TDK-thesis

Magyar Gergely

Takács Tamás

Scalable Distributed Reinforcement Learning in Multi-Agent Environments

EÖTVÖS LORÁND UNIVERSITY

FACULTY OF INFORMATICS

DEPT. OF ARTIFICIAL INTELLIGENCE

Authors:

Magyar Gergely
Computer Science MSc
4. grade

Takács Tamás
Computer Science MSc
4. grade

Supervisor:

Gulyás László
Associate Professor

Budapest, 2024

Contents

Abstract

Novel reinforcement learning algorithms often face scalability and compatibility issues in multi-agent environments, as they are primarily benchmarked and optimized for single-agent settings. The lack of a standardized method for adapting these algorithms to multi-agent contexts restricts their broader applicability. In multi-agent environments where the number of controllable entities can rapidly change over a short timeframe and can scale up to massive numbers, making the utilization of both monolithic and distributed architectures challenging to orchestrate. This complexity arises from the difficulty of credit assignment, extensive memory usage, and increased computational time, leading to slow, destabilized training and suboptimal resource utilization. In our work, we explore various methods to help alleviate such inefficiencies. We worked in a massively dynamic multi-agent, open-source environment featured in a Kaggle competition called Lux AI. For this setting, we study a novel implementation of a monolithic agent control method from the literature and demonstrate that a single-brain global outlook is insufficient for managing populous environments. We propose a **hybrid architecture**, combining the strengths of monolithic and distributed approaches, which led to a **30-times reduction in model size** compared to the best-performing reinforcement learning submission to the competition while still being able to learn basic skills needed in the environment **24 times faster** in terms of training time and with **600-times fewer examples** of training data. We also introduce a technique called **trajectory separation** to massively speed up convergence, yielding a **3-times speed increase in training speed** over our initial trials. Using our approach, we achieved a **5-times reduction in training epochs** needed to reach our designated milestone compared to the baseline model provided by the competition organizers while producing comparable results with **2-times less trainable parameters**.

Chapter 1

Introduction

In recent years, Multi-agent Reinforcement Learning (MARL) has experienced a significant surge in research activity, driven by the ongoing AI advancements and substantial funding dedicated to this field (Gilbert *et al.* 2023). The global market for MARL was valued at 2.8 billion in 2022, with projections indicating a robust compound annual growth rate (CAGR) of 41.5% from 2023 to 2032. By 2032, it is anticipated to soar to a remarkable $88.7 billion (Priyadarshi *et al.* 2023). Advancements in academic research indicate an upcoming trend where both implicit and explicit Reinforcement Learning (RL) systems will display heightened efficacy. These systems are poised to find more general applications in user-facing domains, delivering more significant impacts (Kirk *et al.* 2023).

A significant portion of contemporary research dedicates efforts to exploring emergent intelligence within cooperative environments, serving as a bridge between current research and the pursuit of Artificial General Intelligence (AGI) (Lowe *et al.* 2020; J. N. Foerster *et al.* 2018). These studies center on environments that abstract real-world scenarios, characterized by interacting agents, diverse interests, varied goals, and the sharing of resources. Such environments inherently possess dynamism, displaying both evolving environmental conditions and fuzzy goal dynamics. Research has also demonstrated that competitive environments often foster heightened cooperation among agents (OpenAI *et al.* 2019; B. Baker *et al.* 2020). By utilizing multiple trajectories, the learning agent can gain experience from a wider array of interactions, with more players enhancing the diversity of trajectories encountered. In such settings, agent's objectives shift towards overcoming opposing teams, leading to continual refinement of policies until they surpass those of the opposing team. Competition inherently promotes dynamic interactions, devoid of a stable **Nash equilibrium**, thereby promoting ongoing adaptation and the perpetual search to outwit the opposing team. Nash Equilibrium is a set of strategies, one for each player, such that no player has an incentive to deviate unilaterally from their strategy, given the strategies of the other players (J. F. Nash 1950; J. Nash 1951). In a zero-sum game, the gain of one participant is exactly offset by the losses of other participants, as the total amount of resources available is fixed. Therefore, any resource gained by one player directly results in a loss for another, defining the zero-sum nature of the environment. In such settings, a Nash equilibrium implies a situation where no player would benefit from changing their strategy. However, in reality, using this framework means that one player's gain is another's loss, reflecting the directly opposing payoffs. In 1928, John von Neumann proved the **Minimax theorem**, which states that every finite, two-player, zero-sum game has an equilibrium that coincides with the Nash equilibrium (Neumann 1928; J. F. Nash 1950; J. Nash 1951; Myerson 1997; L. J. Schulman & Vazirani 2019; C. Li *et al.* 2020).

