the rest of components represent the different parameters different unit action types can take. Depending on which unit action type is selected, the game engine will use the corresponding parameters to execute the action. As an example, if the RL agent issues a "move south" unit action to the worker at $x = 3, y = 2$ in a 16×16 map, the unit action will be encoded in the following way:

$$[3 + 2 * 16, 1, 2, 0, 0, 0, 0, 0]$$

In the player action space, we compare two ways to issue player actions to a variable number of units at each frame: **Unit Action Simulation (UAS)** and **Gridnet**[154].

Their mechanisms are best illustrated through an example as shown in Figs. 9.2a and 9.2b, where the player owns two workers and a base in a 4×5 map.

UAS calls the RL policy iteratively. At each step, the policy chooses a unit based on the *source unit masks* (a vector of $h \times w$ scalars). It then chooses the action type and parameter via the *unit*

[5] https://github.com/vwxyzjn/gym-microrts
[6] https://github.com/santiontanon/microrts

Table 9.1: Observation features and action components. $a_r = 7$ is the maximum attack range.

Observation Features	Planes	Description
Hit Points	5	0, 1, 2, 3, ≥ 4
Resources	5	0, 1, 2, 3, ≥ 4
Owner	3	player 1, -, player 2
Unit Types	8	-, resource, base, barrack, worker, light, heavy, ranged
Current Action	6	-, move, harvest, return, produce, attack
Action Components	**Range**	**Description**
Source Unit	$[0, h \times w - 1]$	the location of the unit selected to perform an action
Action Type	$[0, 5]$	NOOP, move, harvest, return, produce, attack
Move Parameter	$[0, 3]$	north, east, south, west
Harvest Parameter	$[0, 3]$	north, east, south, west
Return Parameter	$[0, 3]$	north, east, south, west
Produce Direction Parameter	$[0, 3]$	north, east, south, west
Produce Type Parameter	$[0, 6]$	resource, base, barrack, worker, light, heavy, ranged
Relative Attack Position	$[0, a_r^2 - 1]$	the relative location of the unit that will be attacked

action masks (a vector of $6 + 4 + 4 + 4 + 4 + 7 + 49$ scalars). We then compute a "simulated game state" where that action has been issued (and any potential rewards collected). Once all three units have been issued actions, the simulated game states are discarded, and the three actions are collected and sent to the actual game environment.

Under Gridnet[154], The RL agent receives a *player action mask* (a tensor of shape $(h, w, 1 + 6 + 4 + 4 + 4 + 4 + 7 + 49)$, where the first plane indicates if the source unit is available). It then issues actions to each cell in this map in one single step, that is, issues in total $4 * 5 = 20$ unit actions. The environment executes the three valid actions (actions in cells with no player-owned units are ignored).

9.4.3 The Action Spaces of Gym-μRTS and PySC2

Although Gym-μRTS is heavily inspired by and shares many similarities with PySC2[107], their action space designs are considerably different. Specifically, PySC2 has designed its action space to mimic the human interface, while Gym-μRTS has a more low-level action that require actions being issued for each individual unit. This distinction is rather interesting from a research standpoint because certain challenges are easier for an AI agent and some more difficult.

Consider the canonical task of harvesting resources and returning them to the base. In PySC2, the RL agent would need to issue two actions at two timesteps 1) select an area that has workers and 2) move the selected workers towards to a coordinate that has resources. Then, the workers will continue harvesting resources until otherwise instructed. Note that this sequence of actions is assisted by AI algorithms such as path-finding. After the workers harvest the resources, the engine automatically determines the closest base for returning the resources, and repeating these actions to continuously harvest resources. So the challenge for the RL agent is to learn to select the correct area and move to the correct coordinates. In Gym-μRTS, however, the RL agent can only issue primitive actions to the workers such as "move north for one cell" or "harvest resource that is one cell away at north". Therefore, it needs to constantly issue actions to control units at all times, having to learn how to perform these AI-assisted decisions from scratch[7].

The benefit of PySC2's approach is that it makes it easier to do imitation learning from human datasets and the resulting agent will have a fairer comparison when evaluated against humans since the AI and the human are mostly playing the same game. That being said, the human interface

[7]Notice, however that μRTS offers both the low-level interface and a PySC2-style interface with AI-assisted actions, but for Gym-μRTS, we only expose the former.

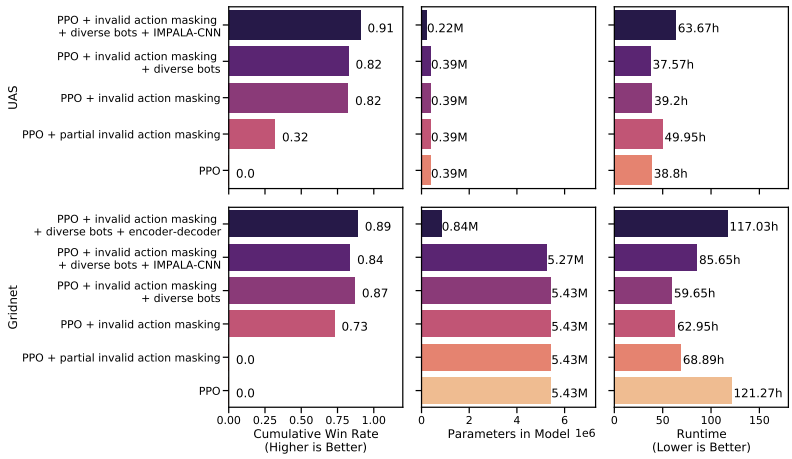


Figure 9.3: Ablation study for UAS and Gridnet.

could be an artificial limitation to the AI system. In particular, the human interface is constructed to accommodate the human limitations: humans' eyes have limited range, so camera locations are designed to help capture larger maps, and humans have limited physical mobility, so hotkeys are set to help control a group of units with one mouse click. However, machines don't have these limitations and can observe the entire map and issue actions to all units individually.

9.4.4 Reward Function

We use a *shaped reward* function to train the agents, which gives the agent +10 for winning, 0 for drawing, -10 for losing, +1 for harvesting one resource, +1 for producing one worker, +0.2 for constructing a building, +1 for each valid attack action it issues, +4 for each combat unit it produces. It gives the rewards to the frame at which the events are initialized (e.g. attack takes 5 game frames to finish, but the attack reward is given at the first frame). For reporting purposes, we also keep track of the *sparse reward*, which is +1 for winning, 0 for drawing, -1 for losing. The shaped reward weights are picked by hand with very little tuning.

Note this shaped reward function is similar to the one used in Open AI Five for Dota 2[66]. Like in Open AI Five, it is possible for the agents to gain more shaped rewards by doing other good behaviors than winning the game outright. Notice we have avoided using very large win/lose rewards because anecdotally large reward numbers could cause worse performance for RL algorithms, which might be the reason why reward normalization[20] or reward clipping[11] have been used in previous work.

9.5 Experimental Study

We use PPO[16], a popular policy gradient algorithm, to train agents for all experiments in this paper. In addition to PPO's core algorithm, many implementation details and empirical setting also have a huge impact on the algorithm's performance[20].

We start with a PPO implementation that matches the implementation details and benchmarked performance in *openai/baselines*[338], and use it along with the architecture from Mnih, et al. (denoted as Nature-CNN)[11] as the baseline. We train the RL agents using UAS and Gridnet by playing against CoacAI, the 2020 μRTS competition winner, in the standard `6x6basesWorkers` map, where the RL agents always spawn from the top left position and end episodes after 2000 game ticks. We then incrementally include augmentations for both UAS and Gridnet and compare their relative performance.

[8] See https://costa.sh/blog-the-32-implementation-details-of-ppo.html

We run each ablation with 4 random seeds each. Then, we select the best performing seeds according to the reported sparse reward function and evaluate them against a pool of 11 bots with various strategies that have participated in previous μRTS competitions (other competition bots are not included due to either staleness or difficulty to set up) and 2 baseline bots which are mainly used for testing. All μRTS bots are configured to use their μRTS competition parameters and setups. The name, category and best result of these bots are listed in Table 9.2. The evaluation involves playing 100 games against each bot in the pool for 4000 maximum game ticks, and we report the cumulative win rate, the model size, and total run time in Figure 9.3. To further provide insights, we record videos of the RL agents against each of the bots in the pool and make them publicly available[9]. Let us now describe the different augmentations we added on top of PPO.

9.5.1 Action Composition

After having solved the problem of issuing actions to a variable number of units (via either UAS or Gridnet), the next problem is that even the action space of a single unit is too large. Specifically, to issue a single action a_t in μRTS using UAS, according to Table 9.1, we have to select a Source Unit, Action Type, and its corresponding action parameters. So in total, there are $hw \times 6 \times 4 \times 4 \times 4 \times 4 \times 6 \times a_r^2 = 9216(hwa_r^2)$ number of possible discrete actions, which includes many invalid actions, which is huge even for small maps (about 50 million in the map size we use in this paper).

To address this problem, we use *action composition*, where we consider an action as composed of some smaller *independent* discrete actions. Namely, a_t is composed of a set of smaller actions $D = \{a_t^{\text{Source Unit}}, a_t^{\text{Action Type}}, a_t^{\text{Move Parameter}}, a_t^{\text{Harvest Parameter}}, a_t^{\text{Return Parameter}}, a_t^{\text{Produce Direction Parameter}}, a_t^{\text{Produce Type Parameter}}, a_t^{\text{Relative Attack Position}}\}$. And the policy gradient is updated in the following way (without considering the PPO's clipping for simplicity):

$$\begin{aligned}\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) G_t &= \sum_{t=0}^{T-1} \nabla_\theta \left(\sum_{a_t^d \in D} \log \pi_\theta(a_t^d|s_t) \right) G_t \\ &= \sum_{t=0}^{T-1} \nabla_\theta \log \left(\prod_{a_t^d \in D} \pi_\theta(a_t^d|s_t) \right) G_t\end{aligned}$$

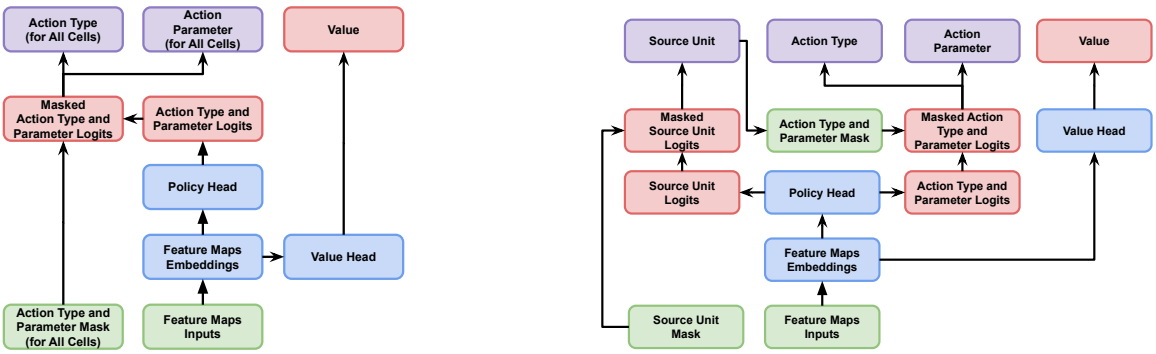
Implementation-wise, for each action component, the logits of the corresponding shape are output by the policy, which we refer to as action component logits. Each action a_t^d is sampled from a softmax distribution parameterized by these action component logits. In this way, the algorithm has to generate $hw + 6 + 4 + 4 + 4 + 4 + 6 + a_r^2 = hw + 36 + a_r^2$ logits, significantly less than $9216(hwa_r^2)$ (301 vs 50 million).

9.5.2 Invalid Action Masking

The next most important augmentation in our experiments is *invalid action masking*, which "masks out" invalid actions out of the action space (by exploiting the fact that we know the rules of the game), significantly reducing it. This is used in PySC2[107], OpenAI Five[66], and a number of related work with large action spaces[111].

Masks are generated and being applied as shown in Figure 9.4. Under UAS, the agent would first sample a source unit based on the source units masks of shape (hw), then query the game client for the action type and parameter mask of the said units with shape (78). Under Gridnet, the agent would receive all the masks up front on source unit, action type and parameter with shape $(hw, 79)$, where the first plane of 79 is the mask on the source unit selection. Note that in both cases, the agent received a *full* action mask that in a sense significantly reduce the search space. In contrast, PySC2 and SMAC (the StarCraft Multi-Agent Challenge)[111] would only provide a *partial* mask on the action type, and the logits of action parameters are unmasked (our action types and action parameters are function identifiers and arguments in PySC2's term). This could explain why invalid

[9] https://wandb.ai/vwxyzjn/gym-microrts-paper-eval/reports/Final-Eval--Vmlldzo0OTY1Mzc



(a) Gridnet. **(b)** UAS.

Figure 9.4: Neural network architectures for Gridnet and UAS. The green boxes are (conditional) inputs from the environments, blue boxes are neural networks, red boxes are outputs, and purple boxes are sampled outputs.

action masking does not seem to cause as drastic of a difference in PySC2 as shown by Kanervisto et al.[120].

In the interest of ablation study, we also conduct experiments that provide masking on the action types but not the action parameters, which is more similar to PySC2's settings. As shown in Figure 9.3, we see that having only a partial mask has little impact whereas having the full mask considerably improves performance. Although the action space and PySC2 is quite different as discussed above, masking all invalid actions maximally reduces the action space, hence simplifying the learning task. We therefore believe that the PySC2 agents could receive a performance boost by providing masks on function arguments as well.

9.5.3 Other augmentations

This section details other additional augmentations that contribute to the agents' performance, but not as much as the previous two (which are essential for having an agent that even starts learning to play the full game).

Diverse Opponents

The baseline setting is to train the agents against CoacAI. However, this lacks a diversified experience and when evaluating, we frequently see the agents being defeated by AIs as simple as WorkerRush. To help alleviate this problem, we train the agents against a diverse set of built-in bots. Since we train with 24 parallel environments for PPO, we set 18 of these environments to have CoacAI as the opponent, 2 to have RandomBiasedAI, 2 to have WorkerRush, and 2 to have LightRush. Per Figure 9.3, we see a rather significant performance boost for Gridnet, whereas in UAS the performance boost is milder.

Nature-CNN vs Impala-CNN vs Encoder-Decoder

To seek better neural network architectures, we experimented with the use of residual blocks[155] (denoted IMPALA-CNN), which have been shown to improve the agents' performance in several domains like DMLab[34]. Additionally, Han et al. [154] also experimented with an encoder-decoder network in Gridnet, so we also conducted experiments using this architecture. Per the ablation study in Figure 9.3, we see IMPALA-CNN helps with the performance of UAS whereas encoder-decoder benefits Gridnet.

Table 9.2: The previous μRTS competition bots.

Name	Category	Best result
CoacAI	Scripted	1st place in 2020
Tiamat	MCTS-based	1st place in 2018
MixedBot	MCTS-based	2nd place in 2019
Droplet	MCTS-based	3rd place in 2019
Izanagi	MCTS-based	4th place in 2019
Rojo	MCTS-based	5th place in 2020
LightRush	Scripted	6th place in 2020
GuidedRojoA3N	MCTS-based	7th place in 2020
WorkerRush	Scripted	8th place in 2020
NaiveMCTS	MCTS-based	9th place in 2020
RandomBiasedAI	Scripted	10th place in 2020
Random	Scripted	-
PassiveAI	Scripted	-

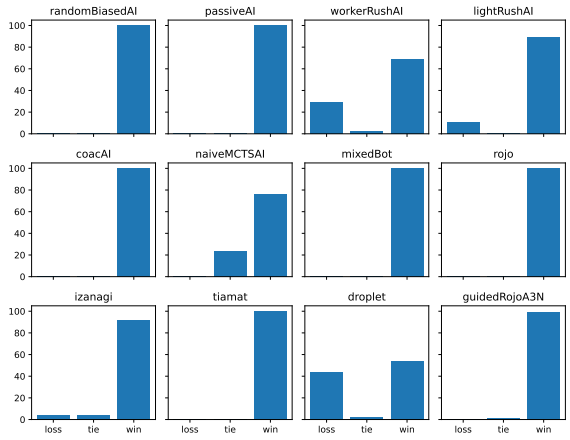


Figure 9.5: Match results: the y-axis shows the number of losses, ties, and wins against AIs listed in Table 9.2. The Random bot's match result is excluded for presentation purposes.

