

The Cell Must Go On: Agar.io for Continual Reinforcement Learning

by

Mohamed Ayman Mohamed

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science
University of Alberta

© Mohamed Ayman Mohamed, 2025

This work is licensed under CC Attribution 4.0 International

Abstract

Continual reinforcement learning (RL) focuses on agents that learn and adapt incrementally over time rather than converging to a fixed policy solely for evaluation. This learning paradigm is particularly well suited to environments that agents perceive as *changing* in which static policies become suboptimal. The few simulators explicitly designed for empirical research in continual RL are often limited in scope or complexity, and it is now common for researchers to modify episodic RL environments by artificially incorporating abrupt task changes during interaction. In this thesis, I introduce **AgarCL**, a continual RL research platform inspired by the game Agar.io. **AgarCL** presents a non-episodic, high-dimensional problem featuring stochastic, ever-evolving dynamics, continuous actions, and partial observability. By enabling a natural progression of increasingly sophisticated and adaptive behaviors, **AgarCL** offers a challenging testbed for evaluating and advancing continual RL research. Additionally, I provide benchmark results reporting the performance of DQN, PPO, and SAC in both the primary, challenging continual RL problem and across a suite of smaller tasks within **AgarCL**, each designed to isolate and characterize the different challenges posed by different facets of the environment. These evaluations demonstrate the difficulty of the domain and highlight important directions for future continual RL research.

Preface

The work presented in this thesis is based on the preprint *The Cell Must Go On: Agar.io for Continual Reinforcement Learning* (Mohamed et al., 2025), which is currently under review. I, Mohamed A. Mohamed, am the first author and carried out most of the work. I developed the complete game environment by extending the AgarLE GitHub repository (Deaton, 2018), refactoring and optimizing the codebase to better support research in continual reinforcement learning. I also designed the experiments and performed hyperparameter tuning for multiple algorithms. Kateryna Nekhomiazh, the second author, contributed by implementing screen-based observations and identifying performance bottlenecks in the C++-Python interface. Vedant Vyas added support for GoBigger-style observations (Zhang et al., 2023a), while Marcos M. Jose assisted in running experiments. Andrew Patterson helped refactor and optimize the code, and provided valuable feedback on system design, reward functions, and observation representations. Finally, Marlos C. Machado supervised the project, guiding key decisions on observation modalities, experimental design, and overall research direction.

To Shorouk, my parents, and my siblings for walking beside me throughout this journey, with unwavering support and countless sacrifices.

“Journey before destination. A journey will have pain and failure. It is not only the steps forward that we must accept. It is the stumbles. The trials. The knowledge that we will fail. That we will hurt those around us. But if we stop, if we accept the person we are when we fall, the journey ends. That failure becomes our destination. To love the journey is to accept no such end. I have found, through painful experience, that the most important step a person can take is always the next one.”

—Brandon Sanderson, Oathbringer.

Acknowledgements

Throughout my Master’s journey, I was fortunate to have many people by my side. Now, I have the opportunity to express my heartfelt gratitude and acknowledge their support.

Marlos C. Machado has been the supervisor I always hoped for, and I am deeply grateful that he accepted me as his student. He not only taught me how to do proper science, but also how to think critically and grow as a researcher. Just as importantly, he created a space where I could speak openly and honestly—supporting me not only academically, but also as a mentor and a friend.

I am also thankful to my colleagues, Alex Lewandowski and Anna Hakhverdyan, for their thoughtful advice and insights throughout this project. My sincere thanks go to my collaborators—Kateryna Nekhomiazh, Vedant Vyas, Marcos M. José, and Andrew Patterson—who contributed greatly to shaping the experiments and deepening my understanding of Agar.io, helping me to show how interesting this research platform is.

Table of Contents

1	Introduction	1
1.1	Thesis Statement	5
1.2	Contribution	5
2	Background	7
2.1	Reinforcement Learning Formulation	7
2.1.1	Continual Reinforcement Learning Formulation	10
2.1.2	Markov Decision Processes and Observability	11
2.2	Model-Free Reinforcement Learning	12
2.2.1	Value-Based Methods	12
2.2.2	Policy-Gradient Methods	15
2.3	Related Work	18
2.3.1	Existing Environments	19
2.3.2	Solution Methods for Continual RL	19
2.3.3	Agar-Like Environments.	21
3	AgarCL: Agar.io for Continual Reinforcement Learning	23
3.1	The Game: Agar.io	23
3.2	Agent-Environment Interface	27
3.2.1	Technical Details	29
4	Evaluating AgarCL as a Platform for Continual Reinforcement Learning	31
4.1	Pellets Collection Tasks	32
4.1.1	Non-Stationarity through Mass Variation	33
4.1.2	Continuing and Continual Problems.	35
4.2	Interacting with Other Agents	38
4.2.1	Large-Dense Setup	39
4.2.2	Small-Sparse Setup	40

4.3	Complex Dynamics and Long-Term Consequences	42
4.4	Hyperparameter Tuning in Continual Reinforcement Learning	43
4.5	How is AgarCL an Effective Continual Reinforcement Learning Testbed?	45
4.5.1	The FULL-GAME Evaluation	45
4.5.2	Positioning AgarCL as a Next-Generation Continual RL Testbed	47
5	Conclusion and Future Work	51
	Bibliography	52
	Appendix A: Computational Resources	56
	Appendix B: Tuning Details	57
B.1	DQN Tuning	57
B.2	PPO Tuning	59
B.3	SAC Tuning	62
	Appendix C: Empirical Results on Different Tasks	64

List of Tables

4.1	PPO + RNN performance across 6 mini-games	38
4.2	PPO performance across difficulty levels	48
4.3	Comparison at key checkpoints and PPO learning agent	50
A.1	Compute resource utilization across four clusters.	56
B.1	Values of hyperparameters that I swept over when tuning DQN. . . .	58
B.2	Best hyperparameters for DQN on each minigame.	58
B.3	Values of hyperparameters that I swept over when tuning PPO. . . .	60
B.4	Updated PPO hyperparameters used for tuning.	61
B.5	Best hyperparameters for PPO on each minigame.	61
B.6	Values of hyperparameters that I swept over when tuning SAC. . . .	62
B.7	Best hyperparameters for SAC on each minigame.	63
C.1	Performance across AgarCL minigames averaged over 10.	64

List of Figures

1.1	Agent-environment interface and main entities in AgarCL	3
3.1	Environment dynamics and actions in AgarCL , with the agent.	24
3.2	First 5,000 time steps of an expert trajectory showing game.	26
3.3	Agent-environment interface and main entities in AgarCL	28
4.1	Architecture of the encoder module α used by each agent.	32
4.2	Pellet Collection mini games. Six mini-games were described.	33
4.3	Performance of RL baselines on <i>episodic</i> pellet-collection.	34
4.4	Performance of RL baselines on <i>continual</i> pellet-collection.	36
4.5	PPO performance on <i>continual randomly regenerated pellet-collection</i>	36
4.6	Performance of PPO with RNN backbone on <i>continual</i> pellet-collection.	38
4.7	Performance of RL baselines—DQN, PPO, and SAC—on <i>episodic</i> other-agent mini-games.	40
4.8	Performance of RL baselines—DQN, PPO, and SAC—on <i>episodic</i> other-agent mini-games.	41
4.9	Virus mini-game	42
4.10	Performance of RL baselines—DQN, PPO, and SAC—on Virus mini-game.	42
4.11	Performance of baseline agents (DQN, PPO, SAC).	44
4.12	Performance of established deep RL algorithms in the full.	46
4.13	PPO performance across varying pellet densities.	48
4.14	Continual Learning vs Fixed Policy.	49

Chapter 1

Introduction

Continual reinforcement learning (RL) refers to an RL setting in which an agent is expected to acquire and adapt its behavior incrementally rather than converging to a fixed policy for evaluation or deployment. Unlike traditional RL settings, continual RL frames learning as an ongoing process: the agent must continually update its knowledge and policy in response to a perceived non-stationarity. In practice, this means designing agents that can handle shifts in rewards, transition dynamics, and tasks even when the boundaries between them are not clearly defined. This formulation can be viewed either as an RL problem (Khetarpal et al., 2022; Abel et al., 2023; Kumar et al., 2025) or as a solution method for problems in which an agent interacts with environments whose transition dynamics and reward function seem to change over time. A key motivation behind continual RL is the “big world hypothesis”, which suggests that the environment is much bigger than the agent (Javed and Sutton, 2024), with continual adaptation emerging as a more effective and robust alternative to fixed-policy approaches (Sutton et al., 2007; Janjua et al., 2024).

Much of the progress in reinforcement learning (RL) has been driven by empirical advances, where experimental results have often inspired new algorithms and guided research direction. As a result, carefully designed evaluation platforms have played a pivotal role in shaping the field. These range from physics-based simulators, like MuJoCo (Todorov et al., 2012), to classic video game environments, like the Arcade

Learning Environment (Bellemare et al., 2013), and more complex 3D settings such as DeepMind Lab (Beattie et al., 2016). In the realm of continual RL, however, most benchmarks are adapted from traditional RL environments. To simulate a world that is larger than the agent, requiring continual adaptation, researchers artificially introduce non-stationarity through mechanisms like periodically switching the episodic task faced by the agent (Powers et al., 2022; Abbas et al., 2023; Anand and Precup, 2023; Tomilin et al., 2023). A smaller number of platforms have been designed specifically for continual RL, including some that avoid episodic assumptions entirely (Platanios et al., 2020).

However, both strategies have significant limitations. Repurposing established benchmarks—even sophisticated simulators—by inserting arbitrary task switches often yields contrived evaluations: the non-stationarity bears little relation to the agent’s actions or the dynamics of the underlying environment. It is as if one suddenly asked someone skilled in soccer to excel in scuba diving without any prior warning or training—an abrupt transition that rarely occurs in real-world settings.

On the other hand, purpose-built continual reinforcement learning (RL) environments are promising in principle, but existing implementations often remain constrained by overly simplistic dynamics or narrowly defined tasks, like JellyBeanWorld (Platanios et al., 2020).

My thesis takes a complementary direction, aiming to preserve the continual, non-episodic nature of these purpose-built environments while introducing richer and more dynamic elements, such as stochasticity, continuous action spaces, partial observability, resource-driven competition, and a scaffolded progression of behavioral complexity within a spatially structured world.

Specifically, in this thesis, I introduce **AgarCL**, a continual RL evaluation platform inspired by the browser game Agar.io (see Figure 1.1). Designed explicitly to support research in continual RL, **AgarCL** places the agent in control of one or more circular cells within a bounded, Petri dish-like arena populated by food pellets, viruses, and

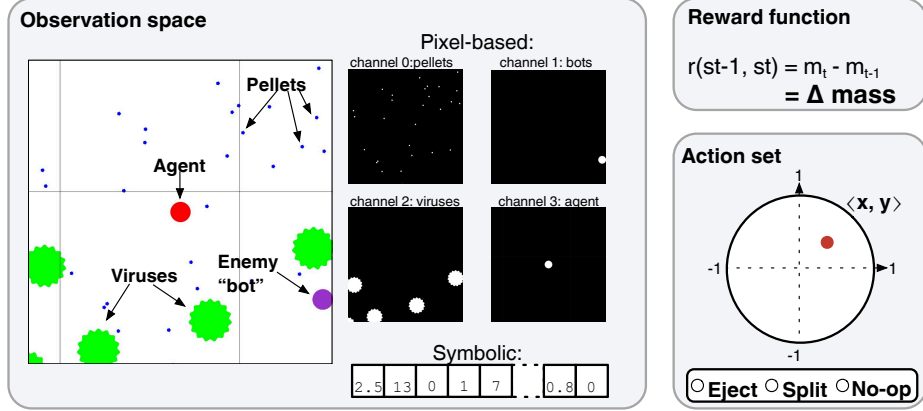


Figure 1.1: Agent-environment interface and main entities in **AgarCL**. The agent has access to one of two observation types: pixel-based or symbolic. The pixel-based observation includes four channels that represent different game entities: pellets, bots, viruses, and the agent itself. Alternatively, the agent can receive a symbolic observation, consisting of pre-processed features such as the distances to nearby enemies, pellets, and other entities. The reward function is defined as the change in the agent’s mass between two consecutive time steps. The action space in **AgarCL** is hybrid: at each time step, the agent selects an $\langle x, y \rangle$ coordinate mimicking where a human player would point their mouse. Additionally, the agent decides whether to split, eject mass, or move. In Section 3, I provide detailed descriptions of these entities and the game dynamics, including the available actions.

other agents. The agent perceives the environment through high-dimensional, four-channel pixel observations that separately represent the locations of pellets, other bots, viruses, and the agent itself—along with spatial gridlines to assist spatial reasoning.¹. Action selection occurs through a hybrid control interface: at each timestep, the agent selects a continuous movement vector, $\langle x, y \rangle$, simulating mouse-like navigation, and simultaneously chooses whether to execute a discrete action—splitting, ejecting mass, or doing nothing. The reward signal is defined as the change in the agent’s mass from one timestep to the next. Thus, an effective agent has a forever-growing mass.

What makes this environment interesting from a continual learning perspective is its game dynamics. The learning agent lives in an environment populated by other agents. It can increase its mass by collecting stationary food pellets or absorbing

¹An alternative symbolic observation mode is also supported, offering a compact, feature-based representation for backward compatibility with other platforms (Zhang et al., 2023a) See Section 3 for more details.

smaller agents. As a result, the agent’s objective is to grow by consuming smaller bots, pellets, and viruses² and to survive by avoiding larger, more threatening agents. At each timestep, the mass of a cell naturally decays at a constant rate, 2% of decay proportional to its mass. Since all agents share the same density, greater mass implies a larger physical size, making heavily weighted agents slower and more vulnerable to dynamic threats. Beyond its never-ending nature and the behavioral complexity introduced by other agents, a key property of **AgarCL** that is interesting from the continual RL perspective is how the agent’s dynamics evolve with its mass. As the agent grows, it becomes slower, and its observation stream is smoothly zoomed out to keep its whole body visible. This causes both the effect of its actions and its perception of the environment to change continually. In addition to these gradual shifts, abrupt transitions can also occur—such as when the agent absorbs another agent and its mass spikes suddenly, when it voluntarily splits into multiple cells, or when it is forced to split upon colliding with a virus. Figure 3.1 and a YouTube video³ depict some of the environment dynamics supported by **AgarCL**. By contrast, earlier Agar-style testbeds (e.g., Wiehe et al., 2018; Ansó et al., 2019) focused primarily on pellet-collection behaviors, without explicitly framing the environment itself as a research artifact.

I demonstrate the feasibility of **AgarCL** as both an evaluation platform and a challenge problem for continual RL by benchmarking three widely used algorithms—DQN (Mnih et al., 2015), PPO (Schulman et al., 2017), and SAC (Haarnoja et al., 2018)—across multiple variations of the environment. In the core game, none of these algorithms succeed in learning an effective policy, highlighting the task’s difficulty. To better understand these failures, I also assess their performance on a set of **AgarCL** mini-games, each designed to isolate specific sources of complexity such as non-stationarity, exploration, and credit assignment. These targeted evaluations

²The agent can consume the virus when the agent has at least 14 cells. Otherwise, the agent will split.

³<https://www.youtube.com/watch?v=CGpvzHIqFLA>

provide indirect insight into the challenges posed by the full game. Additionally, I present empirical findings that highlight persistent challenges in continual RL research, particularly in evaluation methodology and sensitivity to hyperparameters.

1.1 Thesis Statement

The central hypothesis of this thesis is that **AgarCL presents a challenging benchmark for continual RL**, one that exposes the limitations of existing deep RL algorithms and offers a testbed for continually developing adaptive agents. Unlike prior platforms, **AgarCL** integrates hybrid action control, evolving dynamics, partial observability, and unbounded interaction time into a single, non-episodic environment. The latter three evolving dynamics, partial observability, and unbounded interaction are key drivers of continual learning challenges, as they embody the “big world” assumption: the world’s complexity exceeds the agent’s representational capacity (Sutton et al., 2007; Javed and Sutton, 2024), requiring constant adaptation, as shown empirically in Chapter 4.

1.2 Contribution

The main contributions of this thesis are as follows:

- I introduce the **AgarCL** testbed (Chapter 3), which challenges agents to grow and adapt indefinitely, and exposes persistent shortcomings in standard RL algorithms that struggled to learn effectively (Chapter 4).
- I design and implement several mini-games within the **AgarCL** framework that isolate distinct features of the environment. These mini-games selectively mitigate specific dimensions, such as non-stationarity and exploration, allowing us to examine how individual factors impact learning. They serve as controlled sub-problems with tunable difficulty. In contrast, the full **AgarCL** game combines all challenges simultaneously. The inclusion of these mini-games within a unified

evaluation platform is a contribution in itself, enabling ablation studies that were not feasible in prior Agar.io-like environments.

- I analyze the behavior of three widely used deep RL algorithms—Deep Q-Network (DQN; Mnih et al., 2015), Proximal Policy Optimization (PPO; Schulman et al., 2017), and Soft Actor-Critic (SAC; Haarnoja et al., 2018)—in the **AgarCL** environment. This extensive evaluation provides insights into how these methods adapt to non-stationary, non-episodic settings. Results show that none of the algorithms develop an effective mass-growth strategy in the full environment within a reasonable training time, while simpler mini-games reveal partial skill acquisition but persistent difficulties under strong non-stationarity or exploration challenges.
- Through my evaluation, I draw broader lessons for continual RL research. I show that standard evaluation metrics (random seeds, replay buffers, episodic resets) can dramatically affect performance, emphasizing the need for rigorous methodology in continual settings. I also highlight that performance, in **AgarCL**, is very susceptible to choices of hyperparameters and network architecture.