Current research efforts are further enhanced through crowd-funded or organization-sponsored MARL competitions, strategically designed to fulfill various objectives (Suarez *et al.* 2019;

Agapiou *et al.* 2023). Such collaboration yields advantageous outcomes: organizations streamline research and development expenditures, acquire access to esteemed RL specialists, and stimulate a creative momentum for participants, enabling ongoing learning and advancement within the field.

Environments such as LuxAI (Tao, Pan, *et al.* 2023a), upon which our work is centered, introduce inherent dynamism stemming from the evolving population dynamics. This dynamism facilitates intricate social scenarios, necessitating a high degree of interdependence among collaborating agents. However, it also presents a challenge regarding the compatibility of standard reinforcement learning algorithms (Wong *et al.* 2022), typically designed for single-agent or static environments. Furthermore, the calculation of rewards emerges as a critical research area, with prevailing solutions often adopting a global reward framework, sometimes called `perfectly-cooperative` games (Chen *et al.* 2023; Agapiou *et al.* 2023; J. Ye *et al.* 2023; Leroy *et al.* 2020; Suarez *et al.* 2019). This approach raises notable concerns, particularly in environments where actions that push the agents towards the predefined goal are infrequent. In such cases, the model risks reinforcing globally detrimental actions initiated by a single agent.

A MARL solution can be conceptualized along a spectrum of organizational paradigms (Piccoli 2023). At one end of the spectrum, the single-brain approach centralizes decision-making, where a collective of agents is treated as a single entity. In this model, a global observation is used to generate a unified trajectory, and rewards are distributed based on collective outcomes, which can inadvertently reinforce negative behaviors. In contrast, a hybrid model utilizes local information from all agents to generate a single trajectory but incorporates a global reward system.

Our research improves on this hybrid model by allocating rewards to individual agents or groups based on their specific contributions, thus creating a performance-based environment that operates at the individual or small group level. This approach involves dividing the environment's trajectories among agents or groups, allowing for the calculation of rewards, log probabilities, value estimates, advantages, and entropy for each unit or group. This method bootstraps the learning process by ensuring that rewards only reinforce positive behaviors early on. At the other end of the spectrum is the fully multi-agent approach, where each entity interacting with the environment is treated as an independent learning agent. This significantly complicates the learning process by creating trajectories of varied lengths, as entities may enter or exit the scene, or be destroyed. Consequently, gathering a consistent amount of experience for each entity becomes challenging, especially when some entities may only exist in the environment for a fraction of the time compared to others. This situation leads to computational inefficiencies and data sparsity. Our solution effectively addresses these challenges by offering a framework that mitigates the issues associated with both the single-brain and fully multi-agent approaches. Our research also focuses on evaluating these methodologies in dynamically changing environments and offers insights into addressing computational and emergent intelligence challenges without heavy reliance on domain-specific constraints or rigid rules.

Our **key contributions** are the following:

- We conducted a novel **benchmark study** comparing a broad array of contemporary reinforcement learning algorithms, including PPO, R-PPO, M-PPO, A2C, TRPO, DQN, QR-DQN, and ARS, to test their performance in a multi-agent environment. We evaluated the results and provided a general guideline on which algorithms to use in multi-agent systems.