9.6 Discussion

Establishing a SOTA in Gym-μRTS. According to Figure 9.3, our best agent consists of *ppo + coacai + invalid action masking + diverse opponents + impala cnn*, reaching the cumulative win rate of 91%. Additionally, Figure 9.5 shows the specific match results, showing this agent can outperform all other bots in the pool. Note that in the μRTS competition settings the players could start in two different locations of the map whereas our agent always start from the top left. Nevertheless, due to the symmetric nature of the map, we could address this issue by "rotating" the map when needed so that both starting locations look the same to our agent. Therefore, our agent establishes the state of the art for μRTS in the `6x6basesWorkers` map. Note that generalizing to handle a variety of maps (including the asymmetric ones) in μRTS competition settings is part of our future work (also note that some work on StarCraft II also focused in the one map setting[144], while still requiring large computation budgets).

Our best agent struggles the most against the Droplet bot, which typically uses a worker rush strategy, but enhanced thanks to MCTS search. Droplet usually defeats our agent by destroying the first barracks our agent makes, which is a rare experience with other bots. As a result, our agent would keep trying to build a barracks until it exhausts its resources, at which point, Droplet would

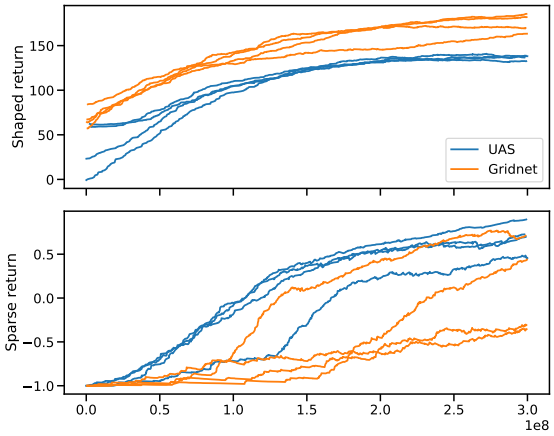


Figure 9.6: The shaped and sparse return over training steps for all 4 random seeds of *PPO + invalid action masking* for Gridnet and UAS. The curve is smoothed using exponential moving average with weight 0.99.

have more left over resources, build more workers and eventually defeat our agent. However, if by chance our agent successfully builds and protects the first barracks and combat units, it is usually able to defeat Droplet. As part of our future work, we would like to include agents like Droplet in our training process. However, search-based bots like Droplet significantly decrease the speed of training.

Hardware Usage and Training Time. Most of our experiments are conducted using 3 vCPUs, 1 GPU, and 16GB RAM. According to Figure 9.3, the experiments take anywhere from 37 hours to 117 hours, where our SOTA agent takes 63 hours.

Model size vs performance. Overall, Gridnet models have more parameters compared to the UAS models. This is because Gridnet predicts the action type and parameter logits for every cell in the map. We did not find a strong correlation between the model's size (in number of trainable parameters) and the performance of the agents. As shown in Figure 9.3, it is clear that the techniques such as invalid action masking or different neural network architectures are more important to the performance than the sheer number of the model's trainable parameters in our experiments.

Variance w.r.t. Shaped and Sparse Reward. In almost all experiments conducted in this paper, we observe the RL agents are able to optimize against the shaped rewards well, showing little variance across different random seeds; however, this is not the case with respect to the sparse reward (win/loss). We report the sum of shaped rewards and sparse rewards in the episode as *shaped return* and *sparse return* respectively in Figure 9.6, where we usually see little difference in the shaped return when the sparse (win/loss) return could be drastically different. This is a common drawback with reward shaping: agents sometimes overfit to the shaped rewards instead of sparse rewards.

UAS vs Gridnet. Figure 9.6 shows a typical result where Gridnet is able to get much higher shaped return, but it receives relatively similar sparse return as UAS. Upon further inspection of the agents actual behaviors, we found the Gridnet agents obtain higher shaped return by 1) producing more barracks, 2) producing more combat units, and 3) harvesting more resources effectively. In fact, Gridnet agents learn to harvest resources using three workers, which is a behavior we haven't observed in any existing bots. We suspect this difference is due to how rewards are attributed in UAS vs Gridnet. UAS attributes rewards to unit actions *individually*, while Gridnet attributes the rewards to the player action *collectively*.

Depending on the implementation, Gridnet agents usually have many more trainable parameters. Also, when the player owns a relatively small amount of units, it is faster to step the environment using UAS because Gridnet has to predict an action for all the cells in the map; however, when the player owns a large number of units, Gridnet's mechanism becomes faster because UAS has to do more simulated steps and thus more inferences.

The Amount of Human Knowledge Injected. In our best-trained agents, there are usually three sources of human knowledge injected: 1) the reward function, 2) invalid action masking, and 3) the use of human-designed bots such as CoacAI. In comparison, AlphaStar uses 1) human replays, 2) its related use of Statistics z and Supervised KL divergence[27], and 3) invalid action masking.

9.6.1 Conclusions and Future Work

We present a new efficient library, Gym-μRTS, which allows DRL research to be realized in the complex RTS environment μRTS. Through Gym-μRTS, we conducted ablation studies on techniques such as action composition, invalid action masking, diversified training opponents, and novel neural network architectures, providing insights on their importance to scale agents to play the full game of μRTS. Our agents can be trained on a single CPU+GPU within 2-4 days, which is a reasonable hardware-and-time budget that is available to many researchers outside of large research labs

For future work, we would like to consider multiple maps and the partial observability setting of μRTS (i.e. fog-of-war). Additionally, we also want to experiment with selfplay, which further reduces human knowledge injected such as the human-designed bots we used in this paper.

Chapter 10: Conclusion

In summary, this thesis researched the reproducibility and efficiency challenges in the field of DRL. Specifically, we identified 37 implementation details that are relevant to reproducing PPO's performance and discussed how they are often left out of the publication. To make DRL more reproducible and transparent, we propose a new framework that utilizes single-file implementations, and the framework is encapsulated in a DRL library called `CleanRL`. We further investigated and identify the reproducibility issues in the context of distributed DRL and proposed a new architecture that is highly reproducible and performs competitively with existing distributed DRL systems.

Furthermore, we addressed the efficiency challenge in the RL by creating a new testbed called Gym-μRTS, which we utilize to research different game representation designs and different methods to deal with large action spaces and sparse rewards. Through a series of work, we were able to produce a state-of-the-art Gym-μRTS that could defeat every μRTS bot we tested.

10.1 Contributions

This dissertation presents the following contributions:

1. We investigated and proposed a new framework in **Part I** of this thesis to address the reproducibility challenge in DRL algorithms.
   (a) In **Chapter 3**, we studied Proximal Policy Optimization (PPO) and identified 37 implementation details relevant to reproducing PPO's performance, whereas many of these implementation details are left out of PPO's original paper.
   (b) In **Chapter 4**, we introduced the `CleanRL` library, which promotes reproducibility and transparency in the implementation details of DRL algorithms.
   (c) In **Chapter 5**, we demonstrated reproducibility issues in distributed DRL and proposed the Cleanba architecture, which ensures reproducibility in different hardware configurations.
2. We addressed the efficiency challenge in **Part II** of this thesis through the development of more efficient DRL testbeds and exploration of efficient DRL techniques. In particular, we introduced Gym-μRTS, an efficient RL interface to the μRTS testbed, capturing the core challenges of RTS games.
   (a) In **Chapter 6**, we compared different observation and action space representations for μRTS.
   (b) In **Chapter 7**, we provided a detailed description and theoretical foundation of the optimization technique called invalid action masking.
   (c) In **Chapter 8**, we proposed a novel method of "action guidance" for better leveraging shaped rewards and sparse rewards simultaneously.
   (d) In **Chapter 9**, we scaled our RL-based approach to the full-game mode of μRTS, allowing the RL agent to control all the player-owned units concurrently.

10.2 Future Work

There are many promising directions for future work, which include

1. **Selfplay.** Our work with Gym-μRTS has primarily relied on custom reward functions and human-engineered bots. human-engineered bots are especially difficult to acquire. It would be valuable to train agents via self-play like done in AlphaStar[27], so we can remove the requirement of human-engineered bots.

2. **Partial Observability.** Investigating partial observability mode of Gym-μRTS. So far our research has focused primarily on fully observability mode, but real-world RTS games such as StarCraft II are partially observable. It would be interesting to investigate methods that could encourage the exploration of DRL-based agents in the partial observability mode.

3. **Generalization.** Generalization of unseen scenarios is an especially challenging topic in DRL. We investigated the agent's generalization ability into different maps in a paper not presented in this thesis[42] and found the agent's low-level skill, such as harvesting, still transfers to unseen maps. However, high-level skills such as strategizing are completely broken. Researching methods that could better generalize to different map settings would be a valuable future work.

Part III

Appendix

Appendix A: CleanRL

A.1 Benchmark experiments

As mentioned in Section 4.4, we rigorously benchmark our single-file implementations to validate their quality. Below are the tables that compare performance against reputable resources when applicable, where the reported numbers are the final average episodic returns of at least 3 random seeds. For more detailed information, see the main documentation site (https://docs.cleanrl.dev/).

A.1.1 Proximal Policy Optimization Variants and Performance

Environment	`ppo.py`	`openai/baselies`' PPO[22]
CartPole-v1	492.40 ± 13.05	497.54 ± 4.02
Acrobot-v1	-89.93 ± 6.34	-81.82 ± 5.58
MountainCar-v0	-200.00 ± 0.00	-200.00 ± 0.00

Environment	`ppo_atari.py`	`openai/baselies`' PPO (Huang et al., 2022)[^1]
BreakoutNoFrameskip-v4	416.31 ± 43.92	406.57 ± 31.554
PongNoFrameskip-v4	20.59 ± 0.35	20.512 ± 0.50
BeamRiderNoFrameskip-v4	2445.38 ± 528.91	2642.97 ± 670.37

Environment	`ppo_continuous_action.py`	`openai/baselies`' PPO[22]
Hopper-v2	2231.12 ± 656.72	2518.95 ± 850.46
Walker2d-v2	3050.09 ± 1136.21	3208.08 ± 1264.37
HalfCheetah-v2	1822.82 ± 928.11	2152.26 ± 1159.84

Environment	`ppo_atari_lstm.py`	`openai/baselies`' PPO[22]
BreakoutNoFrameskip-v4	128.92 ± 31.10	138.98 ± 50.76
PongNoFrameskip-v4	19.78 ± 1.58	19.79 ± 0.67
BeamRiderNoFrameskip-v4	1536.20 ± 612.21	1591.68 ± 372.95

Environment	`ppo_atari_envpool.py` (80 mins)	`ppo_atari.py` (220 mins)
BreakoutNoFrameskip-v4	389.57 ± 29.62	416.31 ± 43.92
PongNoFrameskip-v4	20.55 ± 0.37	20.59 ± 0.35
BeamRiderNoFrameskip-v4	2039.83 ± 1146.62	2445.38 ± 528.91

Environment	`ppo_procgen.py`	`openai/baselies`' PPO[22]
StarPilot	31.40 ± 11.73	33.97 ± 7.86
BossFight	9.09 ± 2.35	9.35 ± 2.04
BigFish	21.44 ± 6.73	20.06 ± 5.34

Environment	`ppo_atari_multigpu.py` (160 mins)	`ppo_atari.py` (215 mins)
BreakoutNoFrameskip-v4	429.06 ± 52.09	416.31 ± 43.92
PongNoFrameskip-v4	20.40 ± 0.46	20.59 ± 0.35
BeamRiderNoFrameskip-v4	2454.54 ± 740.49	2445.38 ± 528.91

The following table for `ppo_pettingzoo_ma_atari.py` reports the *episodic length* instead of *episodic return*:

Environment	`ppo_pettingzoo_ma_atari.py` (160 mins)
pong_v3	4153.60 ± 190.80
surround_v2	3055.33 ± 223.68
tennis_v3	14538.02 ± 7005.54

A.1.2 Deep Deterministic Policy Gradient Variant and Performance

Environment	`ddpg_continuous_action.py`	`OurDDPG.py`[70] Tab. 1	`DDPG.py` using settings from[69] in[70] Tab. 1
HalfCheetah	9382.32 ± 1395.52	8577.29	3305.60
Walker2d	1598.35 ± 862.66	3098.11	1843.85
Hopper	1313.43 ± 684.46	1860.02	2020.46

A.1.3 Twin-Delayed Deep Deterministic Policy Gradient Variant and Performance

Environment	`td3_continuous_action.py`	`TD3.py`[70] Tab. 1
HalfCheetah	9018.31 ± 1078.31	9636.95 ± 859.065
Walker2d	4246.07 ± 1210.84	4682.82 ± 539.64
Hopper	3391.78 ± 232.21	3564.07 ± 114.74

A.1.4 Soft Actor-Critic Variant and Performance

Environment	`sac_continuous_action.py`	Haarnoja et al.[87]
HalfCheetah-v2	10310.37 ± 1873.21	∼11,250
Walker2d-v2	4418.15 ± 592.82	∼4,800
Hopper-v2	2685.76 ± 762.16	∼3,250

A.1.5 Phasic Policy Gradient Variant and Performance

Environment	`ppg_procgen.py`	`ppo_procgen.py`	`openai/phasic-policy-gradient`
Starpilot (easy)	35.19 ± 13.07	33.15 ± 11.99	42.01 ± 9.59
Bossfight (easy)	10.34 ± 2.27	9.48 ± 2.42	10.71 ± 2.05
Bigfish (easy)	27.25 ± 7.55	22.21 ± 7.42	15.94 ± 10.80

A.1.6 Deep Q-learning Variants and Performance

Environment	`dqn_atari.py` 10M steps	Mnih et al.[11] 50M steps	Hessel et al.[23], Fig. 5
BreakoutNoFrameskip-v4	366.928 ± 39.89	401.2 ± 26.9	∼230 (10M steps) ∼300 (50M steps)
PongNoFrameskip-v4	20.25 ± 0.41	18.9 ± 1.3	∼20 (10M steps) ∼20 (50M steps)
BeamRiderNoFrameskip-v4	6673.24 ± 1434.37	6846 ± 1619	∼6000 (10M steps) ∼7000 (50M steps)

Environment	`dqn.py`
CartPole-v1	488.69 ± 16.11
Acrobot-v1	-91.54 ± 7.20
MountainCar-v0	-194.95 ± 8.48

A.1.7 Categorical Deep Q-learning Variants and Performance

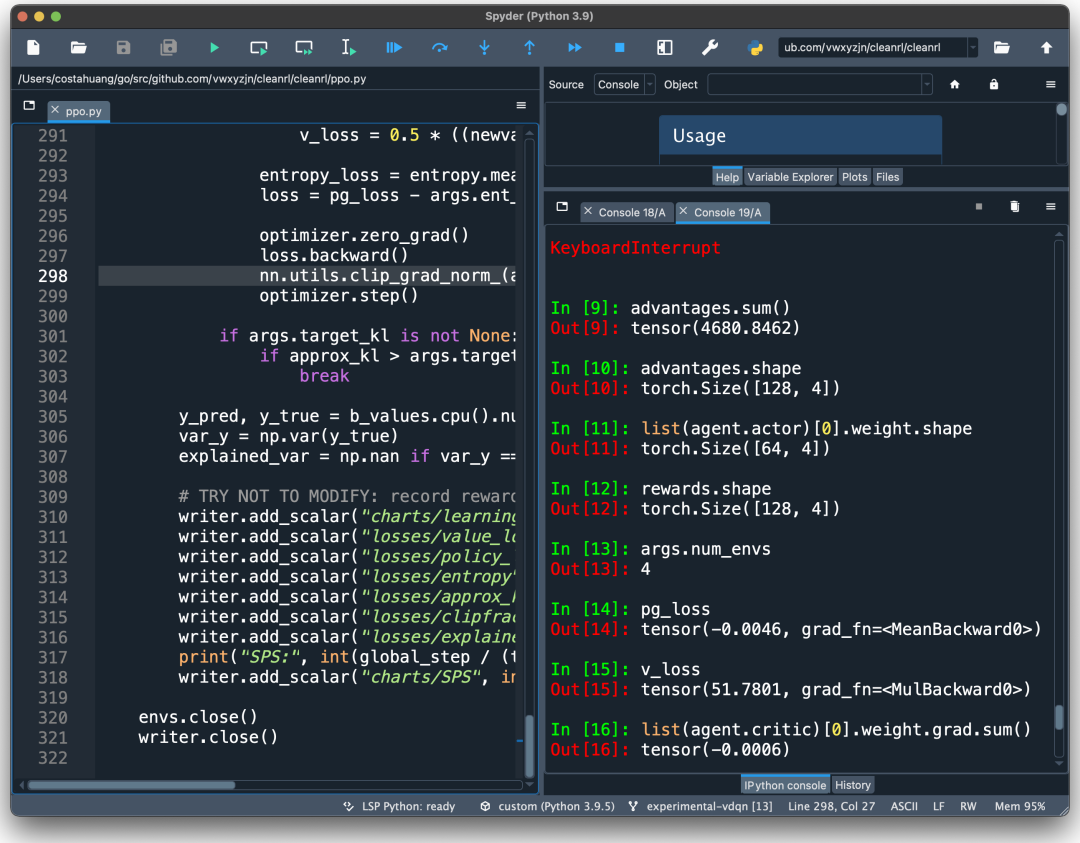
Environment	`c51_atari.py` 10M steps	Bellemare et al.[12], Fig. 14 50M steps	Hessel et al.[23], Fig. 5
BreakoutNoFrameskip-v4	461.86 ± 69.65	748	∼500 (10M steps) ∼600 (50M steps)
PongNoFrameskip-v4	19.46 ± 0.70	20.9	∼20 (10M steps) ∼20 (50M steps)
BeamRiderNoFrameskip-v4	9592.90 ± 2270.15	14,074	∼12000 (10M steps) ∼14000 (50M steps)

Environment	`c51.py`
CartPole-v1	481.20 ± 20.53
Acrobot-v1	-87.70 ± 5.52
MountainCar-v0	-166.38 ± 27.94

A.2 Interactive Shell

In `CleanRL`, we have put most of the variables in the *global python name scope*. This makes it easier to inspect the variables and their shapes. The following figure shows a screenshot of the Spyder

editor [1], where the code is on the left and the interactive shell is on the right. In the interactive shell, we can easily inspect the variables for debugging purposes without modifying the code.