Chapter 2

Background

In this chapter, I present the notation and background necessary for understanding the subsequent chapters of this thesis. I begin by reviewing the standard RL formalism, followed by a discussion of action-value methods, using Deep Q-Networks (DQN; Mnih et al., 2015) as an example, and policy gradient methods, focusing on Proximal Policy Optimization (PPO; Schulman et al., 2017) and Soft Actor-Critic (SAC; Haarnoja et al., 2018). I conclude the chapter by surveying various definitions of Continual RL found in the literature, examining existing benchmark environments, and discussing Agar-like environments most closely related to my work. I also highlight the key differences distinguishing my approach in this thesis from prior work.

In this thesis, I use the following notation conventions: scalar-valued random variables are represented by uppercase letters (e.g., R_t); vectors are indicated by bold lowercase letters (e.g., $\boldsymbol{\theta}$, $\boldsymbol{\phi}$); functions are denoted by standard lowercase letters (e.g., v , q); and sets are represented using calligraphic typeface (e.g., $\mathcal{S}$, $\mathcal{A}$).

2.1 Reinforcement Learning Formulation

Reinforcement Learning (RL) is a problem formalism that addresses sequential decision-making problems. In such problems, an agent interacts with an unknown environment over a series of time steps to maximize some cumulative reward it receives from its actions. This interaction is typically formalized as a *Markov Decision Process* (MDPs; Puterman, 2014), defined by the tuple $(\mathcal{S}, \mathcal{A}, p, r, \mu)$, where $\mathcal{S}$ denotes the set of

environment states, $\mathcal{A}$ is the set of actions available to the agent, $p : \mathcal{S} \times \mathcal{A} \rightarrow \Delta(\mathcal{S})$ is the transition function specifying a probability distribution over next states given a current state-action pair, and $r : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ is the reward function assigning a real-valued signal based on the current state and action. μ is a start-state distribution over $\mathcal{S}$.

At time $t = 0$, the agent begins in a state $S_0 \sim \mu$. However, the agent can only access an observation, $\omega_0 \in \Omega$, generated by an observation function, $\Phi : \mathcal{S} \rightarrow \Delta(\Omega)$. At each time step t , the agent selects an action $A_t \in \mathcal{A}$, receives a reward $R_{t+1} \in \mathbb{R}$, and observes $\omega_{t+1} \sim \Phi(S_{t+1})$, where $S_{t+1} \sim p(\cdot \mid S_t, A_t)$ is the next environment state. The agent chooses actions according to a policy π , which maps histories or past observations—possibly in a stochastic manner—to actions. The agent’s objective is to maximize the cumulative reward, commonly called the *return* G_t . There are multiple ways to define return, depending on whether the interaction is episodic or continuing.

Episodic Case. In the episodic setting, the return is typically defined as the sum of rewards obtained from time step t until the final time step T :

$$G_t \doteq R_{t+1} + R_{t+2} + \cdots + R_T. \quad (2.1)$$

Episodes have a natural finite horizon, terminating either when the agent reaches a goal or when a maximum time limit is exceeded. Performance in episodic tasks is typically measured by the total reward accumulated within each episode. Classic benchmarks, such as Atari 2600 games (Bellemare et al., 2013), feature clear episode boundaries, and the agent’s objective is to maximize the sum of rewards obtained during an episode.

Continuing Case. In many settings, the agent–environment interaction does not naturally break into episodes and instead continues indefinitely. These are called continuing tasks. The standard return formulation (2.1) becomes problematic here, as the final time step T is effectively infinite, making the return potentially unbounded.

Thus, *discounted return* is introduced, denoted G_t^γ , and defined as:

$$G_t^\gamma \doteq \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}, \quad (2.2)$$

where $\gamma \in [0, 1)$ is the discount factor. This formulation emphasizes immediate rewards, often appropriate in tasks where delayed rewards are less certain or less valued.

Alternatively, one may consider the *average-reward* setting. The average-reward formulation is adopted for continuing tasks. Unlike the discounted setting, the agent in the average-reward formulation values future rewards equally to immediate rewards without applying any temporal discounting. In this case, the return is defined relative to the long-run average reward of the policy:

$$G_t^{\bar{r}} \doteq \sum_{k=0}^{\infty} (R_{t+k+1} - r(\pi)), \quad (2.3)$$

with the long-term average reward given by

$$r(\pi) \doteq \lim_{h \rightarrow \infty} \frac{1}{h} \sum_{t=1}^h \mathbb{E}[R_t \mid \omega_0, A_{0:t-1} \sim \pi], \quad (2.4)$$

where ω_0 denotes the initial observation. The choice between discounted and average-reward formulations depends on the nature of the task.

In this work, I use the undiscounted return as the performance metric. Still, I evaluate algorithms designed for the discounted formulation. Because I focus on **AgarCL** as a research environment, not on solution methods, I decided to leverage the much more mature literature on deep RL in the discounted case. Note, this is not uncommon. In Atari 2600 games, for example, agents are evaluated on the score they accumulate while they are trained to maximize the discounted return (Bellemare et al., 2013; Machado et al., 2018). Additionally, for every problem, a critical discount factor exists such that any solution using a higher value will maximize the average reward (Blackwell, 1962).

2.1.1 Continual Reinforcement Learning Formulation

Standard RL assumes a stationary MDP: an agent interacts with an environment with fixed underlying dynamics, until the agent finds an optimal policy and then stops learning. Continual RL, by contrast, treats learning as endless adaptation (Abel et al., 2023). In this setting, the best agents “never stop learning”. Thus, the environment is expected to evolve over time, including its transition dynamics and reward function. This shift in perspective has motivated a growing body of work that aims to formalize the continual RL problem in various ways (e.g., Khetarpal et al., 2022; Abel et al., 2023; Kumar et al., 2025), reflecting different emphases but grounded in a common challenge: building agents that can keep learning under change, uncertainty, and resource constraints.

To unpack how these evolving problem characteristics have been formalized, I now review four perspectives that each capture a different dimension of continual RL. Khetarpal et al. (2022) proposed a taxonomy that organizes continual RL problems along two axes: scope, denoting the degree of variation across tasks or domains, and driver, identifying the source of change, such as shifts in environment or agent behavior. Rather than offering a formal definition, this taxonomy provides a unifying lens for situating different types of non-stationarity. Abel et al. (2023) defined continual RL as the ongoing process of policy improvement in environments with evolving objectives, dynamics, or rewards—framing it as a regime where “the best agents never stop learning.” Kumar et al. (2025) introduced a resource-bounded formalism in which agents must maximize long-term reward under strict per-time-step constraints on computation and memory, emphasizing the tension between efficient online learning and knowledge retention. Finally, Javed and Sutton (2024) argued that real-world environments are inherently too complex for any agent to fully observe or memorize, a perspective formalized as the Big World hypothesis. Thus, it necessitates continual approximation and adaptation using limited capacity. While these perspectives inspire

my work, I adopt a *problem-driven* view: designing environments that an agent can incrementally learn without reaching an optimal solution.

2.1.2 Markov Decision Processes and Observability

A Markov Decision Process (MDP) provides the standard mathematical framework for modeling sequential decision-making in fully observable environments. In an MDP, the environment evolves according to the transition dynamics $p(s' \mid s, a)$, which defines the probability of transitioning to state $s' \in \mathcal{S}$ after taking action $a \in \mathcal{A}$ in state $s \in \mathcal{S}$. MDP assumes full observability in which the agent has direct access to the actual environment state at each time step. Therefore, the observation function can be considered the identity mapping, and the agent’s decision-making is based solely on the current state. This is formalized by the *Markov property*, which indicates that the probability distribution over the next state depends only on the current state and the action taken by the agent, rather than the complete history of prior interactions. However, as I discuss in the next chapter, AgarCL is a partially observable environment in which the underlying dynamics of the environment are not fully accessible to the agent. As a result, it is more appropriately modeled using the Partially Observable Markov Decision Process formalism (POMDP; Kaelbling et al., 1998), which accounts for hidden state information and uncertainty in observations. In such settings, the environment is too complex for the agent to fully capture, whether due to non-stationarity in the transition dynamics or the inherent limitations of partial observability. Therefore, we can remain agnostic to specific definitions of continual RL. As discussed in the previous chapter, my underlying assumption is that continual RL approaches are interesting in this setting because they will be more effective solution methods than any fixed policy (see Sutton et al., 2007; Janjua et al., 2024), allowing me to avoid placing restrictions on agents to ensure the environment is “bigger than the agent” (Javed and Sutton, 2024).

2.2 Model-Free Reinforcement Learning

In the RL formalism, the agent does not have prior knowledge of the transition dynamics, p , or the reward function, r . *Model-free* RL methods address this by learning directly from interactions with the environment, typically using sampled transitions (s_t, a_t, r_t, s_{t+1}) , without explicitly modeling the environment. Model-free algorithms directly estimate value functions v_π , q_π , or policies π . These methods are particularly effective in high-dimensional observations from complex environments where modeling dynamics is difficult. Their compatibility with deep neural networks has enabled strong empirical performance across various domains. Despite requiring more data due to their sampling-based nature, model-free methods have achieved state-of-the-art results in benchmarks such as the Arcade Learning Environment and MuJoCo environments (Bellemare et al., 2013; Todorov et al., 2012).

Model-free reinforcement learning (RL) approaches are generally categorized into value-based and policy-gradient methods, both of which are discussed in the rest of the chapter.

2.2.1 Value-Based Methods

Action-value methods are a fundamental class of value-based reinforcement learning algorithms that estimate the expected return, $\mathbb{E}_\pi[G_t]$, for taking a specific action in a given state. These estimates, known as Q -values¹, guide decision-making by allowing agents to choose actions that maximize expected returns. Since they rely only on these value estimates, action-value methods can learn optimal policies without needing a model of the environment.

¹The action-value function (or Q -function) under policy π is defined as

$$Q^\pi(s, a) \doteq \mathbb{E}_\pi \left[G_t^\gamma \mid S_t = s, A_t = a \right],$$

where $\gamma \in [0, 1)$ is the discount factor and the expectation is taken over both the environment's stochastic transitions and actions chosen by π .

A classic example of an action-value method is Q-learning (Watkins and Dayan, 1992), an off-policy RL algorithm in which the agent learns the value of the *optimal* policy, irrespective of the policy being followed during interaction with the environment, which we call the *behavior* policy. This decoupling allows the agent to learn from past or exploratory behavior while converging toward the optimal action-value function (Sutton and Barto, 1998). The Q-learning update rule is

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_{a \in \mathcal{A}} Q(S_{t+1}, a) - Q(S_t, A_t) \right], \quad (2.5)$$

where α is the algorithm's step size.

The policy learned by Q-learning is the *greedy policy* with respect to the estimated action-value function $Q(s, a)$, defined as

$$\pi(s) \doteq \arg \max_{a \in \mathcal{A}} Q(s, a),$$

where the action selected in state s is the one that maximizes the current estimate of the expected return.

When every value of the state-action pair needs to be stored, the approach is referred to as a *tabular* method. However, this becomes impractical in environments with many states or state-action pairs due to memory limitations and the inefficiency of learning individual values. In such cases, function *approximation* is employed to generalize across states and actions, allowing the agent to learn effectively without explicitly storing every possible state-action pair. We approximate either the state value function, $V(s)$ ², or the action-value function, $Q(s, a)$, using a parameterized

²The state-value function under policy π is defined as

$$V^\pi(s) = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s \right],$$

where $\gamma \in [0, 1)$ is the discount factor and the expectation is taken over the environment's stochastic transitions and actions chosen by π . The optimal state-value function,

$$V^*(s) = \max_{\pi} V^\pi(s),$$

satisfies the Bellman optimality equation:

$$V^*(s) = \max_a \mathbb{E} \left[R_{t+1} + \gamma V^*(S_{t+1}) \mid S_t = s, A_t = a \right].$$

function with a set of parameters θ , denoted as $V(s; \theta) \approx V(s)$ or $Q(s, a; \theta) \approx Q(s, a)$.

Mnih et al. (2015) successfully leveraged deep neural networks to approximate the action-value function, introducing the Deep Q-Network (DQN; Mnih et al., 2015) in which agents can learn directly from high-dimensional sensory inputs, such as raw pixels.

Deep Q-Network

Deep Q-Network (DQN) is a value-based, off-policy RL algorithm that approximates the optimal action-value function, $Q^*(s, a)$, using a deep neural network (Mnih et al., 2015). It extends traditional Q-learning to high-dimensional state spaces by integrating several key techniques that enhance stability and sample efficiency. A central component of DQN is the experience replay buffer, which stores transitions (s, a, r, s') encountered during interaction with the environment. By uniformly sampling mini-batches from this buffer, the agent breaks temporal correlations in the data and reuses past experiences more effectively.

To further stabilize training, DQN employs a separate target network, Q_{θ^-} , a lagged copy of the online Q-network, Q_{θ} , which is periodically updated. This target network is used to compute the target values in the temporal-difference (TD) loss function:

$$\mathcal{L}(\theta) = \mathbb{E}_{(s,a,r,s') \sim \mathcal{D}} \left[\left(r + \gamma \max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta) \right)^2 \right], \quad (2.6)$$

where $\gamma \in [0, 1)$ is the discount factor and $\mathcal{D}$ denotes the distribution over transitions sampled from the replay buffer. This combination of off-policy learning, experience replay, and the use of a target network allows DQN to mitigate instability and divergence, enabling robust learning in complex environments.

My Implementation. In my experiments, I used the DQN implementation provided by PFRL (Fujita et al., 2021). To handle the continuous action space of my environment, I discretized it into 24 discrete actions, formed by pairing 8 directional vectors—UP (0, 1), UP-RIGHT (1, 1), RIGHT (1, 0), DOWN-RIGHT (1, -1), DOWN (0, -1),

DOWN-LEFT $(-1, -1)$, LEFT $(-1, 0)$, and UP-LEFT $(-1, 1)$ —with 3 discrete action types.

Resets in episodic settings help agents recover from suboptimal states by returning them to the start-state distribution, reducing failed exploration costs. In such settings, the standard ϵ -greedy exploration strategy can be more effective than in non-episodic settings, as natural episode boundaries periodically refresh the agent’s exploration opportunities. However, in my continual tasks, I observed that ϵ -greedy was ineffective due to its lack of temporal persistence,³ often failing to escape poorly performing regions. To address this, I adopted ϵz -greedy exploration (Dabney et al., 2021), which extends random actions over multiple steps with durations sampled from a heavy-tailed distribution. This temporally-extended exploration mechanism improves the agent’s ability to explore the environment in a structured and persistent manner. For episodic environments, we retained the standard ϵ -greedy approach, as it was adequate given the natural reset structure of the tasks.

2.2.2 Policy-Gradient Methods

The other primary class of model-free RL methods is policy-gradient methods. Unlike value-based approaches, where the policy is derived from estimated action values, policy-gradient methods directly learn a parameterized policy that selects actions by maximizing an objective function (Sutton and Barto, 1998). Although a value function may still guide policy updates—such as in actor-critic architectures—it is not required for action selection. Policy-gradient methods offer several advantages. They are especially well-suited to the **AgarCL** environment, where the agent must learn over continuous action spaces, along with a discrete action space, making direct policy optimization a natural and practical choice.

In this work, I employ Proximal Policy Optimization (PPO; Schulman et al., 2017) and Soft Actor-Critic (SAC; Haarnoja et al., 2018)—two widely used and state-of-the-

³It does not mean that ϵz -greedy would not help in episodic tasks.

art policy-gradient algorithms known for their stability and performance in continuous control tasks. I adapt both algorithms to handle hybrid action spaces, enabling them to learn in `AgarCL` effectively.

Proximal Policy Optimization

Proximal Policy Optimization (PPO; Schulman et al., 2017) is a widely adopted on-policy gradient algorithm that offers a stable and efficient alternative to Trust Region Policy Optimization (TRPO) (Schulman et al., 2015). PPO updates policies through a surrogate objective that restricts the deviation between the new and old policies to avoid destructive updates and improve learning stability.

At each iteration, PPO collects a batch of trajectories using the current policy and performs multiple epochs of optimization on the following objective function:

$$L_t^{\text{PPO}}(\boldsymbol{\theta}) = \hat{\mathbb{E}}_t \left[\underbrace{\min(r_t(\boldsymbol{\theta}) \hat{A}_t, \text{clip}(r_t(\boldsymbol{\theta}), 1 - \epsilon, 1 + \epsilon) \hat{A}_t)}_{\text{clipped policy loss}} - c_1 \underbrace{(V_{\boldsymbol{\theta}}(s_t) - V_t^{\text{targ}})^2}_{\text{squared-error loss}} + c_2 \underbrace{\mathcal{H}(\pi_{\boldsymbol{\theta}}(\cdot | s_t))}_{\text{entropy bonus}} \right], \quad (2.7)$$

where $r_t(\boldsymbol{\theta}) = \frac{\pi_{\boldsymbol{\theta}}(a_t | s_t)}{\pi_{\boldsymbol{\theta}_{\text{old}}}(a_t | s_t)}$ is the probability ratio between new and old policies, $\hat{A}_t$ is the advantage estimate, typically computed using Generalized Advantage Estimation (GAE; Mnih et al., 2016), V_t^{targ} is the target value, $\mathcal{H}(\pi_{\boldsymbol{\theta}}(\cdot | s_t))$ is the policy entropy, and ϵ , c_1 , and c_2 are hyperparameters for clipping, value loss weight, and entropy bonus respectively.