- We implemented a **single-brain monolithic method as a baseline** for multi-agent reinforcement learning, employing widely used approaches from the literature such as global observations, global rewards, and a unified trajectory for all entities in the environment to assess their performance.
- We expanded the approach into a **hybrid model** by incorporating local observations and a distributed reward system, where entities in the environment received rewards commensurate with their individual or group performance. This modification demonstrated that a distributed reward system significantly enhances the learning process by preventing the reinforcement of negative behaviors early in training. Additionally, we improved on our methods by investigating potential improvements through various grouping methods, such as clustering, random grouping, and expedited training by retaining only the top N performers in a set of trajectories, or opting for a random selection based on metrics such as rewards or advantage. We also showed results with reward assignments based on a combination of individual performance and contributions from a global pool, effectively integrating individual achievement with overall group success.
- In our work, we **benchmarked our results against one of the baselines** from the Lux repository (Kai Yang 2023) and demonstrated that our solution achieves convergence over 14 times faster, completing training in less than 20 minutes, compared to the original implementation which required two days on an A100, using up to 40GB of memory. In contrast, our approach was implemented on a single V100 and used significantly less memory.
- As a result of our comprehensive exploration in the problem space, we also **provide a general framework for developing a network architecture** tailored for multi-agent applications using one of the policy gradient methods, specifically PPO. We demonstrated that proper initialization and seeding are crucial for convergence, and the necessity of employing robust regularization techniques. Our findings also indicate a threshold in network depth beyond which further deepening does not yield performance improvements. Additionally, we outline effective strategies and best practices for implementing Multi-Agent PPO-based reinforcement learning.

1.1 Outline

Given the complexity involved in fully understanding the environment, problem space, advanced applications of reinforcement learning in multi-agent systems, and the development of a specialized neural network structure for this context, we aim to provide a concise overview. This will cover how we conceptualized the project, aspects that can be simplified or skipped, and how we have presented our results. The introduction section is structured as follows:

- section 1.3 introduces basic concepts of reinforcement learning elements, such as **agent, policies, value functions, action-value functions, Bellman equations, states, rewards, discounted returns, episodes and optimality**. If the reader is familiar with these topics we recommend skipping to approximation methods (subsection 1.3.1).
- subsection 1.3.1 introduces concepts of **policy gradient methods, actor-critic methods, value and policy approximator, value-based methods, value and policiy iterations, linear approximators, neural network approximators and evolutionary algorithms**. If the reader is familiar with these topics, we recommend skipping to to the description of the environment (section 1.4).

- section 1.4 introduces the rules of the **Lux AI Competition**. If the reader comfortable with the environment, we recommend completely skipping the introductory section and going straight to the methods section (chapter 2).

The methods and results sections are closely integrated, with our findings presented in three distinct phases, as follows:

- **Single Unit Testbench**: In the methods section, we introduce the novel reinforcement learning algorithms (subsection 2.1.6) used for testing and provide an overview of the feature space, action space, and neural network architecture utilized in the benchmark (section 2.1). We also provide a detailed description of the training environment (subsection 2.1.14) and specific details regarding the evaluation processes (subsection 2.1.15). The results section details the quantitative outcomes and conclusions drawn from these tests (section 3.1), helping us to select the most suitable algorithm for further studies in the subsequent sections.
- **Monolithic Approach**: We established a baseline using novel and contemporary research to create a simple, monolithic approach for our multi-agent reinforcement learning problem (section 2.3). Utilizing the top-performing model from the single unit testbench, we expanded the scope to a multi-agent scenario (section 2.2) that includes multi-unit generation. This section proves particularly valuable as it serves as a benchmark; subsequent studies demonstrate significant enhancements over this single-brained method, highlighting the effectiveness of our research. We maintain consistent sections for training (subsection 2.3.14), evaluation (subsection 2.3.15) and results (section 3.2) within this monolithic approach.
- **Hybrid Approach**: This is the main novelty of our contribution which explores the expansion of the previously mentioned monolithic method towards a fully multi-agent approach. We incorporate features such as reward groupings, individual and group level trajectories, and calculations for rewards, values, and advantages (section 2.4). The methods section in this phase also provides architectural recommendations and insights (subsection 2.4.6). The results are comprehensive (section 3.3), including performance tests and ablation studies. For additional details on initialization techniques, seeding, and the limits of neural network architecture, we direct the reader to the discussion chapter, where we delve into less quantified findings from our research (section 4.4).