A.3 Maintaining Single-file Implementations

Despite the many benefits that single-file implementations offer, one downside is excessive amount of duplicate code, which makes them difficult to maintain. To help address this challenge, we have adopted a series of development tools to reduce maintenance burden. These tools are:

1. **poetry** (https://python-poetry.org/): poetry is a dependency management tool that helps resolve and pins dependency versions. We use poetry to improve reproducibility and provide a smooth dependency installation experience. See our installation documentation (https://docs.cleanrl.dev/get-started/installation/) for more detail.

2. **pre-commit** (https://pre-commit.com/): pre-commit is a tool that helps us automate a sequence of short tasks (called pre-commit "hooks") such as code formatting. In particular, we always use the following hooks when submitting code to the main repository. See https://github.com/vwxyzjn/cleanrl/blob/master/CONTRIBUTING.md for more information.

   (a) **pyupgrade** (https://github.com/asottile/pyupgrade): pyupgrade upgrades syntax for newer versions of the language.

   (b) **isort** (https://github.com/PyCQA/isort): isort sorts imported dependencies according to their type (e.g, standard library vs third-party library) and name.

[1] https://www.spyder-ide.org/

   (c) **black** (https://black.readthedocs.io/en/stable/): black enforces an uniform code style across the codebase.

   (d) **autoflake** (https://github.com/PyCQA/autoflake): autoflake helps remove unused imports and variables.

   (e) **codespell** (https://github.com/codespell-project/codespell): codespell helps avoid common incorrect spelling.

3. **Docker** (https://www.docker.com/): docker helps us package the code into a container which can be used to orchestrate training in a reproducible way.

   (a) **AWS Batch** (https://aws.amazon.com/batch/): Amazon Web Services Batch could leverage our built containers to run thousands experiments concurrently.

   (b) We have built utilities to help package code into a container and submit to AWS Batch using a few lines of command. In 2020 alone, the authors have run over 50,000+ hours of experiments using this workflow. See https://docs.cleanrl.dev/cloud/installation/ for more documentation.

A.4 W&B Editing Panel

A screenshot of the W&B panel that allows the the users to change smoothing weight, add panels to show different metrics like losses, visualize the videos of the agents' gameplay, filter, group, sort, and search for desired experiments.

A.5 Stepping Through Stable-baselines 3 Code with a Debugger

In this section, we attempt to run the following Stable-baselines 3 (v1.5.0)[2] code with a debugger to identify the related modules.

```
from stable_baselines3.common.env_util import make_atari_env
from stable_baselines3.common.vec_env import VecFrameStack
from stable_baselines3 import PPO
env = make_atari_env('PongNoFrameskip-v4', n_envs=4, seed=0)
env = VecFrameStack(env, n_stack=4)
model = PPO('CnnPolicy', env, verbose=1)
model.learn(total_timesteps=25_000)
```

Here is the list of the related python files and their lines of code (LOC):

1. `stable_baselines3/ppo/ppo.py` - 315 LOC
2. `stable_baselines3/common/on_policy_algorithm.py` - 280 LOC
3. `stable_baselines3/common/base_class.py` - 819 LOC
4. `stable_baselines3/common/utils.py` - 506 LOC
5. `stable_baselines3/common/env_util.py` - 157 LOC
6. `stable_baselines3/common/atari_wrappers.py` - 249 LOC

[2] https://github.com/DLR-RM/stable-baselines3/releases/tag/v1.5.0

7. stable_baselines3/common/vec_env/__init__ - .py 73 LOC
8. stable_baselines3/common/vec_env/dummy_vec_env - .py 126 LOC
9. stable_baselines3/common/vec_env/base_vec_env - .py 375 LOC
10. stable_baselines3/common/vec_env/util - .py 77 LOC
11. stable_baselines3/common/vec_env/vec_frame_stack.py - 65 LOC
12. stable_baselines3/common/vec_env/stacked_observations.py - 267 LOC
13. stable_baselines3/common/preprocessing.py - 217 LOC
14. stable_baselines3/common/buffers.py - 770 LOC
15. stable_baselines3/common/policies.py - 962 LOC
16. stable_baselines3/common/torch_layers.py - 318 LOC
17. stable_baselines3/common/distributions.py - 700 LOC
18. stable_baselines3/common/monitor.py - 240 LOC
19. stable_baselines3/common/logger.py - 640 LOC
20. stable_baselines3/common/callbacks.py - 603 LOC

The total LOC involved is 7759. Notice we have labeled the popular utilities such as vectorized environments, Atari environment pre-processing wrappers, and episode statistics recording code with the blue color. This means the total LOC related to core PPO implementation **not** counting the blue color files is 6287.

Appendix B: Gym-μRTS

B.1 Estimated AlphaStar cost

We estimate the cost of AlphaStar based on the latest pricing information from Iowa (us-central1) zone on Google Cloud Platform (GCP). As of February. 17, 2021, on-demand TPU instances with 8 cores cost $8.00 per hour, and preemptible C2 instances costs $0.00822 per hour. Given this pricing, we get the training cost for AlphaStar is $((8/8) * 3072 + 0.00822 * 50400) * 44 * 24 = \$3,681,520.12$.

B.2 Learning curves and match results

All the learning curves related for UAS and Gridnet can be found at Figure B.1 and Figure B.2, respectively. Similarly, the match results can be found at Figure B.4 and Figure B.5.

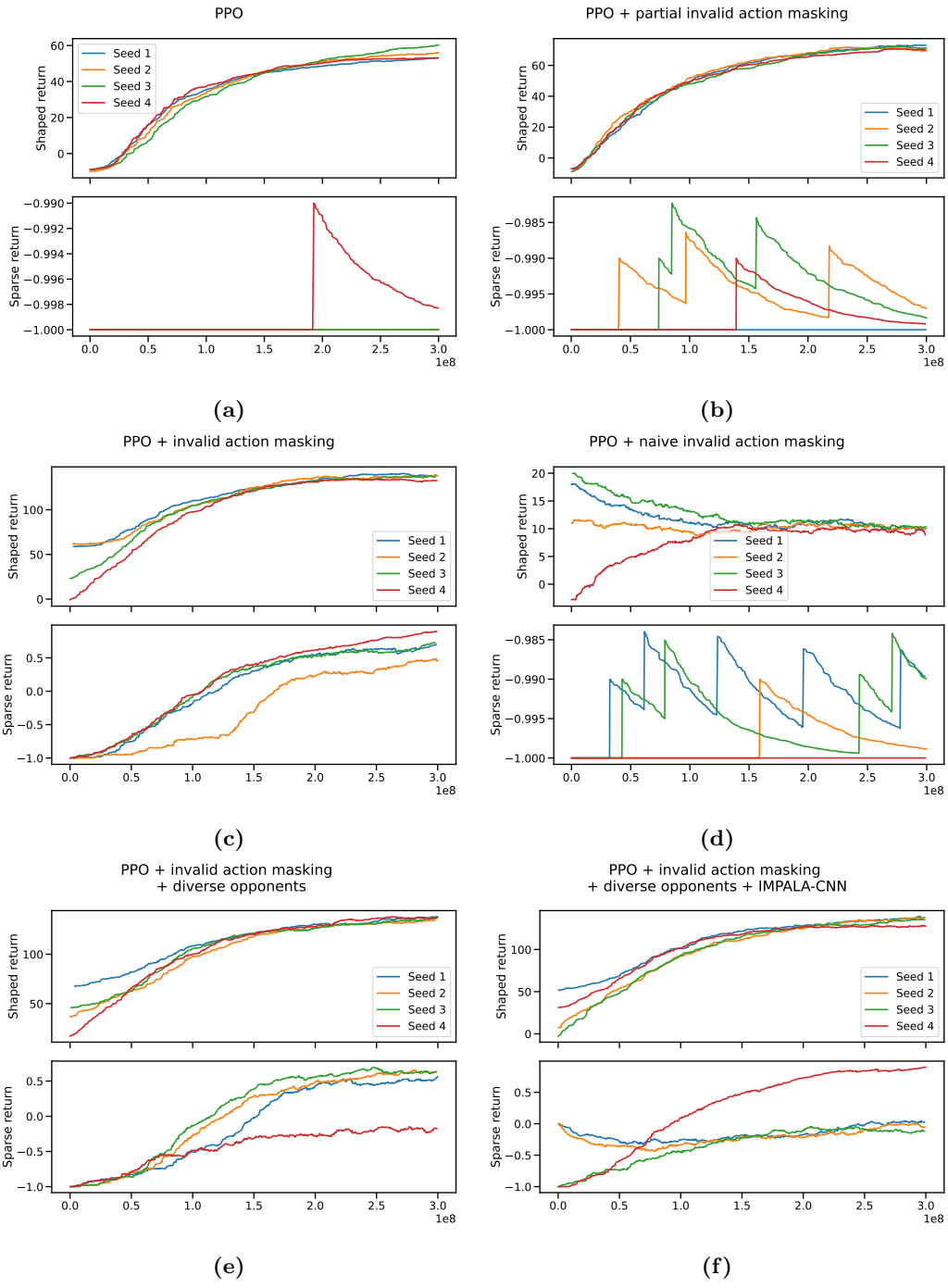


Figure B.1: UAS learning curves.

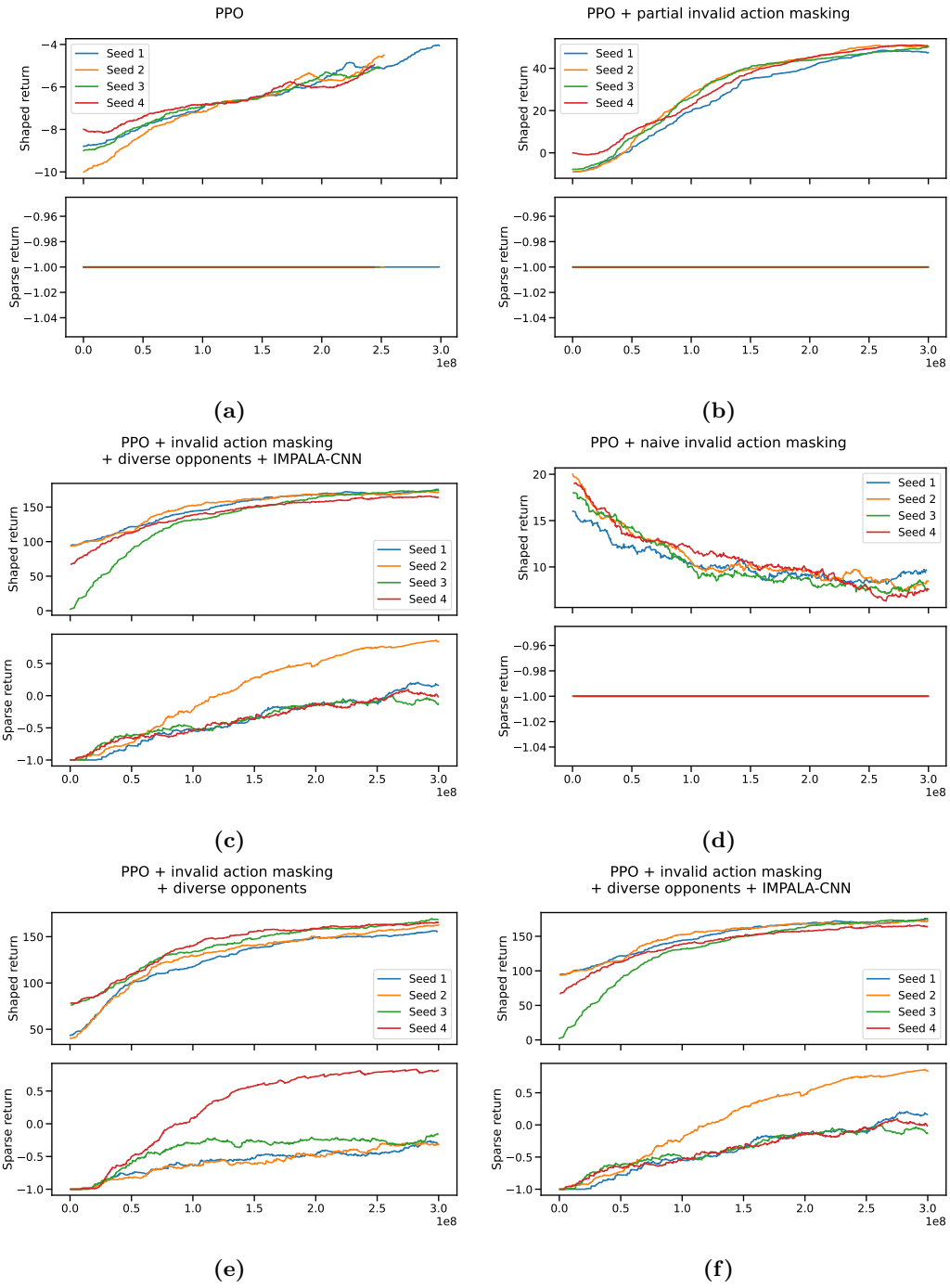
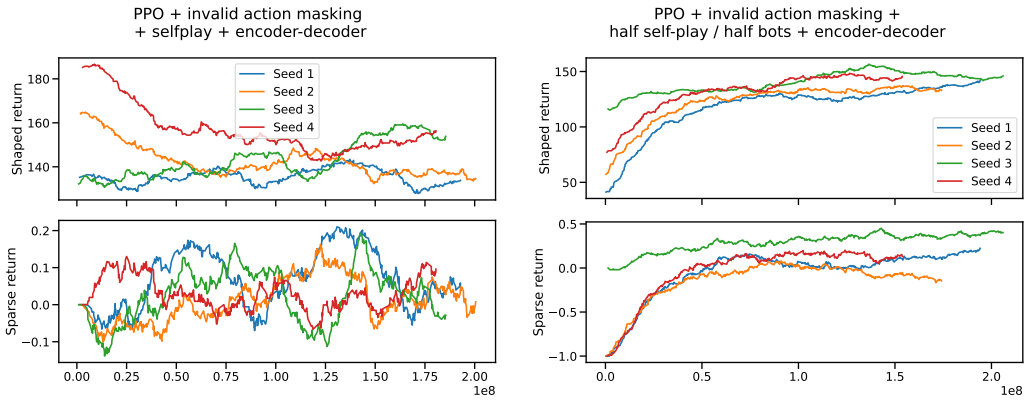


Figure B.2: Gridnet learning curves.