The clipped surrogate loss limits the extent to which the updated policy can deviate from the previous one, enabling reliable and monotonic improvement in performance over time. Entropy regularization encourages sufficient exploration.

My Implementation. My implementation extends PFRL’s to the hybrid action setting. I adopted a shared neural network with two heads: an *actor* head that outputs action probabilities and a value *critic* head that estimates state values. In my experiments, this was a more effective architecture than having two independent

networks. The actor head splits a 256-dimensional feature vector into discrete and continuous branches: a softmax head for categorical action probabilities and a Gaussian head (state-independent covariance) for continuous action means and variances, sampling both at each timestep to form a factored joint policy. The critic head is a compact one-layer MLP producing a scalar $V(s)$, trained with PPO’s clipped value-function loss and generalized advantage estimation. Finally, I relied on PPO’s entropy regularization for exploration.

Soft Actor-Critic

Soft Actor-Critic (SAC) is an off-policy actor-critic algorithm designed to enable stable learning and efficient exploration by optimizing a trade-off between the expected return and the entropy of the policy (Haarnoja et al., 2018). The algorithm maintains three neural networks: a stochastic policy network (actor), a soft Q-function (critic), and a value network. The value network, V_ψ , is trained to approximate the expected entropy-augmented Q-value, using the following loss:

$$J_V(\psi) = \mathbb{E}_{s_t \sim \mathcal{D}} \left[\frac{1}{2} (V_\psi(s_t) - \mathbb{E}_{a_t \sim \pi} [Q_\theta(s_t, a_t) - \log \pi(a_t | s_t)])^2 \right], \quad (2.8)$$

where $\mathcal{D}$ denotes the replay buffer. The Q-function, Q_θ , is updated by minimizing the soft Bellman residual:

$$J_Q(\theta) = \mathbb{E}_{(s_t, a_t) \sim \mathcal{D}} \left[\frac{1}{2} \left(Q_\theta(s_t, a_t) - \hat{Q}(s_t, a_t) \right)^2 \right], \quad (2.9)$$

with the target defined as:

$$\hat{Q}(s_t, a_t) = r(s_t, a_t) + \gamma \mathbb{E}_{s_{t+1} \sim p} [V_{\bar{\psi}}(s_{t+1})], \quad (2.10)$$

where $V_{\bar{\psi}}$ is a slowly moving target network used to stabilize training.

The policy network, π_ϕ , is optimized using the reparameterization trick, enabling low-variance gradient estimates while encouraging high Q-values and high policy entropy. The policy objective is given by:

$$J_\pi(\phi) = \mathbb{E}_{s_t \sim \mathcal{D}, \epsilon_t \sim \mathcal{N}} [\log \pi_\phi(f_\phi(\epsilon_t; s_t) | s_t) - Q_\theta(s_t, f_\phi(\epsilon_t; s_t))], \quad (2.11)$$

where ϵ_t denotes noise sampled from a standard Gaussian distribution, and $f_\phi(\epsilon_t; s_t)$ represents a differentiable transformation that maps noise to actions. This formulation implicitly minimizes the Kullback–Leibler (KL) divergence between the policy and a Boltzmann distribution over Q-values, thereby promoting a principled balance between exploration and exploitation.

My Implementation. In this thesis, I extended PFRL’s SAC implementation to the hybrid action setting and relied on the algorithm’s entropy maximization term for exploration. Unlike PPO’s single value head, my implementation comprises three fully independent networks—one actor and two critics—each with its encoder, α , that directly processes the raw observation. The actor mirrors the PPO policy architecture. Each critic likewise encodes the same observation through α , concatenates with the predicted continuous action output, and produces Q-values for each discrete action. Then, Q-values are selected with respect to the actor’s sampled discrete action. By structuring the critics this way, I avoid the combinatorial explosion that would arise if I naively input every possible hybrid action pair, and it prevents gradient interference between discrete and continuous parameters.

2.3 Related Work

In this section, I review the environments commonly used in continual reinforcement learning (RL) and outline the motivations behind my own environment design. I first survey existing benchmarks and their limitations in capturing the essence of continual learning. I then discuss a range of algorithmic techniques developed to tackle the core challenges of continual RL—particularly, the tension between adaptation (plasticity) and memory retention (stability).

2.3.1 Existing Environments

Traditional RL testbeds (e.g., Bellemare et al., 2013; Tassa et al., 2018) have driven rapid progress in RL, but they are not tailored for continual RL. Accordingly, many continual RL studies resort to switching among different games to induce non-stationarity (e.g., Abbas et al., 2023), relying on clearly defined train-test boundaries and assuming a well-structured notion of tasks and episodes. However, such an approach is quite often contrived and, arguably, artificial. As I advocate in this thesis, I believe slower and smoother changes are much more representative of the non-stationarity in the problems used to motivate continual RL. JellyBeanWorld (Platanios et al., 2020) is the main platform for continual RL in non-episodic settings that I am aware of. In JellyBeanWorld (Platanios et al., 2020), agents navigate an infinite two-dimensional grid, interacting with various items by collecting or avoiding them. The agent’s states are partially observable. Although JellyBeanWorld is clearly valuable for continual RL research, it has simpler observations and (discrete) dynamics. Other proposals (e.g., Wiehe et al., 2018; Ansó et al., 2019) have toyed with resource-collection games, but these environments lack core continual learning aspects, such as changes in underlying dynamics driven by other bots. These features are critical for modeling non-stationarity.

Despite the limitations of current benchmarks for continual RL, different solutions try to mitigate phenomena caused by non-stationarity, such as loss of plasticity and catastrophic forgetting.

2.3.2 Solution Methods for Continual RL

A central objective of continual learning research is to develop algorithms capable of continually adapting to shifting data distributions (Ring, 1994; Thrun, 1998). While current reinforcement learning (RL) algorithms effectively learn optimal policies in stationary environments, they often struggle to adapt when the data distribution shifts during training (Abbas et al., 2023; Nikishin et al., 2023). The ability to learn

from new data, commonly called plasticity, is a critical property that enables agents to remain adaptable in evolving environments. However, recent studies have identified a loss of plasticity—manifested as a reduced ability to learn new “things”—as a core limitation of neural network-based algorithms in continual learning settings (Abbas et al., 2023; Dohare et al., 2024; Lyle et al., 2023). Another common challenge in continual RL is catastrophic forgetting, where the agent fails to retain previously acquired knowledge and cannot recall it when it becomes relevant again (Abbas et al., 2023). This degradation not only impedes learning of new behaviors but also exacerbates forgetting of old ones (Kirkpatrick et al., 2017), highlighting the need for solution methods that actively preserve or restore plasticity. Several strategies have been proposed in continual RL to mitigate plasticity loss and promote stable, adaptable learning. These include replay-based techniques that keep relevant observations in the replay-buffer, regularization strategies that protect important parameters, and training interventions that preserve gradient flow and representational flexibility.

Replay-Based Methods. Naive replay buffer approaches can confuse the agent by including irrelevant or outdated observations, hindering effective learning. To address this issue, several approaches have been proposed. CLEAR (Rolnick et al., 2019), for example, combines on-policy learning for rapid adaptation with off-policy replay and behavioral cloning to mitigate the catastrophic forgetting. Similarly, RECALL (Zhang et al., 2023b) improves generalization by adaptively normalizing target values and enhancing stability through policy distillation on previous tasks.

Regularization Techniques. Regularization methods address loss of plasticity and catastrophic forgetting in continual RL by constraining changes to parameters critical to previously acquired knowledge. Kaplanis et al. (2019) introduced a policy consolidation model in which the policy network interacts with a cascade of hidden networks capturing the agent’s policy at multiple timescales and regularizing the

current policy based on its history. More recently, Chung et al. (2025) proposed parseval regularization, which enforces the orthogonality of weight matrices to preserve gradient norms and improve training plasticity in dynamic environments.

Plasticity-Preserving Training Dynamics. Abbas et al. (2023) demonstrated that standard activation functions, like ReLU, cause network activation sparsity and gradient starvation over time, leading to plasticity loss; replacing these with Concatenated ReLU (CReLU) improves activation footprint and gradient flow. Complementary strategies include periodic reinitialization of rarely used neurons (Dohare et al., 2024), and plasticity injection—a lightweight intervention that increases the neural network plasticity without any change in either the number of trainable parameters or a bias in predictions (Nikishin et al., 2023). These approaches target the fundamental mechanisms behind plasticity degradation, enabling continual RL agents to maintain adaptability in evolving environments.

Although numerous efforts have been made to develop algorithms and techniques that address the limitations of current methods in continual learning, this thesis aims to introduce a problem setting/environment tailored explicitly for continual RL. To this end, I present **AgarCL**, a non-episodic continual RL platform inspired by the game Agar.io.

2.3.3 Agar-Like Environments.

Inspired by the game Agar.io, most of the results relying on Agar-like environments for evaluation consisted of assessing basic agent capabilities, mostly related to pellet collection (Wiehe et al., 2018; Ansó et al., 2019), without ever putting forward the environment as a key artifact. To the best of my knowledge, the most notable Agar-based benchmark is GoBigger (Zhang et al., 2023a), which scales Agar.io to multi-agent, team-based competition over fixed-length episodes (up to 14,400 frames). It shares certain surface-level similarities with my own, particularly in its Agar-style gameplay

mechanics, but it was introduced to support a fundamentally different problem. Its main goal is to provide a platform for studying collective behaviors in multi-agent reinforcement learning. Thus, its features consist of supporting multiple teams of agents (in the original game, collaboration should emerge from communication) in settings with agents organized into a few teams. The larger environment configuration supports 24 agents organized into four teams (6 each). The game is designed to be *episodic*, as the larger maps artificially have episodes with at most 14,400 frames (12 minutes). These are all in direct contrast to the continual learning problem I focus on, which involves potentially unbounded interactions and no predefined teams, but with a significantly larger number of agents. GoBigger also does not support pixel-based observations; it only supports something akin to my symbolic observations, which include information about objects’ positions and velocities. Finally, I benchmarked the frames per second (FPS) over ten independent trials using the `st_t6p4`⁴ configuration—the largest map setting in the GoBigger implementation. Our environment significantly outperforms GoBigger in simulation speed, achieving an interquartile mean (IQM) of 4,212 FPS with GoBigger-style observations, compared to GoBigger’s IQM of 205 FPS under the same observation setup. This experiment used 2 cores of an *Intel Xeon Gold 6448Y* CPU. In the next chapter, I discuss the game with respect to my implementation.

⁴This configuration typically includes 1000 pellets, a 144×144 arena, and 14,400 frames per episode. I modified the GoBigger implementation to support a single player.

Chapter 3

AgarCL: Agar.io for Continual Reinforcement Learning

I begin by discussing my main contribution: Agar.io for Continual Reinforcement Learning (AgarCL). This environment for research in sequential decision-making supports settings where the environment is constantly changing, but not too abruptly. I believe this setting can potentially instantiate the big world hypothesis (Javed and Sutton, 2024), and it is a setting in which learning continually is more effective solution than a fixed policy learned for a predetermined number of steps.

3.1 The Game: Agar.io

Agar.io is a multiplayer online game in which each player controls one or more circular cells. The game draws an analogy to a Petri dish containing interacting cells, food sources, and viruses. Players aim to grow by absorbing smaller entities, such as static pellets and other players’ cells, while avoiding larger opponents and strategically interacting with viruses, which can fragment or shield cells depending on their size (see Figure 3.3). Players can also split their cells for tactical reasons, enabling simultaneous control of multiple cells. The game’s rules are simple, but complex dynamics emerge from these interactions. Figure 3.1 depicts some game dynamics.

Three types of entities populate the arena: static *pellets*, player-controlled *cells*, and *viruses*. Pellets are randomly scattered, maintain a fixed size, and grant 1 mass

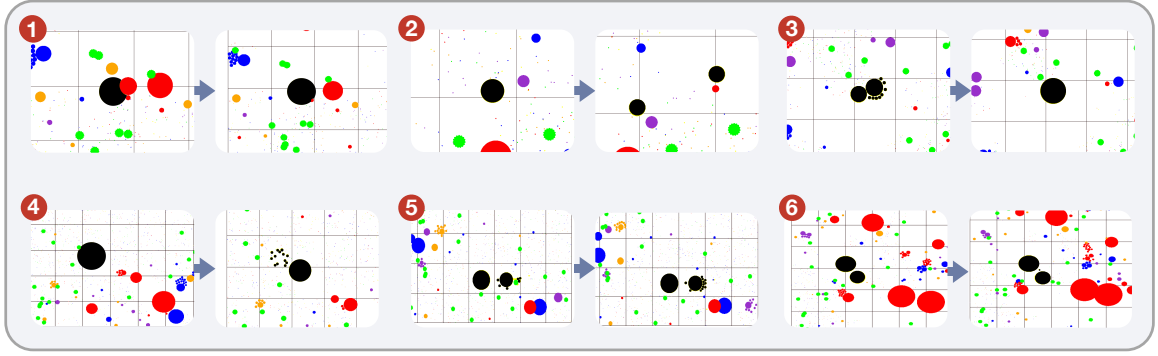


Figure 3.1: Environment dynamics and actions in **AgarCL**, with the agent shown in black. ① The agent can eat smaller cells to gain mass. ② The **SPLIT** action divides each agent’s cells in half and propels them in a chosen $\langle x, y \rangle$ direction, allowing slower agents to catch faster ones. Each cell moves at a speed inversely related to its mass. ③ Cells can later merge if brought close together. Viruses add complexity: depending on mass, the agent can either ④ be split by a virus or ⑤ consume it. ⑥ The agent can also **EJECT** mass in a chosen direction. Any agent can eat this mass. Ejecting enough mass into a virus spawns a new one and propels the original, enabling smaller cells to attack larger ones by pushing viruses into them. While players in the online game use these mechanics to communicate, this is outside the scope of my work; bot behaviour is scripted.

when consumed. Cells, the primary agents of interaction, can consume pellets, viruses, and smaller cells. When consuming another cell, the player gains the mass of the absorbed cell. Viruses have a mass of 100; depending on the player’s size, a virus can either be absorbed or cause the cell to split.

Players start as a small cell of mass 25. If the player’s cell(s) is consumed by a larger cell, the player’s cell is eliminated from the game. The player then respawns with the same initial mass. While this could be perceived as an episodic task due to the repeated nature of the interaction, more successful agents are expected to “live” much longer and not see such resets. Additionally, the cell that absorbed the player’s cell maintains its new mass after the player has respawned, such that actions from a previous “episode” impact the new one. Currently, my implementation supports the single-player setting; the other players, called *bots*, follow heuristic-based behavior. I made this choice to focus on the continual aspect of the problem. I refer the reader to the **GOBIGGER** (Zhang et al., 2023a) research platform if they are interested in the multi-agent aspect of the game.

In **AgarCL**, I implemented four heuristic bots—Aggressive, Aggressive-Shy, Hungry, and Hungry-Shy—to explore how simple rules shape emergent game dynamics systematically. All bots follow a fixed policy that always targets the nearest pellet, but differ in their interactions with other agents/bots. The four bots’ policies are:

- **Aggressive:** This bot first looks for any smaller opponent within a defined radius and attempts to consume it; if no suitable target is found, it switches to pellet collection.
- **Aggressive-Shy:** Like the Aggressive bot, it will hunt smaller opponents, but if a larger opponent approaches within its “shy” radius, it immediately flees and only returns to hunting once the threat has passed.
- **Hungry:** This bot ignores other players and chases the closest pellet at every step.
- **Hungry-Shy:** Focused on pellet foraging like the Hungry bot, it additionally monitors for larger opponents: if one comes too close, it retreats before resuming its hunt.

The ever-changing nature of the game has many sources. New pellets are generated every 10 time step while there are fewer than 500 pellets in the arena. Likewise, new viruses are generated at every time step whenever their count is below 10. Additionally, every cell belonging to an agent loses 0.2% of its mass each second (60 ticks). As cells grow, they move more slowly. Their speed, v , is determined by the function $v = \text{mass}^{0.439}$. Thus, smaller opponents must continually evade larger players, steer clear of corners, or use viruses for defense. Players can feed viruses until they are large enough in order to split them among larger players. Typically, a split occurs if a player feeds a virus seven times. The player’s field of view also varies according to its mass, as the game needs to depict all of its cells. I refer the reader to Figure 3.2 for an example of how an Agar.io game can progress.