1.2 Related Works

In our review of related works, we'll explore solutions proposed during the competition to tackle multi-agent environments with dynamic entity numbers. We'll examine their unique features, discussing why certain approaches excel while others face challenges. This section will employ terminology specific to the Lux environment, which we'll introduce more briefly in section 1.4. Numerous approaches to large multi-agent systems rely on logic systems characterized by extensive lines of code, meticulous fine-tuning to handle edge cases, and the accumulation of a comprehensive domain knowledge, upon which rule-based agents operate (Du *et al.* 2024; Aguayo-Canela *et al.* 2021). Within the Lux Environment, several logic-based solutions have emerged, demonstrating notable success (Anderson 2023; Tigga 2023; Kostuch 2023). However, the primary drawback of these systems lies in their scalability, particularly in terms of code complexity and the interoperability of rules. Furthermore, the exponential growth in environment complexity significantly amplifies the number of edge cases that agents must account

for. Reproducibility and comprehensibility of the code after inspection pose additional challenges due to the codebase often exhibiting a notoriously known anti-pattern called spaghetti code (Politowski *et al.* 2020). These rule-based distributed agents were designed to prioritize efficient resource collection and allocation, employing alternating phases to effectively generate more resources than expended.

Data-driven methods like Imitation Learning have garnered attention in recent studies, thanks to the abundant hardware resources and investments (Hussein *et al.* 2017). These techniques leverage learning from replay methods, allowing IL models to discern patterns from expert replays. These replays can originate from human-generated data or be acquired from larger models, resulting in more compact, distilled models. However, Imitation Learning demands substantial amounts of data and computational resources to operate efficiently (Goecks *et al.* 2022; Garg *et al.* 2022). Proposed methods within the Lux Environment have shown competitiveness but at significant costs, both in terms of training expenses and data acquisition (Nagradov 2023). Imitation Learning has also been used in neural network architecture search (Limburg 2023), resulting in three months of training and data collection time.

In the current landscape of deep learning and the large-scale era, deep reinforcement learning models of substantial size have set new benchmarks. For instance, models trained from scratch, such as AlphaGo Zero and GOAT, necessitate a considerable amount of computational power for training, with AlphaGo Zero requiring approximately 7.8e+13 GFLOPs (78 PetaFLOPs) and comprising over 46 million parameters, while GOAT has upwards of 35 million parameters (Figure 1.1). Language models exemplify this trend even more starkly, with PalM2 possessing an unprecedented 340 billion trainable parameters (Sevilla *et al.* 2022).

In the domain of complex, high-dimensional grid environments (Eberhardinger *et al.* 2023), the deployment of deep neural networks is crucial for achieving effective generalization. This requirement is particularly pronounced in the context of the Lux environment, where map heterogeneity is guaranteed by design, where the generation of different environments is seed-dependent. To tackle these issues, (Chen *et al.* 2023) proposed a large-scale deep learning solution designed to analyze a wide-ranging, high-dimensional feature space, incorporating diverse environmental attributes like ore maps, ice maps, and distances. They introduced a comprehensive global reward system aimed at recognizing and rewarding any positive developments of an agent, thereby achieving a highly generalized model capable of adapting across various scenarios. This approach necessitated substantial computational resources, specifically requiring an V100 GPU and an additional 600 CPU cores, with the training process extending over multiple phases spanning several days.

Recent advancements include a deep reinforcement learning algorithm specifically engineered for the complexity of industrial multi-agent systems. Addressing a gap in the research, their approach, the K-nearest multi-agent deep RL (Khorasgani *et al.* 2022), is designed to handle scenarios with a fluctuating number of agents and action dependencies among them. Demonstrated through a fleet management simulation by Hitachi, their algorithm showed potential for significant improvements in productivity by optimizing collaborative tasks, such as traffic management, highlighting its applicability in dynamic industrial environments. The paper by (Min *et al.* 2024) presents a novel approach in multi-reward reinforcement learning focused on generating counselor reflections. They introduce two innovative bandit methods, DynaOpt and C-DynaOpt (Min *et al.* 2024), for dynamically adjusting reward weights during training, aiming to optimize text qualities. Their methods outperform existing models, demonstrating significant potential