(a) **(b)**

Figure B.3: Gridnet selfplay learning curves.

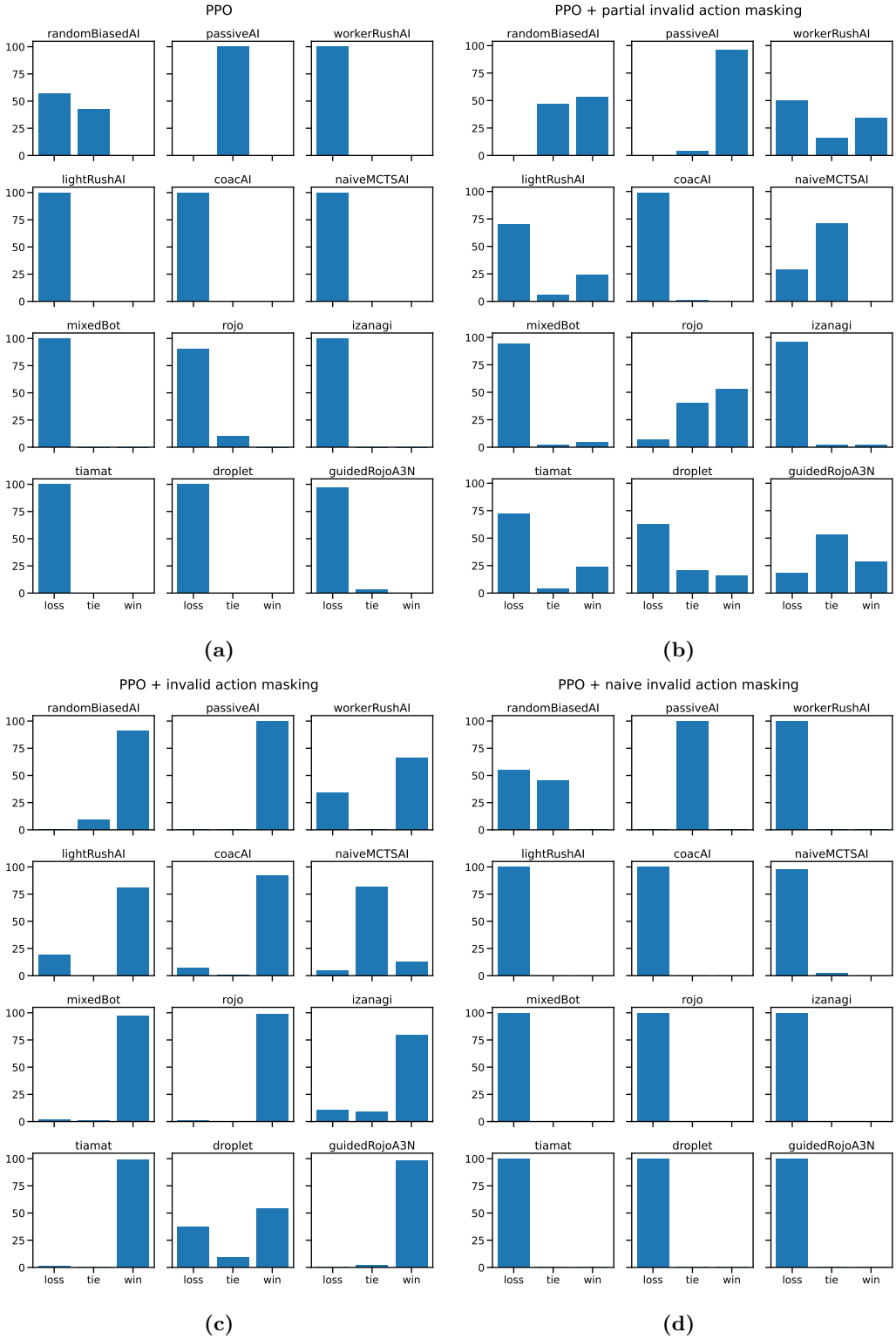
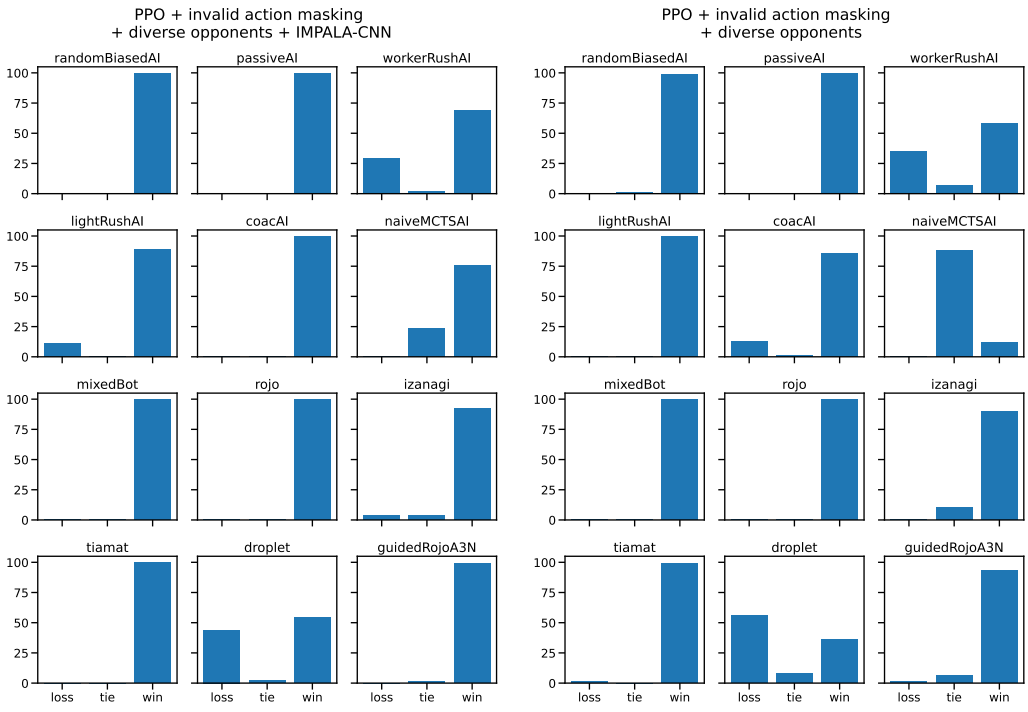


Figure B.4: UAS match results.



(e) **(f)**

Figure B.4: UAS match results.

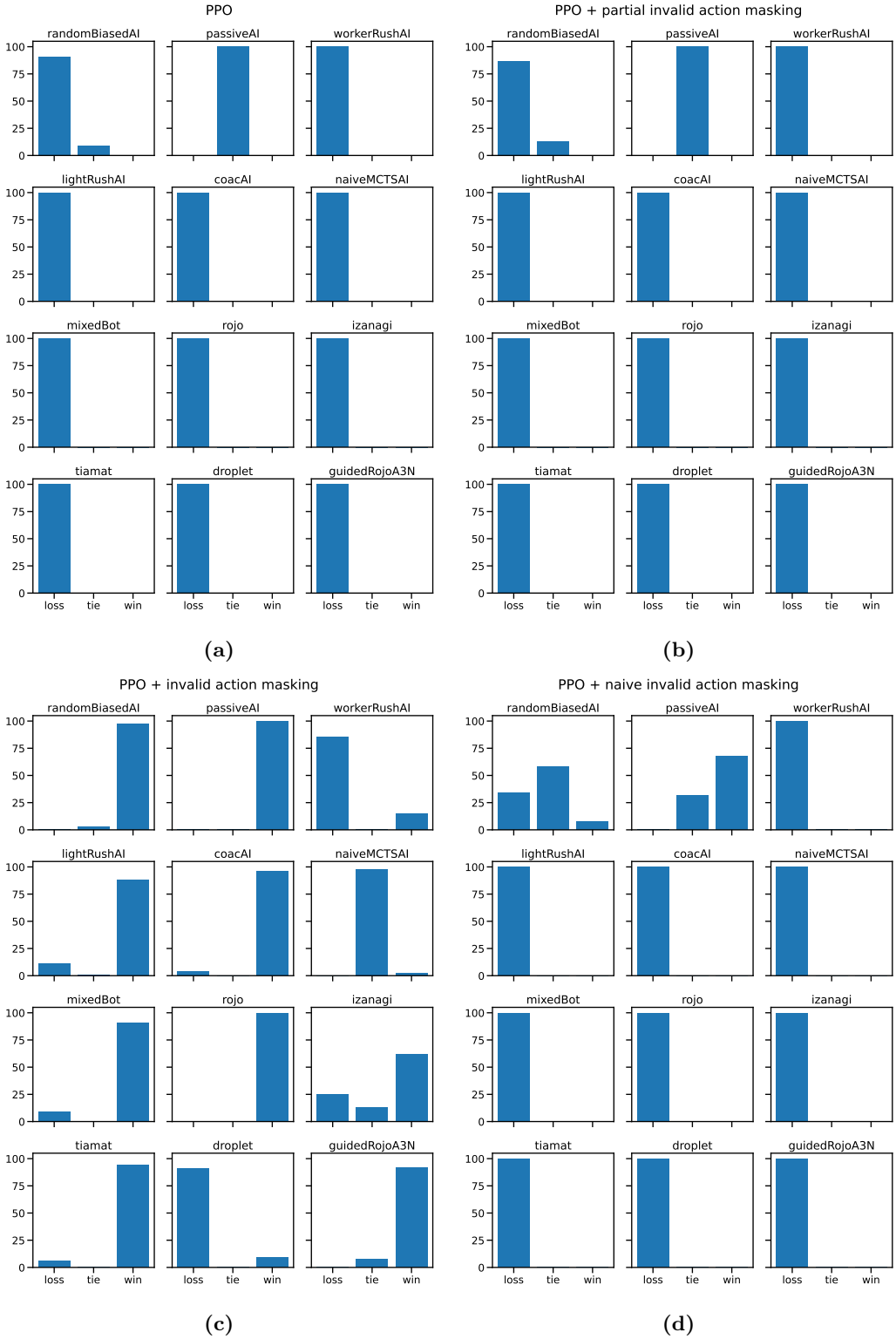


Figure B.5: Gridnet match results.

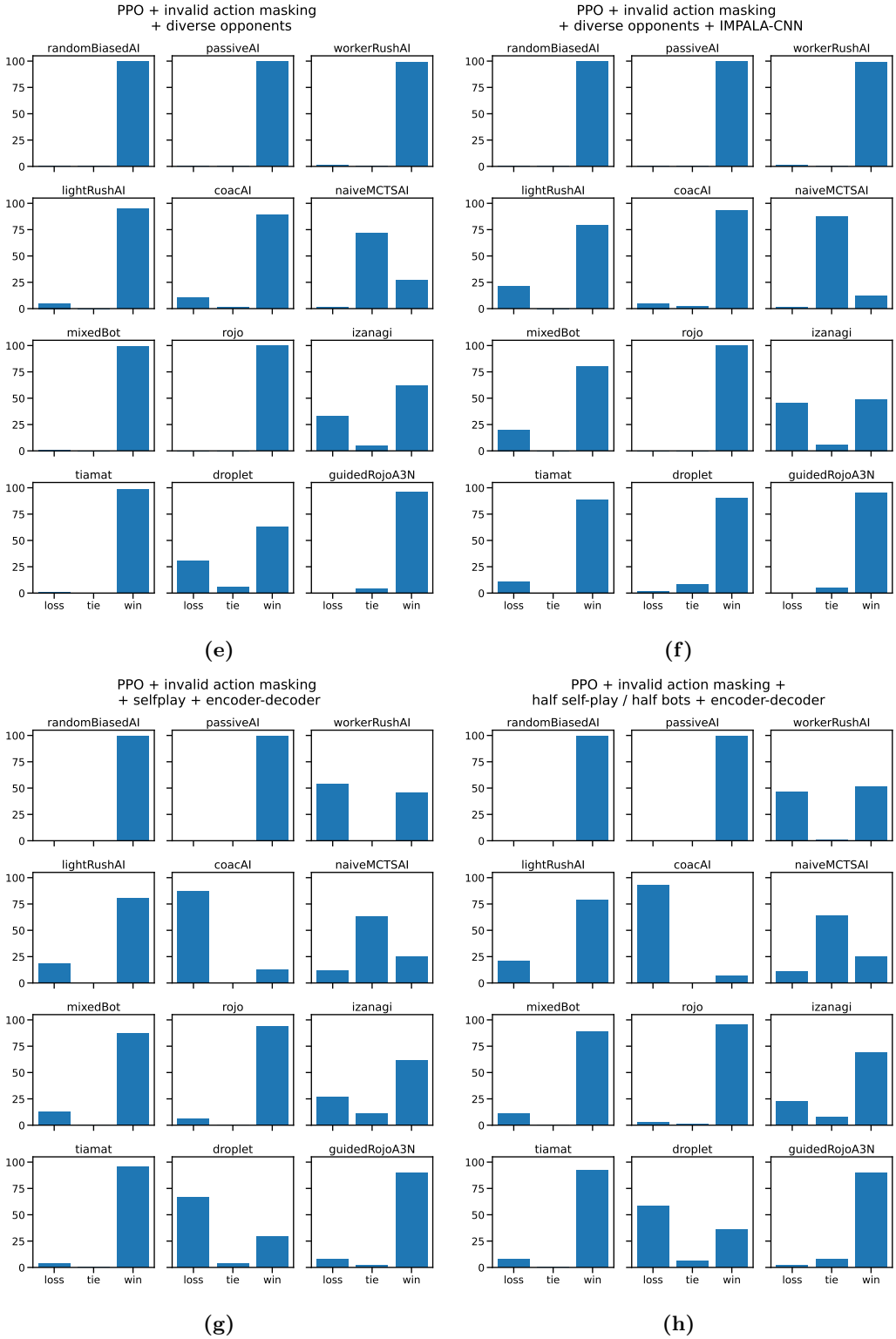


Figure B.5: Gridnet match results.

Appendix C: Cleanba

C.1 Detailed experiment settings

For the experiments, the PPO and IMPALA's hyperparameters can be found in Table C.1 and C.2. The Vtrace implementation can be found in `rlax`[1]. The learning curves can be found at Figure C.1.

Table C.1: PPO hyperparameters.

Parameter Names	Parameter Values
N_{total} Total Time Steps	50,000,000
α Learning Rate	0.00025 Linearly Decreased to 0
N_{envs} Number of Environments	120
N_{steps} Number of Steps per Environment	128
γ (Discount Factor)	0.99
λ (for GAE)	0.95
N_{mb} Number of Mini-batches	4
K (Number of PPO Update Iteration Per Epoch)	4
ε (PPO's Clipping Coefficient)	0.1
c_1 (Value Function Coefficient)	0.5
c_2 (Entropy Coefficient)	0.01
ω (Gradient Norm Threshold)	0.5
Value Function Loss Clipping	False

Table C.2: IMPALA hyperparameters.

Parameter Names	Parameter Values
N_{total} Total Time Steps	50,000,000
α Learning Rate	0.00025 Linearly Decreased to 0
N_{envs} Number of Environments	128
N_{steps} Number of Steps per Environment	128
γ (Discount Factor)	0.99
λ (mixing parameter)	1.0
N_{mb} Number of Mini-batches	4
ρ (Clip Threshold for Importance Ratios)	1.0
ρ_{pg} (Clip Threshold for Policy Gradient Importance Ratios)	1.0
c_1 (Value Function Coefficient)	0.5
c_2 (Entropy Coefficient)	0.01
ω (Gradient Norm Threshold)	0.5

C.2 `moolib` Experiments

We conducted two sets of `moolib` experiments and reported the set with a lower median and higher IQM, as shown in Figure C.2 for legacy reasons. During our debugging, we found the Asteroids

[1] https://github.com/deepmind/rlax/blob/b53c6510c8b2cad6b106b6166e22aba61a77ee2f/rlax/_src/vtrace.py#L162-L193

experiments in the first set of `moolib` experiments to obtain high scores, but we ran Asteroids specifically for ten random seeds and found lower scores; this suggests the Asteroids experiments in the first set were likely due to lucky random seeds, so we re-run the `moolib` experiments.