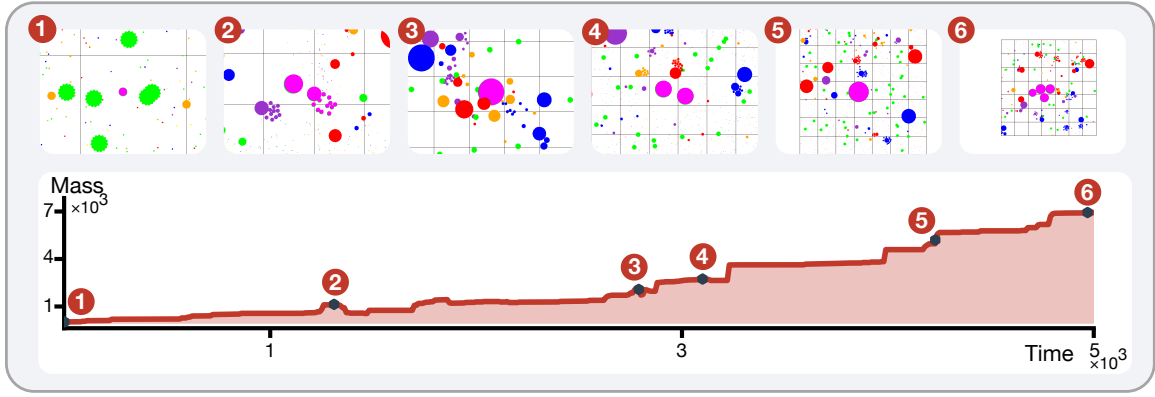


Figure 3.2: The first 5,000-time steps of an expert trajectory show game progression. The agent, shown in pink, steadily gains mass: ① starting small, it eventually passes over a virus; ② splits into smaller cells; ③ grows large enough to consume viruses and other agents; ④ splits to attack, increasing its speed. As it grows, the view zooms out to show its full body. ⑤ The agent ultimately surpasses all opponents in mass and ⑥ can see the entire arena. Mass decays over time, and the view would zoom in again if mass dropped. This run was recorded while I was playing.

Additional nuances to the game increase the complexity of a truly effective strategy. For example, when an agent has split into 14 cells, it can no longer split into more. To prevent larger players from simply consuming all viruses, a penalty activates when an agent consumes three or more viruses in a row within a minute. After hitting three viruses, the mass decay rate increases, accelerating further with each additional virus consumed. This penalty persists across respawns.¹

Due to all these dynamics, players must continuously balance the need for immediate gains with potential long-term risks. A common tactic involves splitting a large cell to increase the chances of consuming smaller opponents, who might otherwise be out of reach due to the large cell’s slower speed. However, this aggressive move introduces significant risks: the newly split, smaller cells can be quickly eaten by larger opponents or even by other small, agile players. This delicate balance between offense and defense, combined with the need to manage mass effectively, forms the core of competitive play. The game’s design inherently promotes these strategies by

¹I admit some of these rules are quite complex and likely unnecessary to capture the spirit of the game, but I decided to discuss them to emphasize my focus on replicating the original game dynamics, which were interesting to people and free of experimenter bias as it was not designed with artificial intelligence research in mind.

ensuring that no single approach guarantees success in every situation, requiring the agent to learn and adapt to survive continually.

3.2 Agent-Environment Interface

Once the environment dynamics are defined, I must describe how an agent can interact with such an environment. I follow the standard RL formalism where the agent-environment interaction takes place in terms of actions, observations, and rewards. I describe those below and in Figure 3.3.

Reward Function I define the reward function as the difference in the agent’s mass between two consecutive time steps to keep the reward bounded. Formally, without mass decay, this is expressed as:

$$R_t = m_t - m_{t-1}. \quad (3.1)$$

Importantly, the problem is non-episodic, so the agent’s death does not terminate the episode. I chose not to incorporate domain knowledge by penalizing the agent for dying. Upon re-spawning, it receives a reward equal to its initial mass minus its death mass, discouraging death as a cost-free reset when the agent is bigger than its initial mass.

Observation Space How the agent perceives the environment is key in any RL problem. Here, I emphasize a pixel-based top-down view of the environment. Such a choice requires the agent to deal with a high-dimensional, partially observable, and ever-changing observation stream. The figures throughout the paper provide many examples of renderings of the actual observations received by the agent, with Figure 3.2 particularly emphasizing the evolution of such an observation stream.

Importantly, variants of Agar.io have been used in RL research before (Wiehe et al., 2018; Ansó et al., 2019; Zhang et al., 2023a), but none of the cited articles have used (or supported) Agar.io with high-dimensional pixel-based observations. In previous

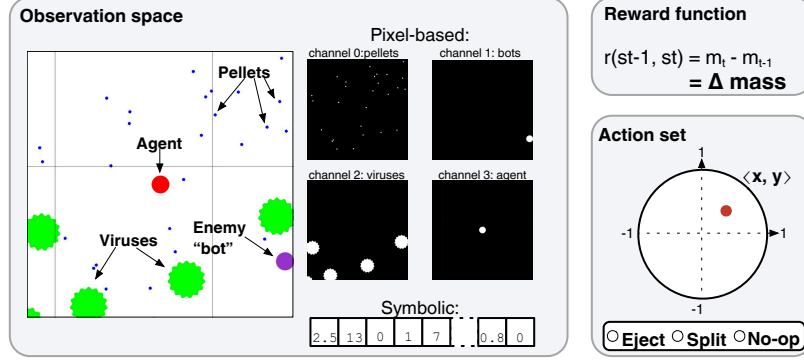


Figure 3.3: Agent-environment interface and main entities in **AgarCL**. The agent has access to one of two observation types: pixel-based or symbolic. The pixel-based observation includes four channels that represent different game entities: pellets, bots, viruses, and the agent itself. Alternatively, the agent can receive a symbolic observation, consisting of pre-processed features such as the distances to nearby enemies, pellets, and other entities. The reward function is defined as the change in the agent’s mass between two consecutive time steps. The action space in **AgarCL** is hybrid: at each time step, the agent selects an $\langle x, y \rangle$ coordinate mimicking where a human player would point their mouse. Additionally, the agent decides whether to split, eject mass, or just move.

work, the observation always consisted of some variation of a grid-like observation. Although I do not provide results in such a setting, **AgarCL** also supports using more symbolic representations such as these. I discuss both choices in more detail below.

Pixel-Based Observation: I represent the game screen, $\mathbf{O}_t$, as a three-dimensional tensor $\mathbf{O}_t \in \mathbb{R}^{N \times N \times 4}$, where $N \times N$ denotes the spatial resolution of the game screen (by default, $N = 128$). The third dimension corresponds to separate channels for pellets, viruses, enemies, and the agent (including gridlines). Fig. 3.3 depicts an example of such an observation (notice the gridlines are too faint to be seen).

Grid-Like Observation: I adapted the GoBigger observation (Zhang et al., 2023a) to the single-agent setting. It is divided into two main parts: the global state and the player state. The global state captures information such as the map size, the number of frames in the game, and the number of frames that have passed. These features provide temporal and spatial context for the agent. In the player state, I focus solely on the current agent’s information, including its field of view, visible entities in that

space, the agent’s score, and its available actions. The overlap field is critical, as it captures details about nearby pellets, viruses, and cells (each with associated positions, velocities, and other necessary attributes).

Action Space The action space is hybrid, and an agent can perform two types of actions simultaneously. The agent controls its cells by selecting a point on the screen, mimicking a human positioning the cursor on the screen, which determines the direction in which all of its cells move. The range of these *continuous* actions is between $[-1, 1]$ in two dimensions, $\langle x, y \rangle$. Simultaneously, at each time step, the agent needs to make a *discrete* choice between splitting, ejecting pellets, or simply moving, with no further discrete action, as shown in Figure 3.3.

The split action divides a cell into two equal parts, each with half the original mass, provided the produced cell has at least a mass of 25, requiring the agent’s mass to be at least 50 for it to work. One of the newly split cells is propelled toward the cursor with significant momentum. After splitting, the player must manage multiple cells simultaneously, using the cursor to navigate each cell effectively. The eject action ejects a small blob of mass (called a pellet) from each cell. The pellets are also ejected toward the cursor. They can be consumed by cells or viruses. Every action taken is repeated four times, and the observation received by the agent is generated by the environment after the execution of the selected action(s) four times; I call this value *frame skip*.

Finally, **AgarCL** has stochastic dynamics. Noise sampled from a normal distribution $\mathcal{N}(0, 1)$ is added to the continuous actions the agent sends to the environment.

3.2.1 Technical Details

AgarCL is released as free, open-source software under the terms of the MIT license. It is implemented on top of the AgarLE (Deaton, 2018), an incomplete implementation of Agar.io with an OpenAI Gym (Brockman et al., 2016) interface. **AgarCL**’s source code is publicly available in:

<https://github.com/machado-research/AgarCL>

AgarCL’s core simulation engine and data pipeline are implemented in C++, rendering is handled by OpenGL, and a Python interface, built with Pybind11, is also supported. In my experiments, when looking at the interquantile mean (IQM) over ten independent trials, a random agent, written in Python, receives 2,016 frames per second with a frame skip of 1. With a frame skip of 4 (default), the agent receives 1,163 observations per second, which represents 4,652 frames in the game and 1163 actions selected by the agent. I report the IQM to discard fluctuations due to other jobs in the system. This experiment used 2 cores of an *Intel Xeon Gold 6448Y* CPU and one *NVIDIA L40s* GPU. Correspondingly, the IQM when evaluating AgarCL without an access to GPU was 1,440 and 1,092, respectively.

Chapter 4

Evaluating AgarCL as a Platform for Continual Reinforcement Learning

To evaluate the learning capabilities of agents in AgarCL, I use three deep reinforcement learning algorithms: Deep Q-Network (DQN; Mnih et al., 2015), Proximal Policy Optimization (PPO; Schulman et al., 2017), and Soft Actor-Critic (SAC; Haarnoja et al., 2018).¹ These algorithms span a spectrum of learning paradigms—value-based and policy-gradient—to characterize the overall performance of different classes of algorithms.

As shown in Figure 4.1, each agent has an encoder component, α , which processes $128 \times 128 \times 4$ raw images. The encoder comprises three convolutional layers with kernel sizes of 8×8 , 4×4 , 3×3 , and strides of 4, 2, and 1. Each convolutional operation is followed by layer normalization (Ba et al., 2016) and a rectified linear unit (ReLU). The final output, a $32 \times 12 \times 12$ feature map, is flattened and passed through a fully connected layer to produce a compact embedding. This embedding serves as the basis for predicting both discrete and continuous actions, supporting hybrid control in the AgarCL environment.

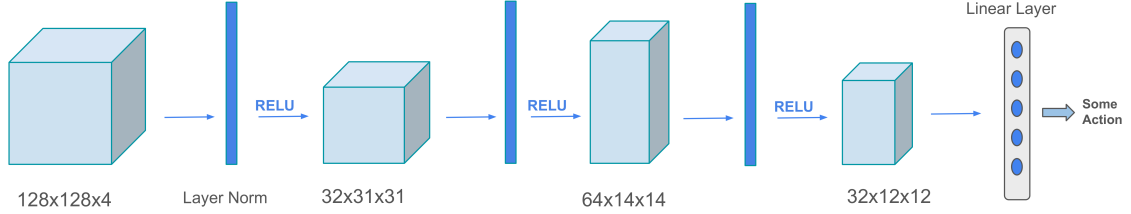


Figure 4.1: Architecture of the encoder module, α , used by each agent in **AgarCL**. The encoder processes raw image observations of shape $128 \times 128 \times 4$ through convolutional layers, followed by layer normalization and ReLU activations. The resulting compact feature representation supports hybrid control.

4.1 Pellets Collection Tasks

AgarCL is a challenging environment, which I believe is needed to propel the design of the next generation of continual RL agents. It does not only require agents to be able to reliably learn continually, which is already a significant challenge by itself (Dohare et al., 2024), but also to address open problems in continual RL research, such as exploration without resets, long-term credit assignment, representation learning in light of a varying observation stream, and more.

In light of the challenges mentioned above, to validate and understand the proposed environment, I first take a step back and evaluate different agents in a collection of mini-games I designed within Agar.io. These mini-games were meant to isolate specific environmental challenges, such as non-stationarity and the non-episodic nature of the problem.

Importantly, I tuned each algorithm’s hyperparameters in every mini-game, each trained with three different random seeds. Then, I selected the best configuration with the maximum mean return of the final 100 episodes of a 20-million-frame run (or 100 time steps of 20 million-frame in continuing tasks). To avoid maximization bias, I evaluated this configuration with ten independent trials. Details on the hyperparameter ranges and selection criteria are discussed in Appendix B.

¹My implementations are available in <https://github.com/machado-research/AgarCL-benchmark>.

4.1.1 Non-Stationarity through Mass Variation

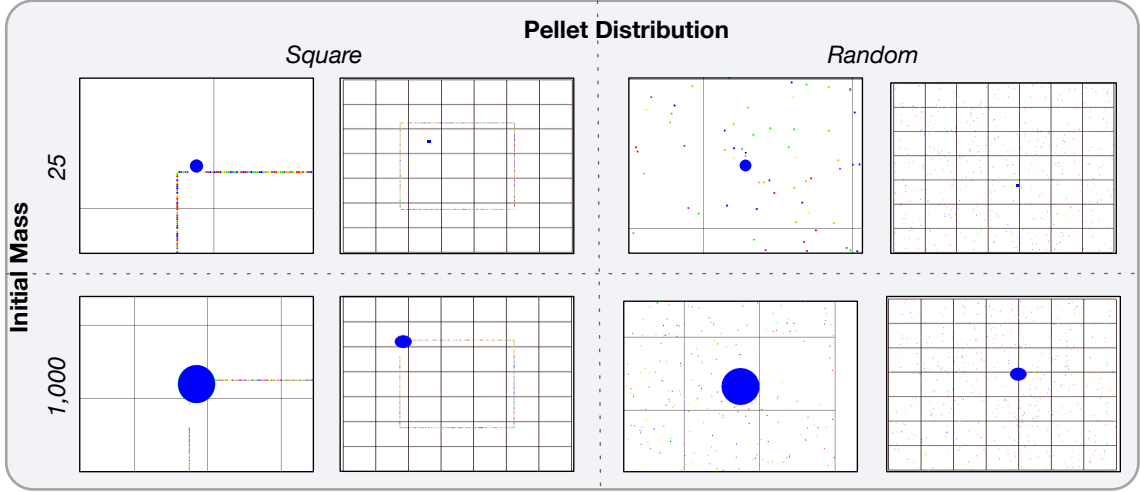


Figure 4.2: Pellet Collection mini games. Six mini-games were described in the main thesis. They were defined in terms of the pellet distribution in the arena, the agent’s initial mass, and whether the agent’s mass decays (which is hard to depict in an image). Note, I do not have eight mini games because starting with a mass of 1,000 and no mass decay is an uninteresting setting. In each quadrant, the agent’s actual view at the beginning of the mini game is shown on the left, and, to give a sense of scale, a zoomed-out perspective of the same setting is shown on the right. The agent is depicted in blue.

I first consider a set of progressively challenging mini-games that require the agent to collect pellets. The environment in such mini-games does not have other bots or viruses. Importantly, this first set of mini-games are *episodic* for simplicity. Episodes were 500 and 3000 time steps long (first three vs latter three mini-games) and had a single start state. Figure 4.2 depicts screenshots of these mini-games.

These mini-games were designed to evaluate the impact of three different aspects of the game: (i) (short-term) exploration, (ii) the impact of the mass decay, and (iii) the impact of having a bigger mass, thus being much slower but also able to split. To factor out exploration, delayed credit assignment, and partial observability as much as I could, MINI-GAME 1 consists of collecting pellets laid out in a square path. In such a setting, the agent starts with the default starting mass of 25 without mass decay. This is the simplest mini-game, but it still features non-stationarity: as the agent gains mass, it becomes larger and correspondingly slower, but all the agent needs to

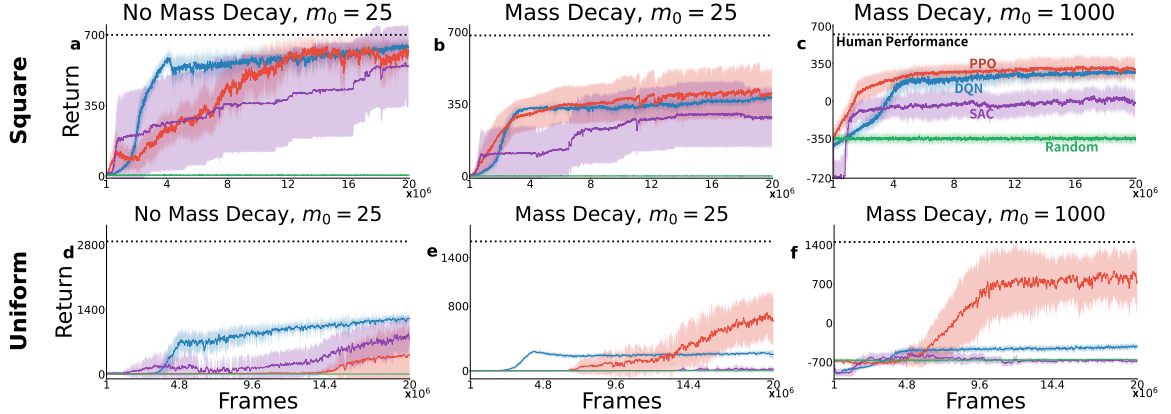


Figure 4.3: Performance of RL baselines on *episodic* pellet-collection mini-games. Panels (a), (b), and (c) show the performance on *the square-path tasks* (mini-games 1, 2, and 3), while Panels (d), (e), and (f) show the performance on *randomly regenerated tasks* (mini-games 4, 5, and 6). Note that the y-axis scales are different from each other. The dashed line marks the final average human-expert performance, and the green line the random policy. The shaded region shows the 95% confidence interval over 10 runs, computed using the t-distribution.

learn is to follow the dense path of “breadcrumbs”. MINI-GAME 2 is the same as MINI-GAME 1, but with mass decay (thus, the amount of negative reward received by the agent varies according to its mass); and MINI-GAME 3 is the same as MINI-GAME 2, but the agent’s starting mass is 1,000 (thus, the agent starts much slower, loses more mass per step, and can split). MINI-GAME 4, 5, and 6 match the first three mini-games, except that pellets are randomly scattered in the environment, not in a dense square path the agent can blindly follow. In these latter three mini-games, credit assignment is more delayed, partial observability matters more, and exploration is slightly more difficult (although the resets upon episode termination still simplify the task).