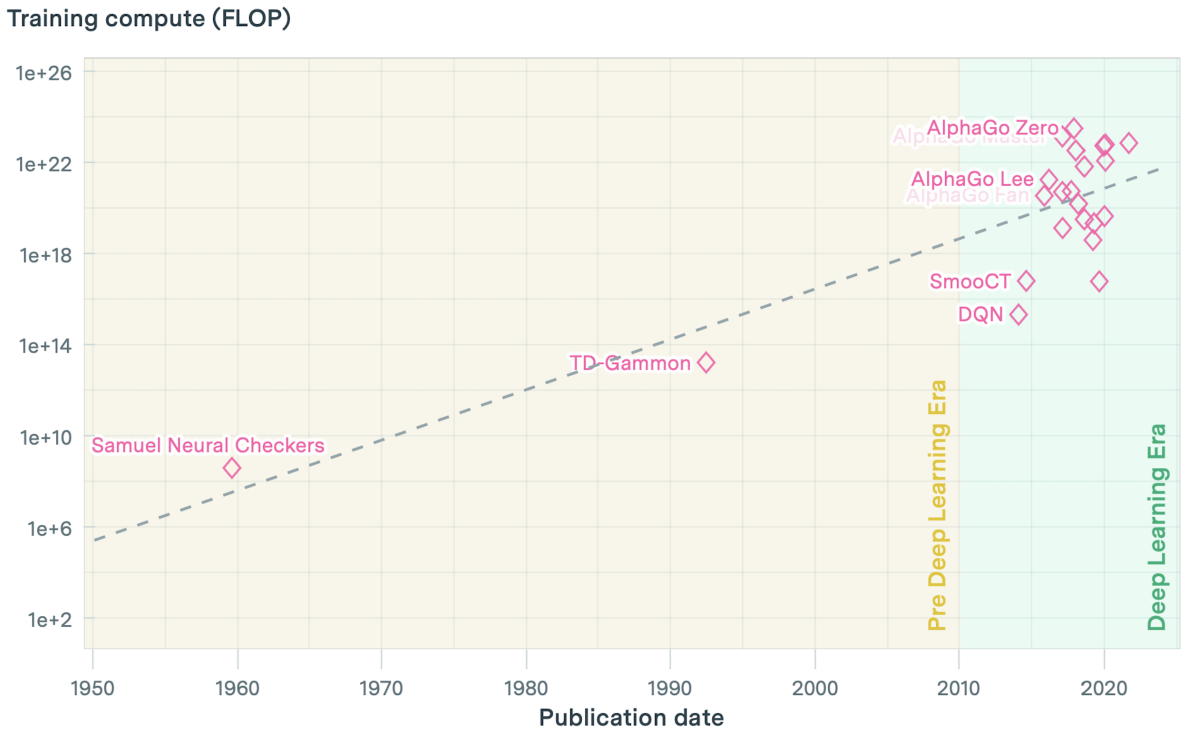


Figure 1.1: Exponential growth in training compute requirements for notable AI systems from the 1950s to the present, illustrating the transition from early machine learning to the current Large Scale Era (Epoch 2024).

for multi-objective problem decomposition and dynamic rewarding systems. (Tan *et al.* 2024) propose an adaptive distributed reinforcement learning method for multi-objective optimization in dynamic, distributed Intelligent Transportation Systems. Their approach addresses the challenge of optimizing multiple, potentially conflicting objectives in a changing environment by integrating multi-agent reinforcement learning with adaptive few-shot learning. Tested in an ITS scenario, their algorithm demonstrates superior adaptation and performance across individual and system metrics. (Zheng & Yu 2024) introduce a novel approach in Multi-Agent Reinforcement Learning (MARL) employing a hierarchy of Reward Machines (RMs) to enhance learning efficiency in cooperative tasks. Their method, MAHRM, adeptly manages complex scenarios with concurrent events and interdependent agents by breaking down tasks into simpler sub-tasks.