C.3 torchbeast logs

```
$ python -m torchbeast.monobeast_study \
--num_actors 80 \
--total_steps 10000000 \
--learning_rate 0.0006 \
--epsilon 0.01 \
--entropy_cost 0.01 \
--batch_size 8 \
--unroll_length 240 \
--num_threads 1 \
--env Pong-v5
actor_index 32 initial policy_version 8 policy_version after rollout 20
actor_index 13 initial policy_version 8 policy_version after rollout 20
actor_index 57 initial policy_version 8 policy_version after rollout 20
actor_index 12 initial policy_version 8 policy_version after rollout 21
actor_index 51 initial policy_version 8 policy_version after rollout 21
actor_index 2 initial policy_version 8 policy_version after rollout 21
actor_index 56 initial policy_version 8 policy_version after rollout 21
actor_index 38 initial policy_version 9 policy_version after rollout 21
actor_index 37 initial policy_version 9 policy_version after rollout 22
actor_index 59 initial policy_version 9 policy_version after rollout 22
actor_index 9 initial policy_version 9 policy_version after rollout 22
actor_index 69 initial policy_version 9 policy_version after rollout 22
actor_index 35 initial policy_version 9 policy_version after rollout 22
actor_index 66 initial policy_version 9 policy_version after rollout 22
actor_index 10 initial policy_version 9 policy_version after rollout 22
actor_index 55 initial policy_version 10 policy_version after rollout 22
actor_index 53 initial policy_version 10 policy_version after rollout 22
actor_index 46 initial policy_version 10 policy_version after rollout 22
actor_index 54 initial policy_version 10 policy_version after rollout 23
actor_index 50 initial policy_version 10 policy_version after rollout 23
actor_index 8 initial policy_version 10 policy_version after rollout 23
actor_index 64 initial policy_version 10 policy_version after rollout 23
actor_index 77 initial policy_version 10 policy_version after rollout 23
actor_index 3 initial policy_version 11 policy_version after rollout 23
actor_index 7 initial policy_version 11 policy_version after rollout 23
actor_index 28 initial policy_version 11 policy_version after rollout 23
actor_index 49 initial policy_version 11 policy_version after rollout 23
actor_index 16 initial policy_version 11 policy_version after rollout 23
actor_index 24 initial policy_version 11 policy_version after rollout 23
actor_index 11 initial policy_version 11 policy_version after rollout 23
actor_index 14 initial policy_version 11 policy_version after rollout 23
actor_index 43 initial policy_version 13 policy_version after rollout 26
actor_index 58 initial policy_version 13 policy_version after rollout 26
actor_index 23 initial policy_version 13 policy_version after rollout 26
actor_index 29 initial policy_version 13 policy_version after rollout 26
actor_index 68 initial policy_version 13 policy_version after rollout 26
actor_index 75 initial policy_version 14 policy_version after rollout 26
actor_index 48 initial policy_version 14 policy_version after rollout 27
actor_index 67 initial policy_version 14 policy_version after rollout 27
actor_index 5 initial policy_version 14 policy_version after rollout 27
actor_index 18 initial policy_version 14 policy_version after rollout 27
```

```
actor_index 41 initial policy_version 15 policy_version after rollout 27
actor_index 78 initial policy_version 14 policy_version after rollout 27
actor_index 15 initial policy_version 15 policy_version after rollout 27
actor_index 34 initial policy_version 15 policy_version after rollout 27
actor_index 45 initial policy_version 15 policy_version after rollout 28
actor_index 22 initial policy_version 15 policy_version after rollout 28
actor_index 4 initial policy_version 16 policy_version after rollout 28
actor_index 6 initial policy_version 16 policy_version after rollout 28
actor_index 20 initial policy_version 16 policy_version after rollout 28
actor_index 39 initial policy_version 16 policy_version after rollout 28
actor_index 33 initial policy_version 16 policy_version after rollout 29
actor_index 74 initial policy_version 16 policy_version after rollout 29
actor_index 60 initial policy_version 16 policy_version after rollout 29
actor_index 42 initial policy_version 17 policy_version after rollout 29
actor_index 72 initial policy_version 17 policy_version after rollout 30
actor_index 25 initial policy_version 17 policy_version after rollout 30
actor_index 31 initial policy_version 17 policy_version after rollout 30
actor_index 19 initial policy_version 17 policy_version after rollout 30
actor_index 1 initial policy_version 18 policy_version after rollout 31
actor_index 79 initial policy_version 18 policy_version after rollout 31
actor_index 65 initial policy_version 18 policy_version after rollout 31
actor_index 73 initial policy_version 18 policy_version after rollout 31
actor_index 36 initial policy_version 18 policy_version after rollout 31
actor_index 21 initial policy_version 18 policy_version after rollout 31
actor_index 0 initial policy_version 18 policy_version after rollout 31
actor_index 30 initial policy_version 18 policy_version after rollout 31
actor_index 44 initial policy_version 18 policy_version after rollout 31
actor_index 63 initial policy_version 19 policy_version after rollout 31
actor_index 76 initial policy_version 19 policy_version after rollout 32
actor_index 47 initial policy_version 19 policy_version after rollout 32
actor_index 52 initial policy_version 19 policy_version after rollout 32
actor_index 26 initial policy_version 19 policy_version after rollout 32
actor_index 71 initial policy_version 19 policy_version after rollout 32
actor_index 70 initial policy_version 19 policy_version after rollout 32
actor_index 17 initial policy_version 20 policy_version after rollout 32
actor_index 62 initial policy_version 20 policy_version after rollout 33
actor_index 40 initial policy_version 20 policy_version after rollout 33
actor_index 27 initial policy_version 20 policy_version after rollout 33
actor_index 13 initial policy_version 20 policy_version after rollout 33
actor_index 57 initial policy_version 20 policy_version after rollout 33
actor_index 32 initial policy_version 20 policy_version after rollout 33
actor_index 51 initial policy_version 21 policy_version after rollout 33
actor_index 61 initial policy_version 20 policy_version after rollout 33
actor_index 2 initial policy_version 21 policy_version after rollout 33
actor_index 56 initial policy_version 21 policy_version after rollout 34
actor_index 12 initial policy_version 21 policy_version after rollout 34


$ python -m torchbeast.monobeast_study \
--num_actors 80 \
--total_steps 10000000 \
--learning_rate 0.0006 \
--epsilon 0.01 \
--entropy_cost 0.01 \
```

```
--batch_size 8 \
--unroll_length 240 \
--num_threads 1 \
--env Pong-v5  \
--learner_delay_seconds 1.0

actor_index 72 initial policy_version 9 policy_version after rollout 10
actor_index 22 initial policy_version 9 policy_version after rollout 10
actor_index 37 initial policy_version 9 policy_version after rollout 10
actor_index 41 initial policy_version 9 policy_version after rollout 10
actor_index 16 initial policy_version 9 policy_version after rollout 10
actor_index 61 initial policy_version 10 policy_version after rollout 11
actor_index 18 initial policy_version 10 policy_version after rollout 11
actor_index 13 initial policy_version 10 policy_version after rollout 11
actor_index 56 initial policy_version 10 policy_version after rollout 11
actor_index 28 initial policy_version 10 policy_version after rollout 11
actor_index 4 initial policy_version 10 policy_version after rollout 11
actor_index 7 initial policy_version 10 policy_version after rollout 11
actor_index 65 initial policy_version 10 policy_version after rollout 11
actor_index 12 initial policy_version 11 policy_version after rollout 12
actor_index 14 initial policy_version 11 policy_version after rollout 12
actor_index 5 initial policy_version 11 policy_version after rollout 12
actor_index 3 initial policy_version 11 policy_version after rollout 12
actor_index 35 initial policy_version 11 policy_version after rollout 12
actor_index 51 initial policy_version 11 policy_version after rollout 12
actor_index 0 initial policy_version 11 policy_version after rollout 12
actor_index 6 initial policy_version 11 policy_version after rollout 12
actor_index 60 initial policy_version 12 policy_version after rollout 13
actor_index 77 initial policy_version 12 policy_version after rollout 13
actor_index 48 initial policy_version 12 policy_version after rollout 13

$ python -m torchbeast.monobeast_study \
--num_actors 40 \
--total_steps 10000000 \
--learning_rate 0.0006 \
--epsilon 0.01 \
--entropy_cost 0.01 \
--batch_size 8 \
--unroll_length 240 \
--num_threads 1 \
--env Pong-v5

actor_index 34 initial policy_version 12 policy_version after rollout 18
actor_index 25 initial policy_version 13 policy_version after rollout 18
actor_index 4 initial policy_version 13 policy_version after rollout 18
actor_index 5 initial policy_version 13 policy_version after rollout 18
actor_index 14 initial policy_version 13 policy_version after rollout 18
actor_index 16 initial policy_version 13 policy_version after rollout 18
actor_index 12 initial policy_version 13 policy_version after rollout 18
actor_index 39 initial policy_version 13 policy_version after rollout 18
actor_index 30 initial policy_version 13 policy_version after rollout 18
actor_index 18 initial policy_version 13 policy_version after rollout 18
actor_index 13 initial policy_version 13 policy_version after rollout 18
actor_index 23 initial policy_version 13 policy_version after rollout 19
```

```
actor_index 35 initial policy_version 13 policy_version after rollout 19
actor_index 3 initial policy_version 14 policy_version after rollout 19
actor_index 17 initial policy_version 14 policy_version after rollout 19
actor_index 9 initial policy_version 14 policy_version after rollout 19
actor_index 6 initial policy_version 14 policy_version after rollout 19
```

C.4 Large Batch Size Training

Cleanba can also scale to the hundreds of GPUs in multi-host and multi-process environments by leveraging the `jax.distributed` package, allowing us to explore training with even larger batch sizes. We conduct experiments with 16, 32, 64, and 128 A100 GPUs. For convenience, we also adjust a few settings: 1) turn off the learning rate annealing, 2) run for 100M steps instead of the standard 50M steps, and 3) keep doubling the `num_envs`, `batch_size`, and `minibatch_size` with a larger number of GPUs.

Due to hardware scheduling constraints, we only ran the experiments for 1 random seed. The results are shown in Figure C.3. We make the following observations:

- **Linear scaling w/ 93% of ideal scaling efficiency.** As we increased the number of GPUs to 16, 32, 64, 128, we observed a linear scaling in steps per second (SPS) in Cleanba achieving 93% of the ideal scaling efficiency. This is likely empowered by the fast connectivity offered by NVIDIA GPUDirect RDMA (remote direct memory access) in Stability AI's HPC. When using 128 GPUs, the agent has an SPS of 403253, translating to over *1.6M FPS* in Breakout.
- **Small batch sizes train more efficiently.** As we increase batch sizes, particularly in the first 40M steps, the sample efficiency tends to decline. This outcome is unsurprising, given that the initial policy is random and Breakout initially has limited explorable game states. In this case, the data in the batch is going to have less diverse data, which makes the large batch size less valuable.
- **Large batch sizes train more quickly.** Like[156], we find increasing the batch size does make the agent reach some given scores faster. This suggests that we could always increase the batch size to obtain shorter training times if sample efficiency is not a concern.

While we observed limited benefits of scaling Cleanba to use 128 GPUs, the objective of the scaling experiments is to show we can scale to large batch sizes. Given a more challenging task, the training data is likely going to be more diverse and have a higher *gradient noise scale*[156], which would help the agent utilize large batch sizes more efficiently, resulting in a reduced decline in sample efficiency.

C.5 torchbeast logs

```
# poetry run python -m torchbeast.monobeast_study_new --exp-name monobeast_cpu80_unroll_length1
actor_index 32 initial policy_version 8 policy_version after rollout 20
actor_index 13 initial policy_version 8 policy_version after rollout 20
actor_index 57 initial policy_version 8 policy_version after rollout 20
actor_index 12 initial policy_version 8 policy_version after rollout 21
actor_index 51 initial policy_version 8 policy_version after rollout 21
actor_index 2 initial policy_version 8 policy_version after rollout 21
actor_index 56 initial policy_version 8 policy_version after rollout 21
actor_index 38 initial policy_version 9 policy_version after rollout 21
actor_index 37 initial policy_version 9 policy_version after rollout 22
actor_index 59 initial policy_version 9 policy_version after rollout 22
actor_index 9 initial policy_version 9 policy_version after rollout 22
actor_index 69 initial policy_version 9 policy_version after rollout 22
```

```
actor_index 35 initial policy_version 9 policy_version after rollout 22
actor_index 66 initial policy_version 9 policy_version after rollout 22
actor_index 10 initial policy_version 9 policy_version after rollout 22
actor_index 55 initial policy_version 10 policy_version after rollout 22
actor_index 53 initial policy_version 10 policy_version after rollout 22
actor_index 46 initial policy_version 10 policy_version after rollout 22
actor_index 54 initial policy_version 10 policy_version after rollout 23
actor_index 50 initial policy_version 10 policy_version after rollout 23
actor_index 8 initial policy_version 10 policy_version after rollout 23
actor_index 64 initial policy_version 10 policy_version after rollout 23
actor_index 77 initial policy_version 10 policy_version after rollout 23
actor_index 3 initial policy_version 11 policy_version after rollout 23
actor_index 7 initial policy_version 11 policy_version after rollout 23
actor_index 28 initial policy_version 11 policy_version after rollout 23
actor_index 49 initial policy_version 11 policy_version after rollout 23
actor_index 16 initial policy_version 11 policy_version after rollout 23
actor_index 24 initial policy_version 11 policy_version after rollout 23
actor_index 11 initial policy_version 11 policy_version after rollout 23
actor_index 14 initial policy_version 11 policy_version after rollout 23
actor_index 43 initial policy_version 13 policy_version after rollout 26
actor_index 58 initial policy_version 13 policy_version after rollout 26
actor_index 23 initial policy_version 13 policy_version after rollout 26
actor_index 29 initial policy_version 13 policy_version after rollout 26
actor_index 68 initial policy_version 13 policy_version after rollout 26
actor_index 75 initial policy_version 14 policy_version after rollout 26
actor_index 48 initial policy_version 14 policy_version after rollout 27
actor_index 67 initial policy_version 14 policy_version after rollout 27
actor_index 5 initial policy_version 14 policy_version after rollout 27
actor_index 18 initial policy_version 14 policy_version after rollout 27
actor_index 41 initial policy_version 15 policy_version after rollout 27
actor_index 78 initial policy_version 14 policy_version after rollout 27
actor_index 15 initial policy_version 15 policy_version after rollout 27
actor_index 34 initial policy_version 15 policy_version after rollout 27
actor_index 45 initial policy_version 15 policy_version after rollout 28
actor_index 22 initial policy_version 15 policy_version after rollout 28
actor_index 4 initial policy_version 16 policy_version after rollout 28
actor_index 6 initial policy_version 16 policy_version after rollout 28
actor_index 20 initial policy_version 16 policy_version after rollout 28
actor_index 39 initial policy_version 16 policy_version after rollout 28
actor_index 33 initial policy_version 16 policy_version after rollout 29
actor_index 74 initial policy_version 16 policy_version after rollout 29
actor_index 60 initial policy_version 16 policy_version after rollout 29
actor_index 42 initial policy_version 17 policy_version after rollout 29
actor_index 72 initial policy_version 17 policy_version after rollout 30
actor_index 25 initial policy_version 17 policy_version after rollout 30
actor_index 31 initial policy_version 17 policy_version after rollout 30
actor_index 19 initial policy_version 17 policy_version after rollout 30
actor_index 1 initial policy_version 18 policy_version after rollout 31
actor_index 79 initial policy_version 18 policy_version after rollout 31
actor_index 65 initial policy_version 18 policy_version after rollout 31
actor_index 73 initial policy_version 18 policy_version after rollout 31
actor_index 36 initial policy_version 18 policy_version after rollout 31
actor_index 21 initial policy_version 18 policy_version after rollout 31
actor_index 0 initial policy_version 18 policy_version after rollout 31
```

```
actor_index 30 initial policy_version 18 policy_version after rollout 31
actor_index 44 initial policy_version 18 policy_version after rollout 31
actor_index 63 initial policy_version 19 policy_version after rollout 31
actor_index 76 initial policy_version 19 policy_version after rollout 32
actor_index 47 initial policy_version 19 policy_version after rollout 32
actor_index 52 initial policy_version 19 policy_version after rollout 32
actor_index 26 initial policy_version 19 policy_version after rollout 32
actor_index 71 initial policy_version 19 policy_version after rollout 32
actor_index 70 initial policy_version 19 policy_version after rollout 32
actor_index 17 initial policy_version 20 policy_version after rollout 32
actor_index 62 initial policy_version 20 policy_version after rollout 33
actor_index 40 initial policy_version 20 policy_version after rollout 33
actor_index 27 initial policy_version 20 policy_version after rollout 33
actor_index 13 initial policy_version 20 policy_version after rollout 33
actor_index 57 initial policy_version 20 policy_version after rollout 33
actor_index 32 initial policy_version 20 policy_version after rollout 33
actor_index 51 initial policy_version 21 policy_version after rollout 33
actor_index 61 initial policy_version 20 policy_version after rollout 33
actor_index 2 initial policy_version 21 policy_version after rollout 33
actor_index 56 initial policy_version 21 policy_version after rollout 34
actor_index 12 initial policy_version 21 policy_version after rollout 34


# poetry run python -m torchbeast.monobeast_study_new --exp-name monobeast_cpu80_unroll_length2

actor_index 72 initial policy_version 9 policy_version after rollout 10
actor_index 22 initial policy_version 9 policy_version after rollout 10
actor_index 37 initial policy_version 9 policy_version after rollout 10
actor_index 41 initial policy_version 9 policy_version after rollout 10
actor_index 16 initial policy_version 9 policy_version after rollout 10
actor_index 61 initial policy_version 10 policy_version after rollout 11
actor_index 18 initial policy_version 10 policy_version after rollout 11
actor_index 13 initial policy_version 10 policy_version after rollout 11
actor_index 56 initial policy_version 10 policy_version after rollout 11
actor_index 28 initial policy_version 10 policy_version after rollout 11
actor_index 4 initial policy_version 10 policy_version after rollout 11
actor_index 7 initial policy_version 10 policy_version after rollout 11
actor_index 65 initial policy_version 10 policy_version after rollout 11
actor_index 12 initial policy_version 11 policy_version after rollout 12
actor_index 14 initial policy_version 11 policy_version after rollout 12
actor_index 5 initial policy_version 11 policy_version after rollout 12
actor_index 3 initial policy_version 11 policy_version after rollout 12
actor_index 35 initial policy_version 11 policy_version after rollout 12
actor_index 51 initial policy_version 11 policy_version after rollout 12
actor_index 0 initial policy_version 11 policy_version after rollout 12
actor_index 6 initial policy_version 11 policy_version after rollout 12
actor_index 60 initial policy_version 12 policy_version after rollout 13
actor_index 77 initial policy_version 12 policy_version after rollout 13
actor_index 48 initial policy_version 12 policy_version after rollout 13

# poetry run python -m torchbeast.monobeast_study_new   --num_actors 40 --total_steps 10000000

actor_index 34 initial policy_version 12 policy_version after rollout 18
actor_index 25 initial policy_version 13 policy_version after rollout 18
```

```
actor_index 4 initial policy_version 13 policy_version after rollout 18
actor_index 5 initial policy_version 13 policy_version after rollout 18
actor_index 14 initial policy_version 13 policy_version after rollout 18
actor_index 16 initial policy_version 13 policy_version after rollout 18
actor_index 12 initial policy_version 13 policy_version after rollout 18
actor_index 39 initial policy_version 13 policy_version after rollout 18
actor_index 30 initial policy_version 13 policy_version after rollout 18
actor_index 18 initial policy_version 13 policy_version after rollout 18
actor_index 13 initial policy_version 13 policy_version after rollout 18
actor_index 23 initial policy_version 13 policy_version after rollout 19
actor_index 35 initial policy_version 13 policy_version after rollout 19
actor_index 3 initial policy_version 14 policy_version after rollout 19
actor_index 17 initial policy_version 14 policy_version after rollout 19
actor_index 9 initial policy_version 14 policy_version after rollout 19
actor_index 6 initial policy_version 14 policy_version after rollout 19
```