Figure 4.3 presents the performance of the algorithms I evaluate in **AgarCL** across these mini-games. A table with numerical results summarizing these plots is available in Appendix C. Shortly, in the simpler problem, MINI-GAME 1, DQN, PPO, and SAC achieve a very similar performance to that of a human player. A particularly interesting result is that introducing the ever-changing dynamics makes the problem much harder, even in the square-path task, as all agents achieve approximately half

of the human performance here. Starting the agents with a larger mass makes the problem even more challenging, as a random policy accumulates negative rewards and at least SAC is unable to go much above 0. These insights carry over to the mini-games in which pellets are uniformly distributed throughout the arena. However, these mini-games seem to be a much harder setting for the algorithms I considered and, once mass decay is introduced, only PPO is able to learn *something*.² Also, surprisingly, in MINI-GAME 6, where the agent begins with a large initial mass and the mass decay is active, performance shows signs of improvement over MINI-GAME 4 with PPO. This counterintuitive result may be due to the reduced partial observability: a larger mass increases the agent’s field of view, thus mitigating some exploration difficulty. However, this hypothesis remains to be further investigated. Note that SAC is an algorithm 1.5–3× slower than PPO in terms of wall clock time and, at least in my experiments, it is not more sample-efficient than PPO.³

4.1.2 Continuing and Continual Problems.

After evaluating the discussed algorithms in the episodic mini-games, I made those mini-games continual. I removed the episodic resets and, instead, I replenish all pellets in the environment every 600 frames (150 actions if using a frame skip of 4). The maximum number of pellets is 500, which are replenished somewhat randomly (although still within the path for the square-path mini-games). I retuned the hyperparameters I considered in each mini-game for all algorithms evaluated. Figure 4.4 presents the performance of the evaluated algorithms in this setting. A table with numerical results summarizing these plots is available in Appendix C.

In my results, I observe that this task is exceedingly difficult. In square-path tasks (making the tasks continual, not only continuing), no agent is able to succeed.

²In MINI-GAME 5, PPO effectively mitigates the challenge of mass decay by strategically ejecting food when a significant loss in mass is detected. This behavior enables the agent later to consume the ejected food and nearby pellets, resulting in an unexpectedly strong performance.

³To obtain these results, I ran over 900 jobs lasting between 15 hours (DQN and PPO) and 23 hours (SAC).

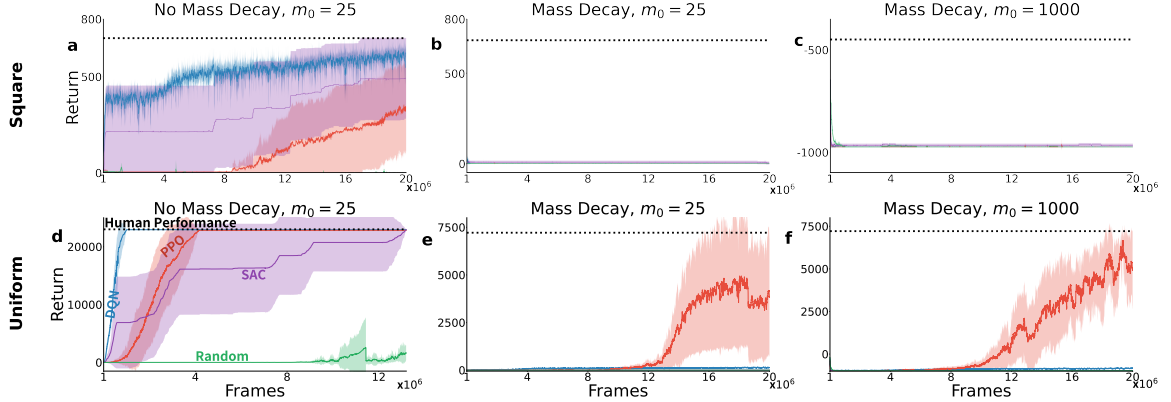


Figure 4.4: Performance of RL baselines on *continual* pellet-collection mini-games. Panels (a), (b), and (c) show the performance on *the square-path tasks* (mini-games 1, 2, and 3), while panels (d), (e), and (f) show the performance on *randomly regenerated tasks* (mini-games 4, 5, and 6). Note that the y-axis scales are different from each other. The dashed line marks the final average human-expert performance, and the green line the random policy. The shaded region shows the 95% confidence interval over 10 runs, computed using the t-distribution.

This is in part due to exploration and partial observability. If the agent strays away from the square path and stops being able to see it, the agent struggles to find the path back, and without resets, it needs to find the path back to continue learning. In this continual case, the uniform distribution of pellets makes the problem easier, maybe because the agent

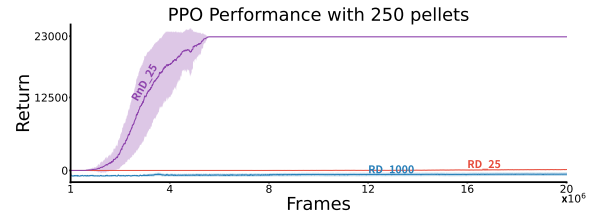


Figure 4.5: PPO performance on *continuing randomly regenerated pellet-collection* mini-games 4, 5, and 6. RD-1000 indicates a setting with mass decay and an initial mass of 1000, while RnD-25 has no decay and starts at 25. The plateau in RnD-25 after 1M steps reflects saturation in mini-game ⑤. Shaded areas show 95% confidence intervals over 10 evaluation runs.

can always expect to see at least a few pellets. This is slightly different when I reduce the number of pellets in the environment by half, a setting in which the baselines start to struggle. PPO failed to learn effectively in this setting. Addressing this task requires resolving the exploration challenge. Performance is shown in Figure 4.5.

Note that, in the results I report here, I had to trim the learning process in MINI-GAME 4 in Figures 4.4 because the agents were achieving the maximum mass

allowed in the environment and, without mass decay, they would continue growing. Also, I extended the last point of MINI-GAME 4 as a line to be consistent with the other MINI-GAME 5 and 6. As before, PPO seems to be the most robust algorithm I considered.

Assessing PPO with RNNs in Continuing and Continual Settings.

The natural question to ask is whether an approach designed to tackle partial observability would be effective in the mini-games with a higher degree of partial observability. From the results in the previous section, PPO emerged as the most robust algorithm in my environment. Thus, I re-ran the pellet collection MINI-GAMES using a recurrent version of PPO to assess whether temporal memory would enhance performance.

Specifically, I employed a PPO architecture augmented with Gated Recurrent Units (GRUs; Chung et al., 2014). The model comprises three convolutional layers, followed by a 256-unit GRU, and a final linear layer that splits into actor and critic heads. All other experimental procedures remained consistent with those described in the thesis. To ensure fairness, each mini-game was subjected to separate hyperparameter tuning tailored to the recurrent architecture.

As shown in Figure 4.6, introducing recurrence via GRUs did not lead to much improvements. PPO with GRUs performed well in relatively simple scenarios—for instance, mini-game 1 (see panel [\(a\)](#)), where it achieved human-level performance. However, it struggled in more challenging tasks, such as mini-games 2 (panel [\(b\)](#)) and 3 (panel [\(c\)](#)) with mass decay. Notably, in the uniform variation ([\(d\)-\(f\)](#)), the performance of PPO with GRUs was even worse than that of the non-recurrent version, indicating that recurrence did not help to increase the performance on this set of mini-games. Table 4.1 summarizes these plots with numerical results.

These results further underscore a key point: **AgarCL** poses unique and significant challenges beyond those found in standard RL benchmarks, highlighting the need for novel approaches better suited to continual RL.

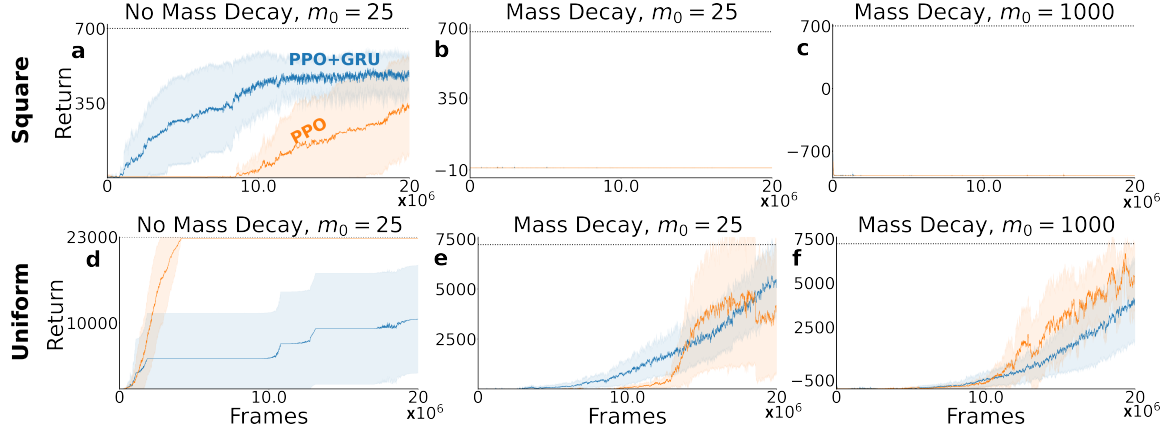


Figure 4.6: Performance of PPO+GRU and PPO on *continual* pellet-collection mini-games. Panels (a), (b), and (c) correspond to the *square-path* tasks (with no mass decay, mass decay $m_0 = 25$, and mass decay $m_0 = 1000$, respectively), while panels (d), (e), and (f) correspond to the *uniformly regenerated* tasks with the same decay settings. The shaded regions show the 95% confidence intervals over 10 evaluation runs, computed using the t-distribution. Note that the y-axis scales differ between panels.

Table 4.1: Performance of PPO with GRU-based recurrent architecture across six mini-games. Results are averaged over the last 100 data points. Values in parentheses denote standard deviations.

Algorithm	Mini-Game 1	Mini-Game 2	Mini-Game 3	Mini-Game 4	Mini-Game 5	Mini-Game 6
PPO	335 (3.0)	0 (0)	-975 (0)	22,463 (48.9)	4,018 (126.8)	5,064 (113.6)
PPO + RNN	484 (10.0)	0 (0)	-974 (0.3)	20,533 (250)	5,329 (30.9)	4,036 (22.4)

4.2 Interacting with Other Agents

In the previous section, I highlighted the difficulty agents face in collecting pellets when their total number is limited, particularly due to exploration and the mass decay mechanism. A natural follow-up question arises: how does an agent’s performance change when the environment includes a standard number of pellets but also features an additional bot? To isolate this factor, I conducted experiments in an episodic setting, where each episode terminates either when the agent is eaten or after 10,000 time steps—whichever occurs first. I conducted these experiments with standard configuration, which I call LARGE-DENSE setup, and with a limited arena (more later in this section), which I call SMALL-SPARSE setup.

4.2.1 Large-Dense Setup

I designed two main experiments for each bot to evaluate whether the current learning agents used in this study can recognize and adapt to the fixed policies of various bots. The first setup uses a standard number of pellets (500) and a normal-sized arena (350×350), referred to as the LARGE-DENSE setup. In these experiments, I used two types of fixed-policy bots: the *hungry* bot, which focuses solely on collecting pellets, and the *aggressive* bot, which prioritizes consuming any entity smaller than itself.

To better understand the agent’s behavior, it is essential to visualize the learning curve of the fixed-policy bot. Accordingly, each panel presents the “learning” curve (in fact, the accumulated reward) of the bot alongside that of the other learning agent.

As shown in Figure 4.7, both DQN and SAC struggled in the presence of the aggressive and hungry bots. Gameplay videos recorded at various evaluation checkpoints revealed that these agents frequently became stuck in corners—a failure mode also observed in Figure 4.3e—highlighting a recurring weakness in their behavior. Consequently, the *aggressive* bot could consume the learning agents quickly, as reflected by the early plateau in its return curves in panels ④ and ⑥. Also, the performance of the *hungry* bot in panels ① and ③ appears slightly better, focusing on collecting pellets rather than following the learning agents. In contrast, the PPO agent exhibited more robust behavior. Panel ⑤ shows that PPO learned to avoid the aggressive bot effectively while collecting many pellets. This evasive strategy contributed to the steady increase observed in the agent’s return, albeit at a slower rate than its adversary. However, in the hungry setting (panel ②), PPO failed to consume the bot, even though the hungry bot did not attempt to attack it. This indicates that while PPO is effective at survival, it may not exploit opportunities to eliminate non-aggressive opponents.

This observation raises the question of whether PPO’s success was due to the large arena and high pellet density. The SMALL-SPARSE setting, discussed next, addresses

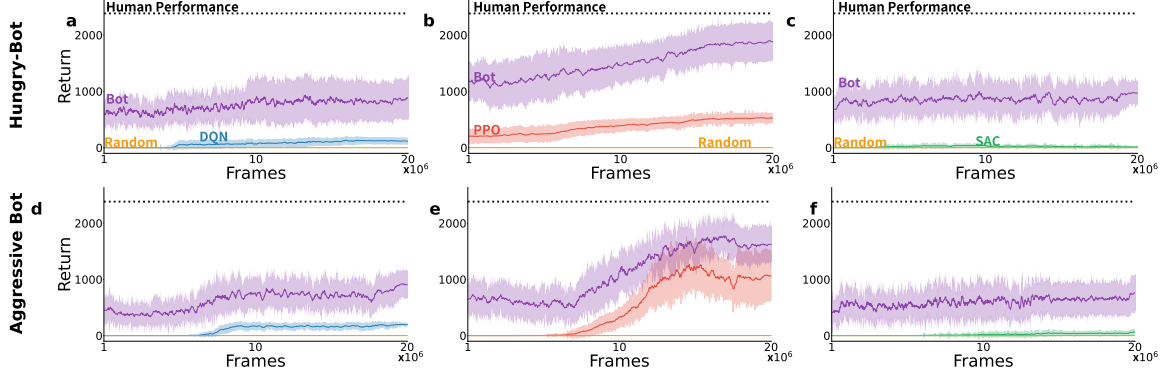


Figure 4.7: Performance of RL baselines—DQN, PPO, and SAC—on *episodic* other-agent mini-games. Panels (a) and (d) show the performance of *DQN*, Panels (b) and (e) show the performance of *PPO*, and Panels (c) and (f) show the performance of *SAC*. The top row ((a)–(c)) corresponds to the *hungry-bot* task (mini-game 7), while the bottom row ((d)–(f)) corresponds to the *aggressive-bot* task (mini-game 8). Each experiment features a fixed-policy bot—either hungry or aggressive—in a standard-sized arena (350×350) with 500 pellets. Both the agent and the fixed-policy bot start with an initial mass of 25. Shaded regions represent the 95% confidence intervals over 10 independent runs, computed using the *t*-distribution.

this question directly.

4.2.2 Small-Sparse Setup

To limit the agent’s ability to avoid interaction with fixed-policy bots, I introduced a more constrained environment characterized by a smaller arena (200×200) and a reduced number of pellets (250), referred to as the SMALL-SPARSE setting. This configuration increases the likelihood of direct encounters between the learning agent and the fixed-policy bot.

Consistent with previous findings, the DQN and SAC baselines failed to demonstrate effective learning, irrespective of the type of fixed-policy bot present. This is primarily attributable to recurring behavioral failures, such as becoming trapped in corners or failing to evade the bot—issues that persisted under the more restrictive conditions imposed by the SMALL-SPARSE setting.

Interestingly, the PPO agent also failed to learn in this environment, regardless of whether it was paired with the aggressive or the hungry bot. PPO could not consistently evade the aggressive bot or exploit the passive behavior of the hungry one.