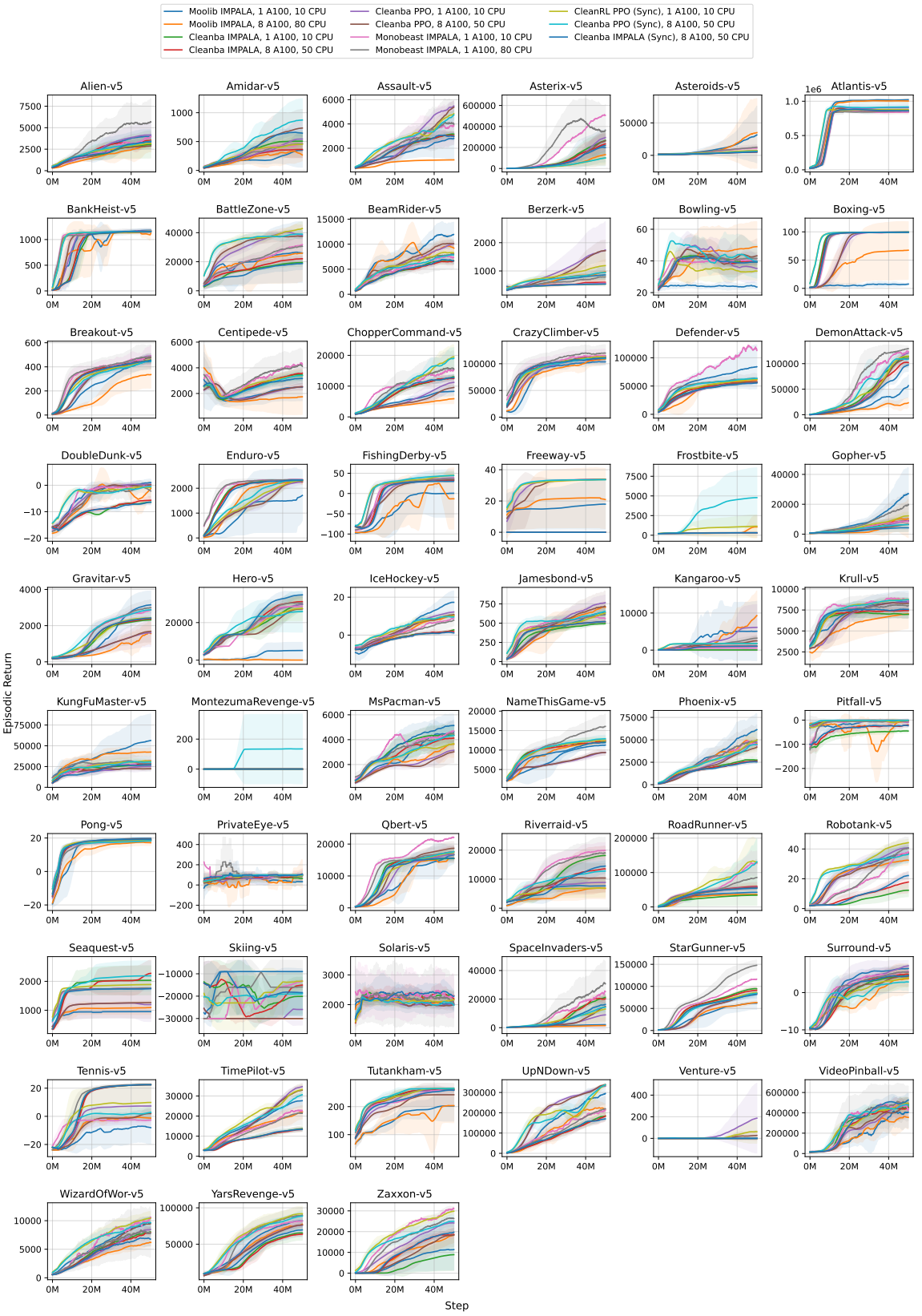


Figure C.1: The learning curves of the experiments.

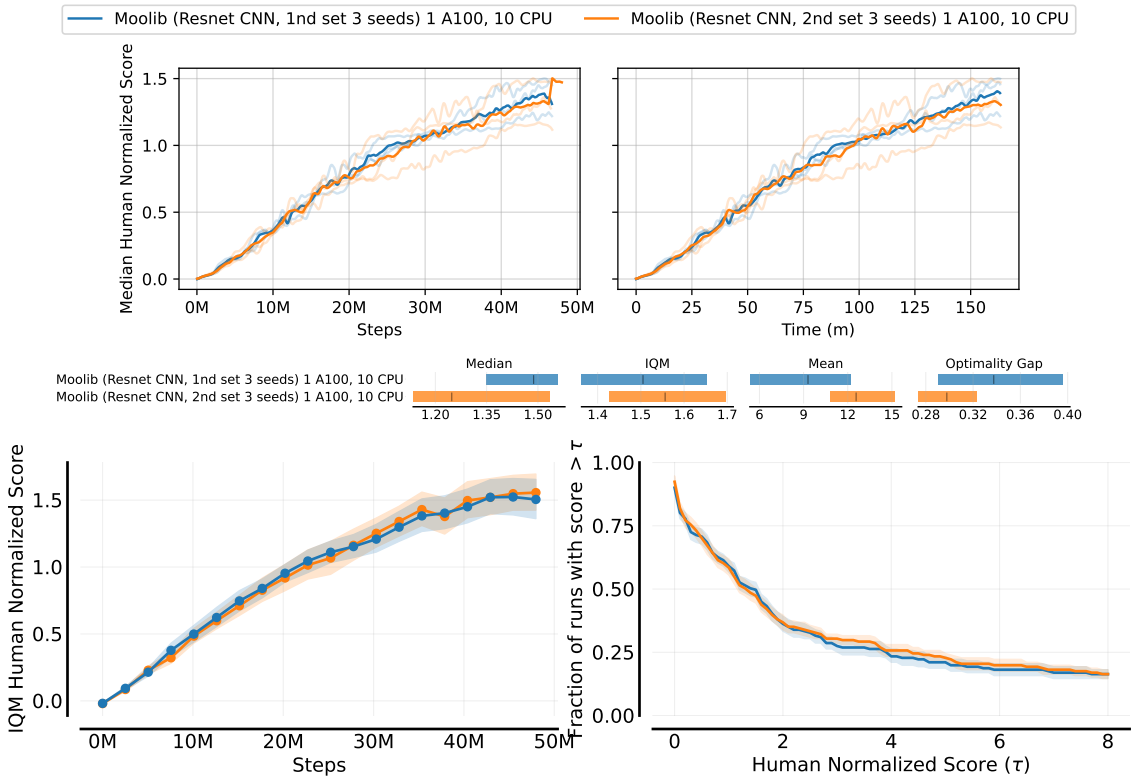


Figure C.2: Top figure: the median human-normalized scores of the two sets of `moolib` experiments. Middle figure: the IQM human-normalized scores and performance profile[1]. Bottom figure: the average runtime in minutes and aggregate human normalized score metrics with 95% stratified bootstrap CIs.

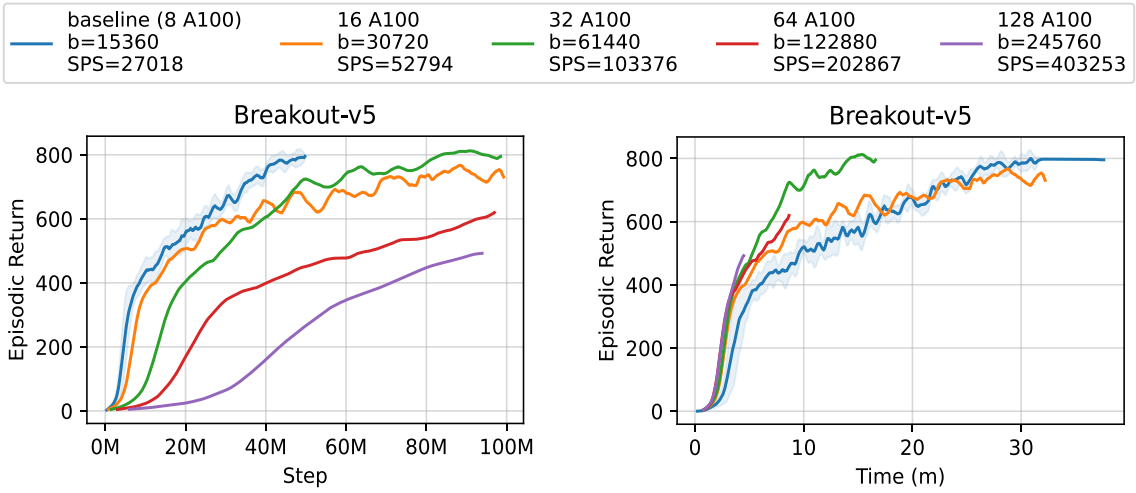


Figure C.3: Cleanba's results from large batch size training. `b=15360` denotes `batch_size=15360`.

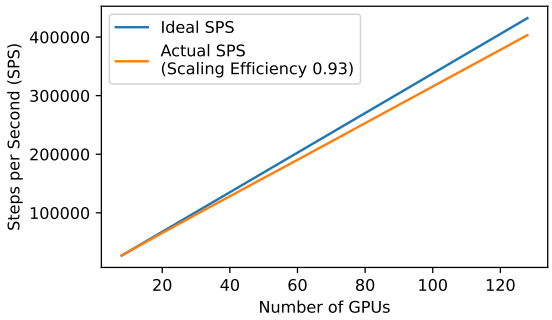


Figure C.4: Cleanba's SPS scaling results from large batch size training.

Bibliography

[1] Rishabh Agarwal, Max Schwarzer, Pablo Samuel Castro, Aaron C Courville, and Marc Bellemare. Deep reinforcement learning at the edge of the statistical precipice. *Advances in Neural Information Processing Systems*, 34, 2021.

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

[3] Xue Han and Edward S Boyden. Multiple-color optical activation, silencing, and desynchronization of neural activity, with single-spike temporal resolution. *PloS one*, 2(3):e299, 2007.

[4] Stuart J Russell. *Artificial intelligence a modern approach*. Pearson Education, Inc., 2010.

[5] Ruslan Salakhutdinov. Deep learning. In Sofus A. Macskassy, Claudia Perlich, Jure Leskovec, Wei Wang, and Rayid Ghani, editors, *The 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '14, New York, NY, USA - August 24 - 27, 2014*, page 1973. ACM, 2014. doi: 10.1145/2623330.2630809. URL https://doi.org/10.1145/2623330.2630809.

[6] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[7] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[8] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

[9] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008, 2017. URL https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html.

[10] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[11] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

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

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

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

[15] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In Maria-Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, volume 48 of *JMLR Workshop and Conference Proceedings*, pages 1928–1937. JMLR.org, 2016. URL http://proceedings.mlr.press/v48/mniha16.html.

[16] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *ArXiv preprint*, abs/1707.06347, 2017. URL https://arxiv.org/abs/1707.06347.

[17] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.

[18] Cédric Colas, Olivier Sigaud, and Pierre-Yves Oudeyer. How many random seeds? statistical power analysis in deep reinforcement learning experiments. *arXiv preprint arXiv:1806.08295*, 2018.

[19] John Schulman, Sergey Levine, Pieter Abbeel, Michael I. Jordan, and Philipp Moritz. Trust region policy optimization. In Francis R. Bach and David M. Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, volume 37 of *JMLR Workshop and Conference Proceedings*, pages 1889–1897. JMLR.org, 2015. URL http://proceedings.mlr.press/v37/schulman15.html.

[20] Logan Engstrom, Andrew Ilyas, Shibani Santurkar, Dimitris Tsipras, Firdaus Janoos, Larry Rudolph, and Aleksander Madry. Implementation matters in deep rl: A case study on ppo and trpo. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=r1etN1rtPB.

[21] Marcin Andrychowicz, Anton Raichuk, Piotr Stańczyk, Manu Orsini, Sertan Girgin, Raphaël Marinier, Leonard Hussenot, Matthieu Geist, Olivier Pietquin, Marcin Michalski, Sylvain Gelly, and Olivier Bachem. What matters for on-policy deep actor-critic methods? a large-scale study. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=nIAxjsniDzg.

[22] Shengyi Huang, Rousslan Fernand Julien Dossa, Antonin Raffin, Anssi Kanervisto, and Weixun Wang. The 37 implementation details of proximal policy optimization. In *ICLR Blog Track*, 2022. URL https://iclr-blog-track.github.io/2022/03/25/ppo-implementation-details/.