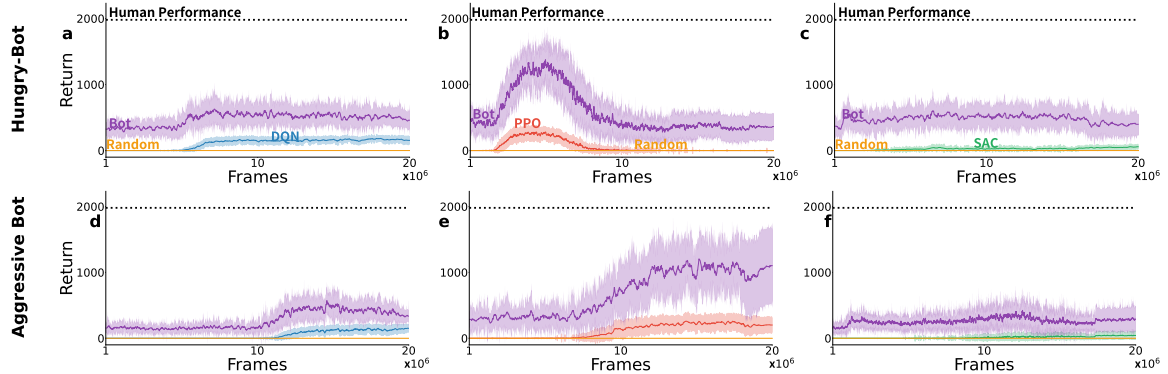


Figure 4.8: Performance of RL baselines—DQN, PPO, and SAC—on *episodic* other-agent mini-games. Panels (a) and (d) show the performance of *DQN*, Panels (b) and (e) show the performance of *PPO*, and Panels (c) and (f) show the performance of *SAC*. The top row (a–c) corresponds to the *hungry-bot* mini-game, while the bottom row (d–f) corresponds to the *aggressive-bot* mini-game. Each experiment features a fixed-policy bot—aggressive or hungry—in a limited arena (200×200) with 250 pellets. The agent and the fixed-policy bot start with an initial mass of 25. Shaded regions represent the 95% confidence intervals over 10 independent runs, computed using the *t*-distribution.

In panel (e), PPO’s return clearly plateaus early, while the aggressive bot’s return steadily increases, indicating its ability to consume the agent with ease repeatedly. Even more interesting is PPO’s performance against the hungry bot. Despite the absence of an active threat, PPO failed to take advantage of the opportunity to survive and collect rewards. This suggests a significant limitation in the agent’s ability to exploit benign opponents, likely due to restricted mobility and sparse rewards. The consistent flattening of the agent’s learning curves across all settings further reinforces this observation, indicating minimal reward acquisition and limited policy improvement over time.

These results underscore the sensitivity of all three algorithms to environmental constraints such as limited space and sparse resources. The findings highlight a need for more robust learning strategies to adapt to densely interactive and resource-scarce environments.

4.3 Complex Dynamics and Long-Term Consequences

Interacting with a single opponent in a limited arena is already a challenging task for the learning agent, especially in the absence of viruses. In this mini-game, I investigate how introducing viruses changes that interaction. To reduce exploration complexity, the mini-game is configured to be fully observable. As illustrated in Figure 4.9, the opposing bot remains stationary, and a clear, linear arrangement of viruses is positioned between the agent and the bot.

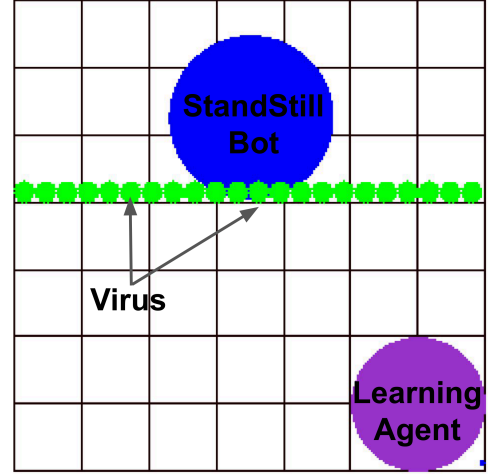


Figure 4.9: Virus mass: 100; stationary bot: 5000; agent: 3000. No pellets, no virus regeneration, and no mass decay.

Notably, there are no pellets in this mini-game; the agent begins with a mass of 3000 and must engage with a larger, stationary bot that has a mass of 5000. To simplify dynamics further, mass decay is disabled. Training in this mini-game follows an episodic setup: an episode ends when one agent is eaten or after 1000 time steps.

The objective is for the agent to learn to eject food toward the viruses, triggering them to split the stationary bot. The agent can then be split into different fragments. Finally, it passes through the virus field, and consume the smaller bot fragments.

As shown in Figure 4.10, all learning agents failed to discover

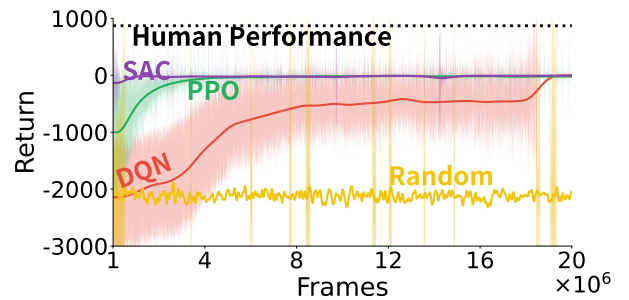


Figure 4.10: Performance of RL baselines—DQN, PPO, and SAC—on the *virus-based* mini-game. The learning agent (mass 3,000) is separated from a standstill bot (mass 5,000) by a line of static viruses (mass 100). The arena is fully observable with no pellets, mass decay, or virus respawning. Shaded regions show the 95% confidence interval across 10 runs using the t -distribution.

this strategy. Instead, they converged to a passive behavior, remaining stationary for the entire episode. This highlights the difficulty these agents face in learning to leverage environmental elements, such as viruses, for strategic interaction, even in a simplified and fully observable setting.

So far, identifying the best hyperparameters for each mini-game has proven to be a nontrivial task. In the next section, I shift focus to the critical issue of hyperparameter tuning in continual reinforcement learning, which plays a significant role in the observed performance differences across settings and algorithms.

4.4 Hyperparameter Tuning in Continual Reinforcement Learning

I ran thousands of GPU jobs to tune and evaluate the baselines discussed in the previous section and did so on a per-environment basis. Different algorithms have very different sensitivities to different hyperparameters. To test the robustness of the hyperparameter I used in my experiments, I evaluated the hyperparameter used in a mini-game in another mini-game. In many tasks, PPO tuning on a particular mini-game often collapses when applied to a different mini-game. Interestingly, in some mini-games, I observe an unexpected pattern with SAC where using hyperparameters tuned for different mini-games yields better results than the hyperparameter optimized for the task itself! DQN, on the other hand, demonstrates surprising robustness—cross-task hyperparameter transfers have minimal impact on its peak performance. This big variability can be due to the complexities in hyperparameter selection strategies (Patterson et al., 2024), including the fact that I was able to afford only three seeds per configuration, which is certainly not enough for an accurate estimate. This is another common practice in the field that can be quite detrimental. As shown in Figure 4.11, the results underscore a critical insight: no single hyperparameter setting is robust across all tasks.

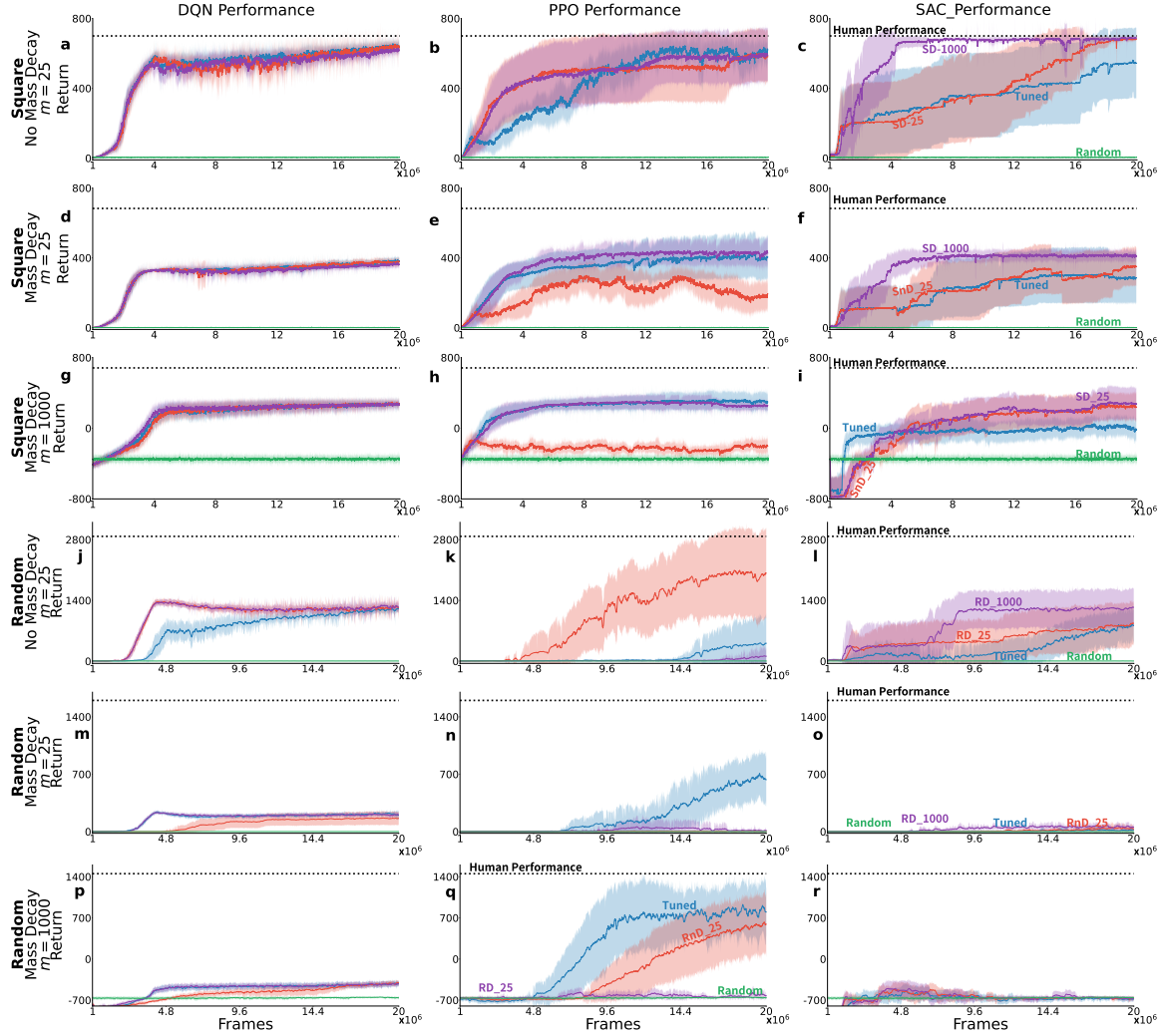


Figure 4.11: Performance of baseline agents (DQN, PPO, SAC) on six mini-games using cross-evaluated hyperparameters. The mini-games are categorized into two: the top three rows correspond to the **square path** group (Square-path Pellets collection tasks), and the bottom three rows to the **random** group (randomly regenerated pellet tasks). For each mini-game, agents are evaluated using their own tuned hyperparameters and those optimized for the other two tasks within the same group. Each row shows the performance of a baseline algorithm across three hyperparameter configurations. For example, panels (a), (b), and (c) show DQN, PPO, and SAC in mini-game 1 under different hyperparameter settings. The naming convention follows: SD-1000 denotes a square path setting with mass decay and an initial mass of 1000, while SnD-25 indicates no decay and an initial mass of 25. Tuned is considered for the best hyperparameter on a particular mini-game. The same notation applies for the Random group, using R instead of S .

It is also worth stepping back to question how I select hyperparameters. In truly

continual RL, the tuning strategies commonly used in RL are impractical (Mesbahi et al., 2024). Beyond their variability, hyperparameters are often tuned for a fixed duration; for example, those for 100k frames in Atari 2600 games differ significantly from those for 200M. How can I tune for a potentially unbounded timeframe? This general limitation of machine learning algorithms becomes starkly apparent in long experiments. As I have shown, RL algorithms are susceptible to hyperparameter selection.

4.5 How is AgarCL an Effective Continual Reinforcement Learning Testbed?

After demonstrating the difficulty of individual game components through the mini-games, a natural question arises: what happens when these challenges—along with additional features—are presented together in a non-episodic setting?

The FULL-GAME is the evaluation platform for continual reinforcement learning that I am introducing in this thesis. It represents a challenging environment that incorporates all the difficulties discussed in the previous sections and further complexities.

4.5.1 The Full-Game Evaluation

In my primary full-game evaluation, I use a 350×350 arena populated with 500 pellets, 10 viruses, and 8 heuristic bots. When an agent or bot is consumed, it respawns immediately with an initial mass of 25, while all other entities in the environment retain their current state. This setup ensures that all of the agent’s cells operate in a persistent, non-episodic world, where the dynamics of the testbed evolve continually over time.

Every 600 ticks (150 ticks if I use frame skipping of 4), pellets and viruses that have been consumed or destroyed are regenerated with a uniform probability across the arena. This maintains fixed totals of 500 and 10, respectively. Mass decay is applied

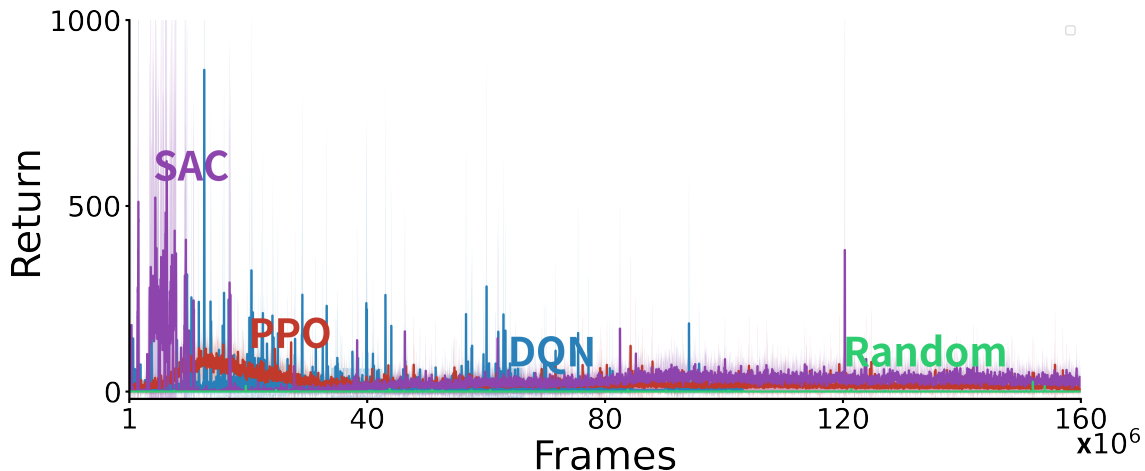


Figure 4.12: Performance of established deep RL algorithms in the full game of **AgarCL**. Each curve is the average over 10 seeds, with a moving average computed over a window of 100 steps.

at intervals of 60 ticks.

I evaluated DQN, PPO, and SAC learning over 160 million frames in the environment, averaging over 10 independent runs. Running SAC, for example, takes more than seven days in this setting. Given the challenges of tuning hyperparameters in continual RL, I did not tune hyperparameters for this specific setting. Instead, I used the hyperparameters I used in the continual mini-Game 4, which was the most similar mini-game to this setting in which I saw some success.

Given the results in the previous sections, it is not surprising that the algorithms I considered here were unable to learn a minimally effective policy in **AgarCL** (see Fig. 4.12). This further emphasizes the challenge I am posing.

I want to emphasize again that hyperparameter tuning is unfeasible in this setting, and success on simplified tasks rarely generalizes. For instance, despite starting from a working configuration, I observed NaNs after 160 million frames in the **FULL-GAME**. A key open question is how to ensure hyperparameters for which the agent’s performance remains stable over extended runs. All experiments were run synchronously with a single actor, and I find it unsatisfying if success depends on extensive tuning across billions of frames.

4.5.2 Positioning AgarCL as a Next-Generation Continual RL Testbed

So far, I have highlighted the challenges the FULL-GAME setting poses—even in its simplified mini-game variants. Our baselines failed to learn in the FULL-GAME. To mitigate this issue, I tuned the hyperparameters of the FULL-GAME in which regenerating pellets and viruses occur more frequently: specifically, pellets and viruses are regenerated every 120 ticks instead of 600.

The subsequent subsections focus on investigating two central research questions:

- Can PPO effectively learn when the environment is more densely populated with pellets?
- Does continually updating the agent lead to better long-term performance than freezing the policy at a fixed checkpoint?

Effect of Pellet Density on Learning

The main experiment (Figure 4.12) in the FULL-GAME setting revealed it to be a highly challenging problem that standard deep RL baselines struggle to solve effectively. To reduce complexity, I designed a simplified variant of the environment in which the number of bots were decreased from 8 to 4, while maintaining 10 viruses. Additionally, the regeneration interval for both pellets and viruses was shortened from 600 to 120 ticks. To further ease the learning process, I introduced multiple configurations varying the number of pellets. Each experiment was conducted with 86 million training steps. This setting allowed PPO agents to grow more rapidly and gain a competitive advantage early on, increasing their chances of successfully outcompeting the other bots. Across all experiments, PPO consistently demonstrated the most robust performance in this environment. As a result, I used PPO for the rest of the experiments in my thesis.

As illustrated in Figure 4.13—and summarized in Table 4.2—increasing the number of pellets significantly accelerates early-stage growth for the agent, allowing it to accumulate mass more rapidly. This trend is especially evident in configurations with higher pellet densities (e.g., P1024 and P400). However, beyond approximately 50 million environment steps, performance plateaus or even degrades across all configurations, suggesting a failure to sustain learning over time.

From this observation, two insights can be drawn: (1) while a higher pellet density facilitates faster initial growth, it does not appear to encourage strategic behaviors such as attacking or outmaneuvering other bots; and (2) the drop in performance may indicate a neural network problem. Further investigation is needed. This aligns with known challenges in continual RL, where agents struggle to adapt once they converge prematurely: learned to eat as many pellets as the agent can consume and evade bots.