[23] Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Gheshlaghi Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. In Sheila A. McIlraith and Kilian Q. Weinberger, editors, *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, pages 3215–3222. AAAI Press, 2018. URL https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/17204.

[24] Dan Horgan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado van Hasselt, and David Silver. Distributed prioritized experience replay. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018. URL https://openreview.net/forum?id=H1Dy---0Z.

[25] Lasse Espeholt, Raphaël Marinier, Piotr Stanczyk, Ke Wang, and Marcin Michalski. Seed rl: Scalable and efficient deep-rl with accelerated central inference. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=rkgvXlrKwH.

[26] Steven Kapturowski, Georg Ostrovski, Will Dabney, John Quan, and Remi Munos. Recurrent experience replay in distributed reinforcement learning. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=r1lyTjAqYX.

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

[28] Johan S Obando-Ceron and Pablo Samuel Castro. Revisiting rainbow: Promoting more insightful and inclusive deep reinforcement learning research. In *Proceedings of the 38th International Conference on Machine Learning*, Proceedings of Machine Learning Research. PMLR, 2021.

[29] Michael Buro. Real-time strategy games: A new ai research challenge. In *IJCAI*, volume 2003, pages 1534–1535, 2003.

[30] Santiago Ontanón, Gabriel Synnaeve, Alberto Uriarte, Florian Richoux, David Churchill, and Mike Preuss. A survey of real-time strategy game ai research and competition in starcraft. *IEEE Transactions on Computational Intelligence and AI in games*, 5(4):293–311, 2013.

[31] Shengyi Huang and Santiago Ontañón. Comparing observation and action representations for deep reinforcement learning in μrts. *AIIDE Workshop on Artificial Intelligence for Strategy Games*, 2019.

[32] Xingyou Song, Yiding Jiang, Stephen Tu, Yilun Du, and Behnam Neyshabur. Observational overfitting in reinforcement learning. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. URL https://openreview.net/forum?id=HJli2hNKDH.

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

[34] Lasse Espeholt, Hubert Soyer, Rémi Munos, Karen Simonyan, Volodymyr Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, Shane Legg, and Koray Kavukcuoglu. IMPALA: scalable distributed deep-rl with importance weighted actor-learner architectures. In Jennifer G. Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine Learning Research*, pages 1406–1415. PMLR, 2018. URL http://proceedings.mlr.press/v80/espeholt18a.html.

[35] Shengyi Huang, Santiago Ontañón, Chris Bamford, and Lukasz Grela. Gym-μrts: Toward affordable full game real-time strategy games research with deep reinforcement learning. In *2021 IEEE Conference on Games (CoG)*, pages 1–8. IEEE, 2021.

[36] Shengyi Huang and Santiago Ontañón. A closer look at invalid action masking in policy gradient algorithms. volume 35, May 2022. doi: 10.32473/flairs.v35i.130584. URL https://journals.flvc.org/FLAIRS/article/view/130584.

[37] Jiayi Weng, Min Lin, Shengyi Huang, Bo Liu, Denys Makoviichuk, Viktor Makoviychuk, Zichen Liu, Yufan Song, Ting Luo, Yukun Jiang, Zhongwen Xu, and Shuicheng YAN. Envpool: A highly parallel reinforcement learning environment execution engine. In *Thirty-sixth Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2022. URL https://openreview.net/forum?id=BubxnHpuMbG.

[38] Rhys Compton, Ilya Valmianski, Li Deng, Costa Huang, Namit Katariya, Xavier Amatriain, and Anitha Kannan. Medcod: A medically-accurate, emotive, diverse, and controllable dialog system. In *Machine Learning for Health*, pages 110–129. PMLR, 2021.

[39] Chris Bamford, Shengyi Huang, and Simon Lucas. Griddly: A platform for ai research in games, 2020.

[40] Shengyi Huang and Santiago Ontañón. Action guidance: Getting the best of sparse rewards and shaped rewards for real-time strategy games. *AIIDE Workshop on Artificial Intelligence for Strategy Games*, abs/2010.03956, 2020. URL https://arxiv.org/abs/2010.03956.

[41] Shengyi Huang, Anssi Kanervisto, Antonin Raffin, Weixun Wang, Santiago Ontañón, and Rousslan Fernand Julien Dossa. A2c is a special case of ppo, 2022.

[42] Shengyi Huang and Santiago Ontañón. Measuring generalization of deep reinforcement learning with real-time strategy games. *AAAI Reinforcement Learning in Games Workshop*, 2021.

[43] Rousslan Fernand Julien Dossa, Shengyi Huang, Santiago Ontañón, and Takashi Matsubara. An empirical investigation of early stopping optimizations in proximal policy optimization. *IEEE Access*, 9:117981–117992, 2021.

[44] Shengyi Huang, Rousslan Fernand Julien Dossa, Chang Ye, Jeff Braga, Dipam Chakraborty, Kinal Mehta, and João G.M. Araújo. Cleanrl: High-quality single-file implementations of deep reinforcement learning algorithms. *Journal of Machine Learning Research*, 23(274):1–18, 2022. URL http://jmlr.org/papers/v23/21-1342.html.

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

[46] John Schulman, Philipp Moritz, Sergey Levine, Michael I. Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016. URL http://arxiv.org/abs/1506.02438.

[47] Marlos C Machado, Marc G Bellemare, Erik Talvitie, Joel Veness, Matthew Hausknecht, and Michael Bowling. Revisiting the arcade learning environment: Evaluation protocols and open problems for general agents. *Journal of Artificial Intelligence Research*, 61:523–562, 2018.

[48] John Schulman. *Optimizing Expectations: From Deep Reinforcement Learning to Stochastic Computation Graphs*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2016. URL http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-217.html.

[49] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033. IEEE, 2012.

[50] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *ArXiv preprint*, abs/1606.01540, 2016. URL https://arxiv.org/abs/1606.01540.

[51] Matthew W. Hoffman, Bobak Shahriari, John Aslanides, Gabriel Barth-Maron, Nikola Momchev, Danila Sinopalnikov, Piotr Stańczyk, Sabela Ramos, Anton Raichuk, Damien Vincent, Léonard Hussenot, Robert Dadashi, Gabriel Dulac-Arnold, Manu Orsini, Alexis Jacq, Johan Ferret, Nino Vieillard, Seyed Kamyar Seyed Ghasemipour, Sertan Girgin, Olivier Pietquin, Feryal Behbahani, Tamara Norman, Abbas Abdolmaleki, Albin Cassirer, Fan Yang, Kate Baumli, Sarah Henderson, Abe Friesen, Ruba Haroun, Alex Novikov, Sergio Gómez Colmenarejo, Serkan Cabi, Caglar Gulcehre, Tom Le Paine, Srivatsan Srinivasan, Andrew Cowie, Ziyu Wang, Bilal Piot, and Nando de Freitas. Acme: A research framework for distributed reinforcement learning, 2020.

[52] Richard Bellman. A markovian decision process. *Journal of mathematics and mechanics*, pages 679–684, 1957.

[53] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In Dale Schuurmans and Michael P. Wellman, editors, *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*, pages 2094–2100. AAAI Press, 2016. URL http://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/view/12389.

[54] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016. URL http://arxiv.org/abs/1511.05952.

[55] Marc G. Bellemare, Will Dabney, and Rémi Munos. A distributional perspective on reinforcement learning. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, volume 70 of *Proceedings of Machine Learning Research*, pages 449–458. PMLR, 2017. URL http://proceedings.mlr.press/v70/bellemare17a.html.

[56] Adrià Puigdomènech Badia, Bilal Piot, Steven Kapturowski, Pablo Sprechmann, Alex Vitvitskyi, Zhaohan Daniel Guo, and Charles Blundell. Agent57: Outperforming the Atari human benchmark. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 507–517. PMLR, 13–18 Jul 2020. URL https://proceedings.mlr.press/v119/badia20a.html.

[57] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. Dueling network architectures for deep reinforcement learning. In Maria-Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, volume 48 of *JMLR Workshop and Conference Proceedings*, pages 1995–2003. JMLR.org, 2016. URL http://proceedings.mlr.press/v48/wangf16.html.

[58] Meire Fortunato, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, Matteo Hessel, Ian Osband, Alex Graves, Volodymyr Mnih, Rémi Munos, Demis Hassabis, Olivier Pietquin, Charles Blundell, and Shane Legg. Noisy networks for exploration. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018. URL https://openreview.net/forum?id=rywHCPkAW.

[59] Joshua Achiam. Spinning Up in Deep Reinforcement Learning. 2018.

[60] Yuhuai Wu, Elman Mansimov, Roger B. Grosse, Shun Liao, and Jimmy Ba. Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5279–5288, 2017. URL https://proceedings.neurips.cc/paper/2017/hash/361440528766bbaaaa1901845cf4152b-Abstract.html.

[61] Chris Nota and Philip S. Thomas. Is the policy gradient a gradient? In *Proceedings of the 19th International Conference on Autonomous Agents and MultiAgent Systems*, AAMAS '20, page 939–947, Richland, SC, 2020. International Foundation for Autonomous Agents and Multiagent Systems. ISBN 9781450375184.

[62] Shangtong Zhang, Romain Laroche, Harm van Seijen, Shimon Whiteson, and Rémi Tachet des Combes. A deeper look at discounting mismatch in actor-critic algorithms. In *AAMAS*, 2022.

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

[64] Ziyu Wang, Victor Bapst, Nicolas Heess, Volodymyr Mnih, Rémi Munos, Koray Kavukcuoglu, and Nando de Freitas. Sample efficient actor-critic with experience replay. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. URL https://openreview.net/forum?id=HyM25Mqel.

[65] Karol Kurach, Anton Raichuk, Piotr Stanczyk, Michal Zajac, Olivier Bachem, Lasse Espeholt, Carlos Riquelme, Damien Vincent, Marcin Michalski, Olivier Bousquet, and Sylvain Gelly. Google research football: A novel reinforcement learning environment. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 4501–4510. AAAI Press, 2020. URL https://aaai.org/ojs/index.php/AAAI/article/view/5878.

[66] Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemyslaw Debiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub W. Pachocki, Michael Petrov, Henrique Pond'e de Oliveira Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang. Dota 2 with large scale deep reinforcement learning. *ArXiv preprint*, abs/1912.06680, 2019. URL https://arxiv.org/abs/1912.06680.

[67] Peng Sun, Xinghai Sun, Lei Han, Jiechao Xiong, Qing Wang, Bo Li, Yang Zheng, Ji Liu, Yongsheng Liu, Han Liu, et al. Tstarbots: Defeating the cheating level builtin ai in starcraft ii in the full game. *ArXiv preprint*, abs/1809.07193, 2018. URL https://arxiv.org/abs/1809.07193.

[68] Andrew B Kahng. Ai system outperforms humans in designing floorplans for microchips, 2021.

[69] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016. URL http://arxiv.org/abs/1509.02971.

[70] Scott Fujimoto, Herke van Hoof, and David Meger. Addressing function approximation error in actor-critic methods. In Jennifer G. Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine Learning Research*, pages 1582–1591. PMLR, 2018. URL http://proceedings.mlr.press/v80/fujimoto18a.html.

[71] Yannis Flet-Berliac. *Sample-efficient deep reinforcement learning for control, exploration and safety*. PhD thesis, Université de Lille, 2021.

[72] Viktor Makoviychuk, Lukasz Wawrzyniak, Yunrong Guo, Michelle Lu, Kier Storey, Miles Macklin, David Hoeller, Nikita Rudin, Arthur Allshire, Ankur Handa, and Gavriel State. Isaac gym: High performance GPU based physics simulation for robot learning. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*, 2021. URL https://openreview.net/forum?id=fgFBtYgJQX_.

[73] Daniel M Ziegler, Nisan Stiennon, Jeffrey Wu, Tom B Brown, Alec Radford, Dario Amodei, Paul Christiano, and Geoffrey Irving. Fine-tuning language models from human preferences. *arXiv preprint arXiv:1909.08593*, 2019.

[74] Rajkumar Ramamurthy, Prithviraj Ammanabrolu, Kianté Brantley, Jack Hessel, Rafet Sifa, Christian Bauckhage, Hannaneh Hajishirzi, and Yejin Choi. Is reinforcement learning (not) for natural language processing?: Benchmarks, baselines, and building blocks for natural language policy optimization. 2022. URL https://arxiv.org/abs/2210.01241.

[75] Jan Kossmann, Alexander Kastius, and Rainer Schlosser. Swirl: Selection of workload-aware indexes using reinforcement learning. In *EDBT*, pages 2–155, 2022.

[76] Hyoungsik Nam, Young-In Kim, Jina Bae, and Junhee Lee. Gaterl: Automated circuit design framework of cmos logic gates using reinforcement learning. *Electronics*, 10(9):1032, 2021.

[77] Azalia Mirhoseini, Anna Goldie, Mustafa Yazgan, Joe Wenjie Jiang, Ebrahim Songhori, Shen Wang, Young-Joon Lee, Eric Johnson, Omkar Pathak, Azade Nazi, et al. A graph placement methodology for fast chip design. *Nature*, 594(7862):207–212, 2021.

[78] Aleksei Petrenko, Zhehui Huang, Tushar Kumar, Gaurav S. Sukhatme, and Vladlen Koltun. Sample factory: Egocentric 3d control from pixels at 100000 FPS with asynchronous reinforcement learning. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pages 7652–7662. PMLR, 2020. URL http://proceedings.mlr.press/v119/petrenko20a.html.

[79] Karl Cobbe, Christopher Hesse, Jacob Hilton, and John Schulman. Leveraging procedural generation to benchmark reinforcement learning. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pages 2048–2056. PMLR, 2020. URL http://proceedings.mlr.press/v119/cobbe20a.html.

[80] Yuri Burda, Harrison Edwards, Amos Storkey, and Oleg Klimov. Exploration by random network distillation. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=H1lJJnR5Ym.

[81] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *ArXiv preprint*, abs/1312.5602, 2013. URL https://arxiv.org/abs/1312.5602.

[82] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021. URL http://jmlr.org/papers/v22/20-1364.html.

[83] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph Gonzalez, Michael Jordan, and Ion Stoica. Rllib: Abstractions for distributed reinforcement learning. In *International Conference on Machine Learning*, pages 3053–3062. PMLR, 2018.

[84] Carlo D'Eramo, Davide Tateo, Andrea Bonarini, Marcello Restelli, and Jan Peters. Mushroomrl: Simplifying reinforcement learning research. *Journal of Machine Learning Research*, 2020.

[85] Yasuhiro Fujita, Prabhat Nagarajan, Toshiki Kataoka, and Takahiro Ishikawa. Chainerrl: A deep reinforcement learning library. *Journal of Machine Learning Research*, 22(77):1–14, 2021.

[86] Jiayi Weng, Huayu Chen, Dong Yan, Kaichao You, Alexis Duburcq, Minghao Zhang, Yi Su, Hang Su, and Jun Zhu. Tianshou: A highly modularized deep reinforcement learning library. *Journal of Machine Learning Research*, 23(267):1–6, 2022. URL http://jmlr.org/papers/v23/21-1127.html.

[87] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*, pages 1861–1870. PMLR, 2018.

[88] Karl W Cobbe, Jacob Hilton, Oleg Klimov, and John Schulman. Phasic policy gradient. In *International Conference on Machine Learning*, pages 2020–2027. PMLR, 2021.

[89] Lukas Biewald. Experiment tracking with weights and biases, 2020. URL https://www.wandb.com/. Software available from wandb.com.

[90] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *arXiv preprint arXiv:2203.02155*, 2022.

[91] Steven Kapturowski, Georg Ostrovski, John Quan, Remi Munos, and Will Dabney. Recurrent experience replay in distributed reinforcement learning. In *International conference on learning representations*, 2019.

[92] Matteo Hessel, Manuel Kroiss, Aidan Clark, Iurii Kemaev, John Quan, Thomas Keck, Fabio Viola, and Hado van Hasselt. Podracer architectures for scalable reinforcement learning. *arXiv preprint arXiv:2104.06272*, 2021.

[93] Vegard Mella, Eric Hambro, Danielle Rothermel, and Heinrich Küttler. moolib: A Platform for Distributed RL. 2022. URL https://github.com/facebookresearch/moolib.

[94] Mohammad Babaeizadeh, Iuri Frosio, Stephen Tyree, Jason Clemons, and Jan Kautz. Reinforcement learning through asynchronous advantage actor-critic on a GPU. In *International Conference on Learning Representations*, 2017. URL https://openreview.net/forum?id=r1VGvBcxl.