It is worth noting that the curve corresponding to P1024, while initially promising, also succumbs to the same drop pattern—highlighting that a richer environment alone is not sufficient to ensure long-term learning progress. Further investigation is required to confirm whether architectural or algorithmic modifications could mitigate this degradation.

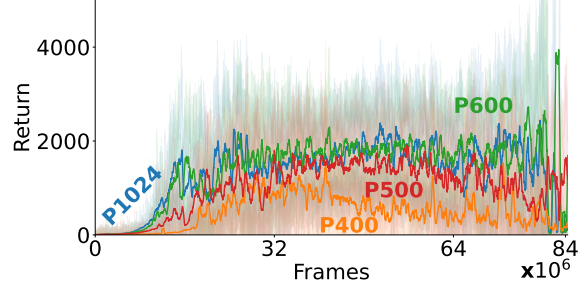


Figure 4.13: PPO performance across varying pellet densities within a simplified version of the FULL-GAME environment in AgarCL. All experiments use 10 viruses and 4 bots; the only variable is the number of pellets. For example, the curve labeled P1024 corresponds to 1024 pellets. Results are averaged over 10 random seeds, and each curve is smoothed using a moving average with a window of 1000 steps.

Table 4.2: PPO performance across difficulty levels in a simplified FULL-GAME environment. Each result is averaged over 10 random seeds.

Difficulty	Return (Mean $\pm$ Std)
Easy	403.13 (± 2.26)
Medium	63.60 (± 0.40)
Medium-2	170.16 (± 1.83)
Difficult	1582.56 (± 5.76)

Continual Learning vs. Fixed Policy Checkpoints

Now, I want to show how interesting my environment is in the context of continual RL from a different perspective. In simple words, can we claim that learning continually is better than having premature fixed-policy checkpoints?

To answer this question, I investigated whether a continually learning agent can outperform agents that rely on fixed policies saved at earlier training checkpoints. Specifically, I compared the performance of a PPO agent trained continually to agents initialized with frozen policies at 32 million and 64 million training steps. All agents were evaluated in the same setting: a simplified version of the FULL-GAME environment featuring 10 viruses, 8 bots, and 1024 pellets, with a regeneration interval of 120 ticks for both pellets and viruses. The agent learned for 120 million frames.

As shown in Figure 4.14—and summarized in Table 4.3—the performance of frozen-policy agents collapses after a certain point. Although these fixed-policy agents initially perform competitively, benefiting from prior training, their returns eventually stagnate or decline. This degradation highlights a key limitation: fixed policies cannot adapt to changing environmental conditions or evolving opponent strategies once deployed. Interestingly, even PPO exhibits collapse at fixed checkpoints

for a different reason. In this case, the decline may stem from a neural network phenomenon, potentially indicating a loss of plasticity. However, further investigation

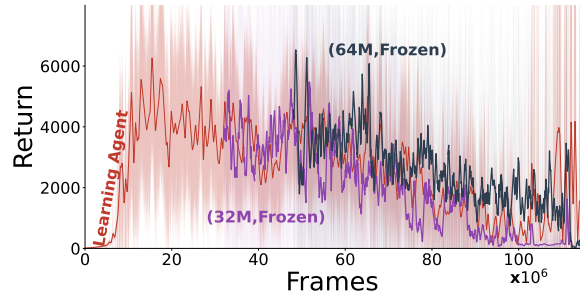


Figure 4.14: Performance comparison between a continually learning PPO agent and fixed-policy agents initialized from checkpoints at 32M and 64M steps, within the simplified FULL-GAME environment of **AgarCL**. The environment includes 10 viruses, 8 bots, and 1024 pellets, with shortened regeneration intervals (120 ticks) for both pellets and viruses. Results are averaged over 10 random seeds, and smoothed using a moving average with a window of 1000 steps.

is needed.

These findings support the argument that AgarCL has strong potential as a continual RL benchmark. However, we could not conclusively demonstrate that AgarCL embodies the “big world” hypothesis. While fixed-policy agents at different checkpoints collapsed sharply, the learning

agents did not consistently perform well. This underscores the need for more robust algorithms that sustain learning over time—an essential direction for future work.

Table 4.3: PPO Performance comparison at key checkpoints versus the learning agent. Each result is the mean return of the last 100 episodes (10000 steps).

Agent	Return (Mean $\pm$ Std)
32M Checkpoint	2.70 (± 0.00)
64M Checkpoint	251.11 (± 0.45)
Learning Agent	55.35 (± 0.82)

Chapter 5

Conclusion and Future Work

In this thesis, I introduced **AgarCL**, a continual reinforcement learning (RL) evaluation platform with partial observability, non-stationarity, and open-ended interaction. Unlike prior continual RL testbeds, **AgarCL** avoids artificial resets or abrupt task switches, allowing agents to accumulate experience and continually adapt. Through extensive experiments with DQN, PPO, and SAC, I demonstrated: (1) the difficulty of the game from isolated challenges (e.g., pellet-collection, fixed-policy opponents) to integrated difficulty of the full game, (2) the limitations of deep RL methods, and (3) the pronounced sensitivity of these algorithms to hyperparameter choices in a continual learning setting. By open-sourcing **AgarCL**, I aim to empower the community to develop and evaluate new continual learning algorithms under persistent dynamics.

My work has some limitations. First, the computational cost of running simulations and neural network training in **AgarCL** is substantial: in Appendix A, I reported that each experiment requires on the order of hundreds of GPU hours due to hyperparameter tuning in each mini-game. Second, this line of work would also benefit from a faster simulator, maybe entirely written in Jax. Third, I observed that standard hyperparameter sweeps are especially fragile in non-stationary environments; future work should investigate automated hyperparameter adaptation and metalearning techniques to maintain robust performance over long horizons.

Bibliography

- Abbas, Zaheer, Rosie Zhao, Joseph Modayil, Adam White, and Marlos C. Machado (2023). “Loss of Plasticity in Continual Deep Reinforcement Learning”. In: *Conference on Lifelong Learning Agents*.
- Abel, David, Andre Barreto, Benjamin Van Roy, Doina Precup, Hado Philip van Hasselt, and Satinder Singh (2023). “A Definition of Continual Reinforcement Learning”. In: *International Conference on Neural Information Processing Systems*.
- Anand, Nishanth and Doina Precup (2023). “Prediction and Control in Continual Reinforcement Learning”. In: *International Conference on Neural Information Processing Systems*.
- Ansó, Nil Stolt, Anton Orell Wiehe, Madalina M. Drugan, and Marco A. Wiering (2019). “Deep Reinforcement Learning for Pellet Eating in Agar.io”. In: *International Conference on Agents and Artificial Intelligence*.
- Ba, Jimmy Lei, Jamie Ryan Kiros, and Geoffrey E. Hinton (2016). “Layer Normalization”. In: *arXiv*. eprint: 1607.06450.
- Beattie, Charlie, Joel Z. Leibo, Denis Teplyashin, Tom Ward, Marcus Wainwright, Heinrich Küttler, Andrew Lefrancq, Simon Green, Víctor Valdés, Amir Sadik, Julian Schrittwieser, Keith Anderson, Sarah York, Max Cant, Adam Cain, Adrian Bolton, Stephen Gaffney, Helen King, Demis Hassabis, Shane Legg, and Stig Petersen (2016). “DeepMind Lab”. In: *arXiv* /abs/1612.03801.
- Bellemare, Marc G., Yavar Naddaf, Joel Veness, and Michael Bowling (2013). “The Arcade Learning Environment: An Evaluation Platform for General Agents”. In: *Journal of Artificial Intelligence Research* 47, pp. 253–279.
- Blackwell, David (1962). “Discrete Dynamic Programming”. In: *The Annals of Mathematical Statistics* 33.2, pp. 719–726.
- Brockman, Greg, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba (2016). “OpenAI Gym”. In: *arXiv* abs/1606.01540.
- Chung, Junyoung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio (2014). “Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling”. In: *arXiv* abs/1412.3555.
- Chung, Wesley, Lynn Cherif, David Meger, and Doina Precup (2025). “Parseval Regularization for Continual Reinforcement Learning”. In: *International Conference on Neural Information Processing Systems*.
- Dabney, Will, Georg Ostrovski, and Andre Barreto (2021). “Temporally-Extended ϵ -Greedy Exploration”. In: *International Conference on Learning Representations*.

- Deaton, Jonathan (2018). *AgarLE: Agar.io OpenAI Gym Learning Environment*. <https://github.com/jondeaton/AgarLE>. Accessed: 2025-04-09.
- Dohare, Shibhansh, J. Fernando Hernandez-Garcia, Qingfeng Lan, Parash Rahman, A. Rupam Mahmood, and Richard S. Sutton (2024). “Loss of Plasticity in Deep Continual Learning”. In: *Nature* 632.8026, pp. 768–774.
- Fujita, Yasuhiro, Prabhat Nagarajan, Toshiki Kataoka, and Takahiro Ishikawa (2021). “ChainerRL: A Deep Reinforcement Learning Library”. In: *Journal of Machine Learning Research* 22.77, pp. 1–14.
- Haarnoja, Tuomas, Aurick Zhou, Pieter Abbeel, and Sergey Levine (2018). “Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor”. In: *International Conference on Machine Learning*.
- Henderson, Peter, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger (2018). “Deep Reinforcement Learning That Matters”. In: *AAAI Conference on Artificial Intelligence (AAAI)*.
- Janjua, Muhammad Kamran, Haseeb Shah, Martha White, Erfan Miah, Marlos C. Machado, and Adam White (2024). “GVFs in the Real World: Making Predictions Online for Water Treatment”. In: *Machine Learning* 113.8, pp. 5151–5181.
- Javed, Khurram and Richard S. Sutton (2024). “The Big World Hypothesis and its Ramifications for Artificial Intelligence”. In: *Finding the Frame Workshop at Reinforcement Learning Conference*.
- Kaelbling, Leslie Pack, Michael L. Littman, and Anthony R. Cassandra (1998). “Planning and Acting in Partially Observable Stochastic Domains”. In: *Artificial Intelligence* 101.1-2, pp. 99–134.
- Kaplanis, Christos, Murray Shanahan, and Claudia Clopath (2019). “Policy Consolidation for Continual Reinforcement Learning”. In: *International Conference on Machine Learning*.
- Khetarpal, Khimya, Matthew Riemer, Irina Rish, and Doina Precup (2022). “Towards Continual Reinforcement Learning: A Review and Perspectives”. In: *Journal of Artificial Intelligence Research* 75, pp. 1401–1476.
- Kirkpatrick, James, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A. Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, Demis Hassabis, Claudia Clopath, Dharmashan Kumaran, and Raia Hadsell (2017). “Overcoming Catastrophic Forgetting in Neural Networks”. In: *Proceedings of the National Academy of Sciences* 114.13, pp. 3521–3526.
- Kumar, Saurabh, Henrik Marklund, Ashish Rao, Yifan Zhu, Hong Jun Jeon, Yueyang Liu, and Benjamin Van Roy (2025). “Continual Learning as Computationally Constrained Reinforcement Learning”. In: *arXiv*. eprint: 2307.04345.
- Lecun, Yann, Leon Bottou, Genevieve Orr, and Klaus-Robert Muller (1998). “Efficient BackProp”. In: *Neural Networks: Tricks of the Trade, This Book is an Outgrowth of a 1996 NIPS Workshop*. Berlin, Heidelberg: Springer-Verlag, pp. 9–50.
- Lyle, Clare, Zeyu Zheng, Evgenii Nikishin, Bernardo Avila Pires, Razvan Pascanu, and Will Dabney (2023). “Understanding Plasticity in Neural Networks”. In: *International Conference on Machine Learning*.
- Machado, Marlos C., Marc G. Bellemare, Erik Talvitie, Joel Veness, Matthew J. Hausknecht, and Michael Bowling (2018). “Revisiting the Arcade Learning En-

- vironment: Evaluation Protocols and Open Problems for General Agents”. In: *Journal of Artificial Intelligence Research* 61, pp. 523–562.
- Mesbahi, Golnaz, Olya Mastikhina, Parham Mohammad Panahi, Martha White, and Adam White (2024). “K-percent Evaluation for Lifelong RL”. In: *arXiv abs/2404.02113*.
- Mnih, Volodymyr, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu (2016). *Asynchronous Methods for Deep Reinforcement Learning*. arXiv: 1602.01783.
- Mnih, Volodymyr, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin A. Riedmiller, Andreas Fidfjeld, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharmashan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis (2015). “Human-level Control through Deep Reinforcement Learning”. In: *Nature* 518.(7540), pp. 529–533.
- Mohamed, Mohamed A., Kateryna Nekhomiaz, Vedant Vyas, Marcos M. Jose, Andrew Patterson, and Marlos C. Machado (2025). “The Cell Must Go On: Agar.io for Continual Reinforcement Learning”. In: *arXiv*. arXiv: 2505.18347.
- Nikishin, Evgenii, Junhyuk Oh, Georg Ostrovski, Lyle Clare, Razvan Pascanu, Will Dabney, and André Barreto (2023). “Deep Reinforcement Learning with Plasticity Injection”. In: *International Conference on Neural Information Processing Systems*.
- Patterson, Andrew, Samuel Neumann, Martha White, and Adam White (2024). “Empirical Design in Reinforcement Learning”. In: *Journal of Machine Learning Research* 25.318, pp. 1–63.
- Platanios, Emmanouil Antonios, Abulhair Saparov, and Tom M. Mitchell (2020). “Jelly Bean World: A Testbed for Never-Ending Learning”. In: *International Conference on Learning Representations*.
- Powers, Sam, Eliot Xing, Eric Kolve, Roozbeh Mottaghi, and Abhinav Gupta (2022). “CORA: Benchmarks, Baselines, and Metrics as A platform for Continual Reinforcement Learning Agents”. In: *Conference on Lifelong Learning Agents*.
- Puterman, Martin L. (2014). *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons.
- Ring, Mark Bishop (1994). “Continual learning in reinforcement environments”. UMI Order No. GAX95-06083. PhD thesis.
- Rolnick, David, Arun Ahuja, Jonathan Schwarz, Timothy Lillicrap, and Gregory Wayne (2019). “Experience Replay for Continual Learning”. In: *International Conference on Neural Information Processing Systems*.
- Schulman, John, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz (2015). “Trust Region Policy Optimization”. In: *International Conference on Machine Learning*.
- Schulman, John, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov (2017). “Proximal Policy Optimization Algorithms”. In: *arXiv abs/1707.06347*. arXiv: 1707.06347.
- Sutton, Richard S. and Andrew G. Barto (1998). *Reinforcement learning: An introduction*. MIT press Cambridge.

- Sutton, Richard S., Anna Koop, and David Silver (2007). “On the Role of Tracking in Stationary Environments”. In: *International Conference on Machine Learning*.
- Tassa, Yuval, Yotam Doron, Alistair Muldal, Tom Erez, Yazhe Li, Diego de Las Casas, David Budden, Abbas Abdolmaleki, Josh Merel, Andrew Lefrancq, Timothy P. Lillicrap, and Martin A. Riedmiller (2018). “DeepMind Control Suite”. In: *arXiv abs/1801.00690*.
- Thrun, Sebastian (1998). “Lifelong Learning Algorithms”. In: *Learning to Learn*. Springer US, pp. 181–209.
- Todorov, Emanuel, Tom Erez, and Yuval Tassa (2012). “MuJoCo: A Physics Engine for Model-based Control”. In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*.
- Tomilin, Tristan, Meng Fang, Yudi Zhang, and Mykola Pechenizkiy (2023). “COOM: A Game Benchmark for Continual Reinforcement Learning”. In: *International Conference on Neural Information Processing Systems*.
- Watkins, Christopher J. C. H. and Peter Dayan (1992). “Q-Learning”. In: *Machine Learning* 8.3, pp. 279–292. DOI: 10.1007/BF00992698.
- Wiehe, Anton Orell, Nil Stolt Ansó, Madalina M. Drugan, and Marco A. Wiering (2018). “Sampled Policy Gradient for Learning to Play the Game Agar.io”. In: *arXiv abs/1809.05763*.
- Zhang, Ming, Shenghan Zhang, Zhenjie Yang, Lekai Chen, Jinliang Zheng, Chao Yang, Chuming Li, Hang Zhou, Yazhe Niu, and Yu Liu (2023a). “GoBigger: A scalable Platform for Cooperative-Competitive Multi-Agent Interactive Simulation”. In: *International Conference on Learning Representations*.
- Zhang, Tiantian, Kevin Zehua Shen, Zichuan Lin, Bo Yuan, Xueqian Wang, Xiu Li, and Deheng Ye (2023b). “Replay-enhanced Continual Reinforcement Learning”. In: *Transactions on Machine Learning Research*.

Appendix A: Computational Resources

To thoroughly evaluate my approach, I conducted extensive hyperparameter tuning through grid search, running more than a thousand individual experiments to identify high-performing configurations for each task. Although this ensured strong performance, it came at a significant computational cost, a limitation worth noting for future work and reproducibility.

Table A.1: Compute resource utilization across four clusters.

Cluster	Elapsed (days)	Total CPU Usage (years)	Total GPU Usage (years)	Memory (32 GB units, years)
Cluster 1	3435.0	20.12	9.40	301.2
Cluster 2	1162.0	10.20	3.16	96.0
Cluster 3	162.9	1.65	0.44	14.3
Cluster 4	884.0	7.76	2.42	77.6
Total	5644.0	39.73	15.42	489.1

Cluster hardware details:

- **Cluster 1:** Intel Xeon Gold 6448Y CPU; NVIDIA L40s (48GB) GPU.
- **Cluster 2:** Intel E5-2683 v4, Silver 4216, Platinum 8160F/8260 CPUs; NVIDIA P100 (12 GB), V100 (32 GB) GPUs.
- **Cluster 3:** Intel Xeon Gold 6238/6248, Silver 4110, AMD EPYC 7742/7713 CPUs; NVIDIA P100 (12 GB), V100 (16/32 GB), T4 (16 GB), A100, RTX A5000 GPUs.
- **Cluster 4:** Intel Xeon Gold 6148 CPU; NVIDIA V100SXM2 (16 GB) GPU.

Appendix B: Tuning Details

Hyperparameter tuning is especially critical in reinforcement learning due to the inherent instability and sensitivity of RL algorithms. Unlike supervised learning, RL involves exploration-exploitation trade-offs, nonstationary data distributions, and delayed rewards, all of which can magnify the effects of poorly chosen hyperparameters. Parameters such as step size, discount factor, exploration noise, entropy regularization (in policy gradient methods), and update frequencies can drastically influence the learning dynamics and final policy performance. Improper tuning can lead to divergence, suboptimal policies, or excessive variance in performance. This hyperparameter tuning is exceedingly challenging in face of computational constraints, as I discussed in Chapter 4.4.

B.1 DQN Tuning

For tuning DQN, I swept over the hyperparameters listed in Table B.1. As discussed in 2.2.1, the agent’s and target networks consist of three convolutional layers, each followed by a ReLU activation and a Layer Normalization layer (Ba et al., 2016). In preliminary experiments, I found that omitting Layer Normalization prevented the agent from learning. All weights were initialized using a LeCun Normal Initialization (Lecun et al., 1998). Each hyperparameter combination was trained using three different random seeds. After identifying the best configuration of the hyperparameter, I conducted ten additional independent runs using the best-performing configuration (see Table B.2).

Table B.1: Values of hyperparameters that I swept over when tuning DQN.

Hyperparameter	Values / Settings
Step size (<code>--lr</code>)	10^{-5} , $3 \cdot 10^{-5}$, 10^{-4} , $3 \cdot 10^{-4}$
Batch Accumulator (<code>--batch_accumulator</code>)	"sum", "mean"
Soft Update Coefficient (<code>--tau</code>)	10^{-2} , 5^{-3}
Batch Size (<code>--batch-size</code>)	32, 64
Replay Buffer Size (<code>--replay-buffer</code>)	10^5
Number of Epochs (<code>--epochs</code>)	1
Target Network Update Interval (<code>--update_interval</code>)	4
Gamma ($-\gamma$)	0.99
Exploration algorithm	ϵ -Greedy, ϵ z-Greedy

Table B.2: Best hyperparameters for DQN on each minigame.

Category	Mini-game	Hyperparameters				Exploration Algorithm
		Step Size	Batch Accumulator	Soft Update Coefficient	Batch Size	
Episodic	①	10^{-4}	mean	$5 \cdot 10^{-5}$	64	ϵ -Greedy
	②	$3 \cdot 10^{-4}$	sum	$5 \cdot 10^{-5}$	32	ϵ -Greedy
	③	10^{-4}	mean	$5 \cdot 10^{-5}$	32	ϵ -Greedy
	④	$3 \cdot 10^{-4}$	sum	10^{-2}	32	ϵ -Greedy
	⑤	10^{-5}	mean	10^{-2}	64	ϵ -Greedy
	⑥	10^{-5}	mean	10^{-2}	64	ϵ -Greedy
Continuing	①	10^{-4}	sum	$5 \cdot 10^{-5}$	32	ϵ z-Greedy
	②	$3 \cdot 10^{-4}$	sum	10^{-2}	32	ϵ z-Greedy
	③	10^{-4}	mean	$5 \cdot 10^{-5}$	32	ϵ z-Greedy
	④	10^{-4}	sum	$5 \cdot 10^{-5}$	32	ϵ z-Greedy
	⑤	10^{-4}	mean	10^{-2}	32	ϵ z-Greedy
	⑥	$3 \cdot 10^{-4}$	mean	10^{-2}	32	ϵ z-Greedy
Other Agents	⑦-Small-Sparse ¹	10^{-5}	mean	5×10^{-2}	32	ϵ -Greedy
	⑦-Normal-Dense ²	3×10^{-4}	mean	10^{-2}	64	ϵ -Greedy
	⑧-Small-Sparse ¹	10^{-5}	mean	5×10^{-2}	64	ϵ -Greedy
	⑧-Normal-Dense ²	1×10^{-5}	sum	10^{-2}	64	ϵ -Greedy
Virus	⑨	3×10^{-4}	sum	10^{-2}	128	ϵ -Greedy

¹ Limited (Sparse) arena: 200×200 with 200 randomly respawned pellets.² Normal (Dense) arena: 350×350 with 500 randomly respawned pellets.

As shown in Table B.1, the discount factor, γ , the number of epochs, the replay buffer size, and the target-network update interval were each fixed at a single value, while the remaining hyperparameters were swept. Finally, I applied the hyperparameter settings from mini-game 4 to train the full-game agent.

B.2 PPO Tuning

Hyperparameter tuning for PPO is particularly challenging due to the configuration space size: minimally covering relevant combinations requires on the order of ~ 324 individual runs per experiment. Using the same architecture described in the previous section, I employed a shared neural network to learn both actor and critic, since our preliminary studies showed that a shared network consistently outperforms separate networks. All network weights are initialized using LeCun normal initialization (Lecun et al., 1998). The policy head is a Gaussian distribution with a state-independent, learned covariance—a configuration that is standard in PPO implementations (e.g., Henderson et al., 2018).

In Table B.3, I have two reward normalization schemes in hyperparameter tuning. The first is *min-max normalization*, which linearly rescales the raw reward r_t into the range $[-1, 1]$:

$$\tilde{r}_t^{(\text{min-max})} = \frac{r_t - r_{\min}}{r_{\max} - r_{\min} + \epsilon}, \quad (\text{B.1})$$

where $\epsilon > 0$ is a small constant added for numerical stability.

The second method, *variance normalization*, uses a weighted moving average of the returns:

$$G_t = \gamma G_{t-1} + r_t, \quad (\text{B.2}) \quad \tilde{r}_t^{(\text{var-norm})} = \frac{r_t}{\sqrt{\text{Var}[G_t] + \epsilon}}, \quad \text{where} \quad (\text{B.3})$$

G_t is the smoothed return, and normalization is performed by dividing the reward by the square root of its running variance. This setup ensures that the normalized reward maintains approximately unit variance under an exponential moving average with discount factor γ . After normalization, the reward is clipped to lie within the fixed range $[-10, 10]$ to limit the effect of outliers during training:

$$\tilde{r}_t^{(\text{clipped})} = \text{clip}\left(\tilde{r}_t^{(\text{var-norm})}, -10, 10\right). \quad (\text{B.4})$$

Initially, I swept over different hyperparameter combinations for PPO in some episodic settings following Table B.3.

Table B.3: Values of hyperparameters that I swept over when tuning PPO.

Hyperparameter	Values / Settings
Reward function (<code>--reward</code>)	<code>min_max</code> , <code>variance_norm</code>
Step size (<code>--lr</code>)	10^{-5} , 3×10^{-5} , 3×10^{-4} , 10^{-4}
Epochs (<code>--epochs</code>)	10, 15, 20
Max gradient norm (<code>--max-grad-norm</code>)	0.5, 0.7, 0.9
Entropy coefficient (<code>--entropy-coef</code>)	0.05, 0.01, 0.1, 0.5
Clipping epsilon (<code>--clip-eps</code>)	0.2, 0.4
Discount factor (<code>--gamma</code>)	0.995
GAE parameter (<code>--lambda</code>)	0.97
Value-function coefficient (<code>--value-func-coef</code>)	0.9, 0.5
Batch size (<code>--batch-size</code>)	64
Update interval (<code>--update-interval</code>)	1024, 2048, 5000

Through these experiments, I observed that a value function coefficient of 0.9, an update interval of 5000, step sizes of either 10^{-5} or 3×10^{-5} , and epochs set to either 10 or 15 yielded consistently strong performance. Based on sensitivity analyses across most minigame tasks, I narrowed down the range of hyperparameters I swept over for the other tasks. The smaller set of hyperparameters I swept over is shown in Table B.4. Table B.5 shows the best-performing hyperparameters that I used in each task at the end. Finally, I applied the hyperparameter settings from MINI-GAME 4 to train the full-game agent.

Table B.4: Updated PPO hyperparameters used for tuning.

Hyperparameter	Values / Settings
Reward function (<code>--reward</code>)	<code>min_max</code> , <code>variance_norm</code>
Step size (<code>--lr</code>)	10^{-5} , 3×10^{-5}
Epochs (<code>--epochs</code>)	10, 15
Max gradient norm (<code>--max-grad-norm</code>)	0.5, 0.7, 0.9
Entropy coefficient (<code>--entropy-coef</code>)	0.05, 0.01, 0.1, 0.5
Clipping epsilon (<code>--clip-eps</code>)	0.2, 0.4
Discount factor (<code>--gamma</code>)	0.995
GAE parameter (<code>--lambda</code>)	0.97
Value-function coefficient (<code>--value-func-coef</code>)	0.9
Batch size (<code>--batch-size</code>)	64
Update interval (<code>--update-interval</code>)	5000

Table B.5: Best hyperparameters for PPO on each minigame.

Category	Mini-game	Hyperparameters					
		Reward Function	Step Size	Epochs	Max Grad Norm	Entropy Coef	Clip Eps
Episodic	①	Min Max	10^{-4}	15	0.7	0.01	0.4
	②	Min Max	10^{-5}	10	0.7	0.01	0.2
	③	Min Max	10^{-5}	10	0.9	0.01	0.2
	④	Min Max	10^{-5}	15	0.9	0.05	0.4
	⑤	Min Max	10^{-4}	10	0.7	0.01	0.4
	⑥	Min Max	3×10^{-5}	10	0.5	0.05	0.2
Continuing	①	Min Max	10^{-4}	4	0.5	0.005	0.2
	②	Variance Norm	10^{-5}	10	0.5	0.05	0.4
	③	Min Max	10^{-4}	4	0.5	0.01	0.3
	④	Variance Norm	10^{-5}	10	0.5	0.1	0.4
	⑤	Min Max	3×10^{-5}	10	0.7	0.05	0.4
	⑥	Min Max	3×10^{-4}	15	0.7	0.05	0.2
Other Agents	⑦-Small-Sparse ¹	Min Max	3×10^{-5}	10	0.5	0.05	0.4
	⑦-Normal-Dense ²	Variance Norm	10^{-5}	10	0.9	0.1	0.4
	⑧-Small-Sparse ¹	Min Max	3×10^{-5}	10	0.5	0.05	0.2
	⑧-Normal-Dense ²	Min-Max	3×10^{-5}	10	0.7	0.05	0.2
Virus	⑨	Min Max	10^{-5}	10	0.7	0.01	0.4

¹ Uses a 200×200 arena with 200 randomly respawned pellets.² Uses a 350×350 arena with 500 randomly respawned pellets.

B.3 SAC Tuning

I swept over the hyperparameter combinations in Table B.6. In SAC, I employed three separate networks. All network weights were initialized using LeCun normal initialization (Lecun et al., 1998). Table B.7 summarizes the best-performing hyperparameters across our tasks. Also, I applied the hyperparameter settings from MINI-GAME 4 to train the full-game agent.

Table B.6: Values of hyperparameters that I swept over when tuning SAC.

Hyperparameter	Values / Settings
Step size (<code>--lr</code>)	10^{-4} , 3×10^{-5} , 10^{-5}
Reward function (<code>--reward</code>)	<code>min_max</code> , <code>variance_norm</code>
Replay buffer size (<code>--replay-buffer</code>)	10^5
Soft update coefficient (<code>--tau</code>)	10^{-2} , 5×10^{-3} , 10^{-3}
Max gradient norm (<code>--max-grad-norm</code>)	0.5, 0.7, 0.9
Temperature step size (<code>--temperature-lr</code>)	10^{-4} , 3×10^{-4}
Update interval (<code>--update-interval</code>)	4
Batch size (<code>--batch-size</code>)	64
Discount factor (<code>-gamma</code>)	0.99

Table B.7: Best hyperparameters for SAC on each minigame.

Category	Mini-game	Hyperparameters				
		Reward Function	Step Size	Soft Update Coefficient	Temperature LR	Max Grad Norm
Episodic	①	Min Max	3×10^{-5}	0.001	10^{-4}	0.7
	②	Min Max	3×10^{-5}	0.001	10^{-4}	0.7
	③	Min Max	10^{-4}	0.01	10^{-4}	0.7
	④	Min Max	10^{-5}	0.001	10^{-4}	0.7
	⑤	Min Max	10^{-5}	0.001	10^{-4}	0.7
	⑥	Min Max	10^{-4}	0.001	10^{-4}	0.5
Continuing	①	Variance Norm	3×10^{-5}	0.001	10^{-4}	0.5
	②	Variance Norm	3×10^{-5}	0.001	10^{-4}	0.7
	③	Variance Norm	3×10^{-5}	0.005	10^{-4}	0.9
	④	Variance Norm	10^{-5}	0.01	10^{-4}	0.5
	⑤	Variance Norm	10^{-5}	0.001	10^{-4}	0.7
	⑥	Variance Norm	3×10^{-5}	0.005	10^{-5}	0.9
Other Agents	⑦-Small-Sparse ¹	Variance Norm	10^{-4}	0.01	10^{-4}	0.9
	⑦-Normal-Dense ²	Variance Norm	10^{-4}	0.01	10^{-4}	0.7
	⑧-Small-Sparse ¹	Variance Norm	10^{-5}	0.001	10^{-4}	0.9
	⑧-Normal-Dense ²	Variance Norm	10^{-5}	0.005	10^{-4}	0.9
Virus	⑨	Min Max	10^{-5}	0.01	10^{-4}	0.7

¹ Sparse setting: 200×200 arena with 200 pellets.² Dense setting: 350×350 arena with 500 pellets.

Appendix C: Empirical Results on Different Tasks

Table C.1 summarizes the results of the experiments conducted in this paper. Each reported value represents the final performance, calculated as the average reward over the last 100 evaluation episodes.

Table C.1: Performance across **AgarCL** minigames averaged over 10 independent runs. Numbers between parentheses denote the standard deviation across these 10 runs. Results are averaged over the last 100 data points: episodes in episodic tasks and 100 steps in continual tasks.

Category	Mini-game	Scenarios	DQN	PPO	SAC	Human	Random
Pellet Collection (Episodic)	1	—	642 (14.2)	605 (25.3)	546 (6.0)	700 (0)	4.81 (1.3)
	2	—	382 (9.1)	398 (12.9)	285 (9.8)	682 (0)	0.81 (0.7)
	3	—	271 (23.1)	298 (28.8)	−16 (30.3)	612 (0)	−356 (22.7)
	4	—	1189 (44.2)	391 (36.3)	790 (98.6)	2876 (0)	4 (1.3)
	5	—	214 (13.2)	650 (58.9)	16 (8.4)	1600 (0)	0.17 (0.16)
	6	—	−426 (20.0)	812 (118.3)	−681 (12.8)	1452 (0)	−657 (6.9)
Pellet Collection (Continual)	1	—	619 (7.9)	335 (3.0)	419 (0.2)	700 (0)	0.01 (0.1)
	2	—	0 (0)	0 (0)	2 (1.7)	682 (0)	0 (0)
	3	—	−975 (0)	−975 (0)	−975 (0)	−480 (0)	−975 (0)
	4	Sparse ¹	—	22463 (48.9)	—	—	—
		Dense ¹	21402 (762.8)	21970 (277)	21997 (595.6)	23000 (0)	112.2 (12.5)
	5	Sparse	—	144 (10.8)	—	—	—
		Dense	132 (9.3)	4018 (126.8)	0.012 (0.9)	7215 (0)	0.05 (0.09)
	6	Sparse	—	−618.3 (1.4)	—	—	—
		Dense	−828 (10.5)	5064.4 (113.6)	−968 (0.01)	7215 (0)	−973 (1.1)
Other Agent (Episodic)	7—Small, Sparse ²	—	157 (0.7)	253 (35.6)	31 (0.2)	1980 (0)	0.015 (0.04)
	7—Large, Dense ²	—	146 (48.4)	429.3 (136.4)	25 (14.2)	2385 (0)	0.1 (0.16)
	8—Small, Sparse	—	189 (6.03)	267 (26)	114 (18.8)	1800 (0.015)	0.04 (0.04)
	8—Large, Dense	—	191 (14.56)	1037 (55.2)	69 (8.9)	2035 (0.015)	0.04 (0.04)
Virus (Episodic)	9	—	−36 (4.23)	−46 (6.45)	−5 (0.1)	870 (20.5)	−2319 (8.94)
Full Game (Grand Arena)	—	—	4 (3.1)	8 (8.3)	22 (11.1)	—	0.06 (0.18)

¹ In continuing pellet-collection tasks, “Sparse” refers to an arena with 250 pellets. “Dense” uses 500 pellets.

² “Small” arena: 200×200 with 200 pellets. “Large” arena: 350×350 with 500 pellets.