[95] Adam Stooke and Pieter Abbeel. Accelerated methods for deep reinforcement learning. *arXiv preprint arXiv:1803.02811*, 2018.

[96] Erik Wijmans, Abhishek Kadian, Ari Morcos, Stefan Lee, Irfan Essa, Devi Parikh, Manolis Savva, and Dhruv Batra. Dd-ppo: Learning near-perfect pointgoal navigators from 2.5 billion frames. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=H1gX8C4YPr.

[97] Iou-Jen Liu, Raymond A. Yeh, and Alexander G. Schwing. High-throughput synchronous deep RL. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020. URL https://proceedings.neurips.cc/paper/2020/hash/c6447300d99fdbf4f3f7966295b8b5be-Abstract.html.

[98] Heinrich Küttler, Nantas Nardelli, Thibaut Lavril, Marco Selvatici, Viswanath Sivakumar, Tim Rocktäschel, and Edward Grefenstette. Torchbeast: A pytorch platform for distributed rl. *arXiv preprint arXiv:1910.03552*, 2019.

[99] C Daniel Freeman, Erik Frey, Anton Raichuk, Sertan Girgin, Igor Mordatch, and Olivier Bachem. Brax–a differentiable physics engine for large scale rigid body simulation. *arXiv preprint arXiv:2106.13281*, 2021.

[100] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, et al. Jax: composable transformations of python+ numpy programs. 2018.

[101] Ryan Sullivan, Jordan K Terry, Benjamin Black, and John P Dickerson. Cliff diving: Exploring reward surfaces in reinforcement learning environments. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu, and Sivan Sabato, editors, *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pages 20744–20776. PMLR, 17–23 Jul 2022. URL https://proceedings.mlr.press/v162/sullivan22a.html.

[102] Bhaskara Marthi, Stuart J Russell, David Latham, and Carlos Guestrin. Concurrent hierarchical reinforcement learning. In *IJCAI*, pages 779–785, 2005.

[103] U Jaidee and H Muñoz-Avila. Modeling unit classes as agents in real-time strategy games. *Proceedings of the 9th AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, AIIDE 2013*, pages 149–155, 01 2013.

[104] Martin Stolle and Doina Precup. Learning options in reinforcement learning. In *International Symposium on abstraction, reformulation, and approximation*, pages 212–223. Springer, 2002.

[105] Anderson R. Tavares and Luiz Chaimowicz. Tabular reinforcement learning in real-time strategy games via options. *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–8, 2018.

[106] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017.

[107] Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John Agapiou, Julian Schrittwieser, et al. Starcraft ii: A new challenge for reinforcement learning. *ArXiv preprint*, abs/1708.04782, 2017. URL https://arxiv.org/abs/1708.04782.

[108] Yuandong Tian, Qucheng Gong, Wenling Shang, Yuxin Wu, and C Lawrence Zitnick. Elf: An extensive, lightweight and flexible research platform for real-time strategy games. In *Advances in Neural Information Processing Systems*, pages 2659–2669, 2017.

[109] Ruo-Ze Liu, Haifeng Guo, Xiaozhong Ji, Yang Yu, Zitai Xiao, Yuzhou Wu, Zhen-Jia Pang, and Tong Lu. Efficient reinforcement learning with a mind-game for full-length starcraft ii. *arXiv preprint arXiv:1903.00715*, 2019.

[110] Dennis Lee, Haoran Tang, Jeffrey O Zhang, Huazhe Xu, Trevor Darrell, and Pieter Abbeel. Modular architecture for starcraft ii with deep reinforcement learning. In *Fourteenth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2018.

[111] Mikayel Samvelyan, Tabish Rashid, Christian Schroeder De Witt, Gregory Farquhar, Nantas Nardelli, Tim GJ Rudner, Chia-Man Hung, Philip HS Torr, Jakob Foerster, and Shimon Whiteson. The starcraft multi-agent challenge. *ArXiv preprint*, abs/1902.04043, 2019. URL https://arxiv.org/abs/1902.04043.

[112] Tabish Rashid, Mikayel Samvelyan, Christian Schroeder de Witt, Gregory Farquhar, Jakob Foerster, and Shimon Whiteson. Qmix: Monotonic value function factorisation for deep multi-agent reinforcement learning. *arXiv preprint arXiv:1803.11485*, 2018.

[113] Jakob N Foerster, Gregory Farquhar, Triantafyllos Afouras, Nantas Nardelli, and Shimon Whiteson. Counterfactual multi-agent policy gradients. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[114] Peter Sunehag, Guy Lever, Audrunas Gruslys, Wojciech Marian Czarnecki, Vinicius Zambaldi, Max Jaderberg, Marc Lanctot, Nicolas Sonnerat, Joel Z Leibo, Karl Tuyls, et al. Value-decomposition networks for cooperative multi-agent learning. *arXiv preprint arXiv:1706.05296*, 2017.

[115] Ardi Tampuu, Tambet Matiisen, Dorian Kodelja, Ilya Kuzovkin, Kristjan Korjus, Juhan Aru, Jaan Aru, and Raul Vicente. Multiagent cooperation and competition with deep reinforcement learning. *PloS one*, 12(4):e0172395, 2017.

[116] Zuozhi Yang and Santiago Ontañón. Learning map-independent evaluation functions for real-time strategy games. *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–7, 2018.

[117] Deheng Ye, Zhao Liu, Mingfei Sun, Bei Shi, Peilin Zhao, Hao Wu, Hongsheng Yu, Shaojie Yang, Xipeng Wu, Qingwei Guo, Qiaobo Chen, Yinyuting Yin, Hao Zhang, Tengfei Shi, Liang Wang, Qiang Fu, Wei Yang, and Lanxiao Huang. Mastering complex control in MOBA games with deep reinforcement learning. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 6672–6679. AAAI Press, 2020. URL https://aaai.org/ojs/index.php/AAAI/article/view/6144.

[118] Gabriel Dulac-Arnold, Richard Evans, Hado van Hasselt, Peter Sunehag, Timothy Lillicrap, Jonathan Hunt, Timothy Mann, Theophane Weber, Thomas Degris, and Ben Coppin. Deep reinforcement learning in large discrete action spaces. *ArXiv preprint*, abs/1512.07679, 2015. URL https://arxiv.org/abs/1512.07679.

[119] Tom Zahavy, Matan Haroush, Nadav Merlis, Daniel J. Mankowitz, and Shie Mannor. Learn what not to learn: Action elimination with deep reinforcement learning. In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 3566–3577, 2018. URL https://proceedings.neurips.cc/paper/2018/hash/645098b086d2f9e1e0e939c27f9f2d6f-Abstract.html.

[120] Anssi Kanervisto, Christian Scheller, and Ville Hautamäki. Action space shaping in deep reinforcement learning. *ArXiv preprint*, abs/2004.00980, 2020. URL https://arxiv.org/abs/2004.00980.

[121] Matthew Johnson, Katja Hofmann, Tim Hutton, and David Bignell. The malmo platform for artificial intelligence experimentation. In Subbarao Kambhampati, editor, *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, pages 4246–4247. IJCAI/AAAI Press, 2016. URL http://www.ijcai.org/Abstract/16/643.

[122] Marius Stanescu, Nicolas A. Barriga, Andy Hess, and Michael Buro. Evaluating real-time strategy game states using convolutional neural networks. 2016. doi: 10.1109/CIG.2016.7860439.

[123] Matthew J. Hausknecht and Peter Stone. Deep reinforcement learning in parameterized action space. In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016. URL http://arxiv.org/abs/1511.04143.

[124] Thomas G Dietterich. Hierarchical reinforcement learning with the maxq value function decomposition. *Journal of artificial intelligence research*, 13:227–303, 2000.

[125] Solomon Kullback and Richard A Leibler. On information and sufficiency. *The annals of mathematical statistics*, 22(1):79–86, 1951.

[126] Deepak Pathak, Pulkit Agrawal, Alexei A. Efros, and Trevor Darrell. Curiosity-driven exploration by self-supervised prediction. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW,*

Australia, 6-11 August 2017, volume 70 of *Proceedings of Machine Learning Research*, pages 2778–2787. PMLR, 2017. URL http://proceedings.mlr.press/v70/pathak17a.html.

[127] Andrew Y Ng, Daishi Harada, and Stuart Russell. Policy invariance under reward transformations: Theory and application to reward shaping. 1999.

[128] Matthew E. Taylor, Peter Stone, and Yaxin Liu. Transfer learning via inter-task mappings for temporal difference learning. *J. Mach. Learn. Res.*, 8:2125–2167, 2007.

[129] Maxwell Svetlik, Matteo Leonetti, Jivko Sinapov, Rishi Shah, Nick Walker, and Peter Stone. Automatic curriculum graph generation for reinforcement learning agents. In Satinder P. Singh and Shaul Markovitch, editors, *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA*, pages 2590–2596. AAAI Press, 2017. URL http://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14961.

[130] Sanmit Narvekar and Peter Stone. Learning curriculum policies for reinforcement learning. *ArXiv preprint*, abs/1812.00285, 2018. URL https://arxiv.org/abs/1812.00285.

[131] Michael Bain and Claude Sammut. A framework for behavioural cloning. In *Machine Intelligence 15*, pages 103–129, 1995.

[132] Pieter Abbeel and Andrew Y. Ng. Apprenticeship learning via inverse reinforcement learning. In Carla E. Brodley, editor, *Machine Learning, Proceedings of the Twenty-first International Conference (ICML 2004), Banff, Alberta, Canada, July 4-8, 2004*, volume 69 of *ACM International Conference Proceeding Series*. ACM, 2004. doi: 10.1145/1015330.1015430. URL https://doi.org/10.1145/1015330.1015430.

[133] Yuri Burda, Harrison Edwards, Deepak Pathak, Amos J. Storkey, Trevor Darrell, and Alexei A. Efros. Large-scale study of curiosity-driven learning. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. URL https://openreview.net/forum?id=rJNwDjAqYX.

[134] Rein Houthooft, Xi Chen, Yan Duan, John Schulman, Filip De Turck, and Pieter Abbeel. Curiosity-driven exploration in deep reinforcement learning via bayesian neural networks. 2016.

[135] Marc G. Bellemare, Sriram Srinivasan, Georg Ostrovski, Tom Schaul, David Saxton, and Rémi Munos. Unifying count-based exploration and intrinsic motivation. In Daniel D. Lee, Masashi Sugiyama, Ulrike von Luxburg, Isabelle Guyon, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pages 1471–1479, 2016. URL https://proceedings.neurips.cc/paper/2016/hash/afda332245e2af431fb7b672a68b659d-Abstract.html.

[136] Manuel Lopes, Tobias Lang, Marc Toussaint, and Pierre-Yves Oudeyer. Exploration in model-based reinforcement learning by empirically estimating learning progress. In Peter L. Bartlett, Fernando C. N. Pereira, Christopher J. C. Burges, Léon Bottou, and Kilian Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States*, pages 206–214, 2012. URL https://proceedings.neurips.cc/paper/2012/hash/a0a080f42e6f13b3a2df133f073095dd-Abstract.html.

[137] Tom Schaul, Daniel Horgan, Karol Gregor, and David Silver. Universal value function approximators. In Francis R. Bach and David M. Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, volume 37 of *JMLR Workshop and Conference Proceedings*, pages 1312–1320. JMLR.org, 2015. URL http://proceedings.mlr.press/v37/schaul15.html.

[138] Marcin Andrychowicz, Dwight Crow, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5048–5058, 2017. URL https://proceedings.neurips.cc/paper/2017/hash/453fadbd8a1a3af50a9df4df899537b5-Abstract.html.

[139] Zhen-Jia Pang, Ruo-Ze Liu, Zhou-Yu Meng, Yi Zhang, Yang Yu, and Tong Lu. On reinforcement learning for full-length game of starcraft. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*, pages 4691–4698. AAAI Press, 2019. doi: 10.1609/aaai.v33i01.33014691. URL https://doi.org/10.1609/aaai.v33i01.33014691.

[140] Deheng Ye, Zhao Liu, Mingfei Sun, Bei Shi, Peilin Zhao, Hao Wu, Hongsheng Yu, Shaojie Yang, Xipeng Wu, Qingwei Guo, Qiaobo Chen, Yinyuting Yin, Hao Zhang, Tengfei Shi, Liang Wang, Qiang Fu, Wei Yang, and Lanxiao Huang. Mastering complex control in MOBA games with deep reinforcement learning. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 6672–6679. AAAI Press, 2020. URL https://aaai.org/ojs/index.php/AAAI/article/view/6144.

[141] Martin A. Riedmiller, Roland Hafner, Thomas Lampe, Michael Neunert, Jonas Degrave, Tom Van de Wiele, Vlad Mnih, Nicolas Heess, and Jost Tobias Springenberg. Learning by playing solving sparse reward tasks from scratch. In Jennifer G. Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine Learning Research*, pages 4341–4350. PMLR, 2018. URL http://proceedings.mlr.press/v80/riedmiller18a.html.

[142] Thomas Degris, Martha White, and Richard S Sutton. Off-policy actor-critic. *ArXiv preprint*, abs/1205.4839, 2012. URL https://arxiv.org/abs/1205.4839.

[143] Sergey Levine, Aviral Kumar, George Tucker, and Justin Fu. Offline reinforcement learning: Tutorial, review, and perspectives on open problems. *ArXiv preprint*, abs/2005.01643, 2020. URL https://arxiv.org/abs/2005.01643.

[144] Lei Han, Jiechao Xiong, Peng Sun, Xinghai Sun, Meng Fang, Qingwei Guo, Qiaobo Chen, Tengfei Shi, Hongsheng Yu, and Zhengyou Zhang. Tstarbot-x: An open-sourced and comprehensive study for efficient league training in starcraft ii full game. *ArXiv preprint*, abs/2011.13729, 2020. URL https://arxiv.org/abs/2011.13729.

[145] Santiago Ontanón. The combinatorial multi-armed bandit problem and its application to real-time strategy games. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 9, 2013.

[146] Ulit Jaidee and Héctor Muñoz-Avila. Classq-l: A q-learning algorithm for adversarial real-time strategy games. In *Eighth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2012.

[147] Ben Weber and Michael Mateas. Case-based reasoning for build order in real-time strategy games. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 4, 2009.

[148] Santi Ontanón, Kinshuk Mishra, Neha Sugandh, and Ashwin Ram. On-line case-based planning. *Computational Intelligence*, 26(1):84–119, 2010.

[149] Radha-Krishna Balla and Alan Fern. Uct for tactical assault planning in real-time strategy games. In *Proceedings of the 21st international jont conference on Artifical intelligence*, pages 40–45, 2009.

[150] David Churchill, Abdallah Saffidine, and Michael Buro. Fast heuristic search for rts game combat scenarios. In *Proceedings of the AAAI conference on artificial intelligence and interactive digital entertainment*, volume 8, 2012.

[151] Niels Justesen, Bálint Tillman, Julian Togelius, and Sebastian Risi. Script-and cluster-based uct for starcraft. In *2014 IEEE Conference on Computational Intelligence and Games*, pages 1–8. IEEE, 2014.

[152] Santiago Ontanón. Combinatorial multi-armed bandits for real-time strategy games. *Journal of Artificial Intelligence Research*, 58:665–702, 2017.

[153] Per-Arne Andersen, Morten Goodwin, and Ole-Christoffer Granmo. Deep rts: a game environment for deep reinforcement learning in real-time strategy games. In *2018 IEEE conference on computational intelligence and games (CIG)*, pages 1–8. IEEE, 2018.

[154] Lei Han, Peng Sun, Yali Du, Jiechao Xiong, Qing Wang, Xinghai Sun, Han Liu, and Tong Zhang. Grid-wise control for multi-agent reinforcement learning in video game AI. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 2576–2585. PMLR, 2019. URL http://proceedings.mlr.press/v97/han19a.html.

[155] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. *Lecture Notes in Computer Science*, page 630–645, 2016. ISSN 1611-3349. doi: 10.1007/978-3-319-46493-0_38. URL http://dx.doi.org/10.1007/978-3-319-46493-0_38.

[156] Sam McCandlish, Jared Kaplan, Dario Amodei, and OpenAI Dota Team. An empirical model of large-batch training. *arXiv preprint arXiv:1812.06162*, 2018.