1.3 Background

Reinforcement Learning (RL) encapsulates all the processes through which agents learn optimal behaviors through exploration and the evaluation of actions based on rewards and penalties. This framework excels in single-agent, single-objective optimization scenarios because it directly aligns an agent's learning process with the maximization (or minimization) of a specific goal, enabling the agent to iteratively refine its strategy towards achieving optimal performance in well-defined environments (Lee *et al.* 2020). Exploring advanced Reinforcement Learning topics necessitates a solid grasp of the fundamental RL concepts due to the technical complexity of the field. Specifically, Policy Optimization algorithms, which extend basic RL principles, are detailed and form the basis for further discussion. This discourse selectively highlights rel-

evant concepts, omitting details beyond the scope of our research. For a thorough foundational understanding, the seminal work "Reinforcement Learning: An Introduction" by Sutton and Barto (Sutton & Barto 2018) is an essential reference.

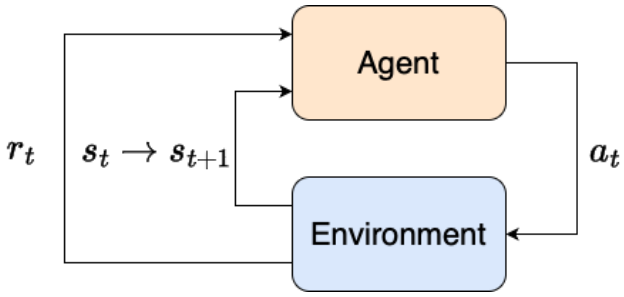


Figure 1.2: The standard representation of the agent-environment control loop depicts the agent as the actuator and the environment as a sensor, sending signals to this actuator. These signals typically consist of reward signals, reflecting the efficacy of the agent's last action a_t, and a new environment signal s_{t+1}, updated based on the agent's actuator action a_t.

In reinforcement learning, the foundational components are the **environment** and the **agents** operating within it. Agents interact with the environment by executing actions informed by **observations**, which may vary in terms of their completeness and detail. Based on these observations, the agent decides on **actions**, influencing the environment, which can also change independently. The agent aims to maximize its **total rewards** over time, known as the return, by learning optimal actions through its interactions. Reinforcement learning techniques enable this learning process by guiding the agent to achieve its goal of reward maximization.

In deep reinforcement learning, a state s is a detailed representation of the environment, containing all pertinent information, while an observation o provides a subset of this information, possibly omitting crucial details. Deep RL frameworks typically use vectors, matrices, or tensors to model these states and observations. For example, an image might be represented by its RGB values, and a robot's status by vectors of its joint angles and velocities. The environment's **observability** is classified based on the agent's access to information: it is fully observed if the agent perceives the entire state, and partially observed if the agent only gets incomplete observations. We will not consider partial observability, since our work focuses only on a fully observed environment (section 1.4).

The **action space** of an environment defines all possible actions an agent can execute. Environments may feature **discrete** action spaces, where the choices are finite and distinct (Mnih, Kavukcuoglu, *et al.* 2013; Silver *et al.* 2016), or **continuous** action spaces, characterized by actions as real-valued vectors that offer a spectrum of possibilities. This research explores an environment that integrates both discrete and bounded continuous actions (section 1.4; Figure 1.6).

Regarding the actions available to the agent, we define the concept known as a **policy**. Policies may be **deterministic**, implying that, for a given state, the action selected is consistent and not derived from a probability distribution of actions (Sutton & Barto 2018; Equation 1.1) .

$$a_t = \mu(s_t) \tag{1.1}$$

The chosen action can be **stochastic**, derived by sampling from a probability distribution over possible actions (Sutton & Barto 2018; Equation 1.2).

$$a_t \sim \pi_\theta(\cdot|s_t). \tag{1.2}$$

In deep reinforcement learning we are dealing with prameterized policies, which are characterized by their outputs being computable functions reliant on a distinct parameter set, such as the weights inside an **artificial neural network** (ANN), that we can adjust dynamically to the modify the actions of the agent via optimization methods. These weights are usually denoted by θ (Equation 1.2). In our research, we exclusively focus on stochastic policies to facilitate exploration; hence, discussions on deterministic policies are beyond the scope of this research.

The two primary stochastic policies implemented in current deep reinforcement learning frameworks are **categorical** and **diagonal Gaussian policies** (Achiam 2018). Categorical policies are designed for discrete action spaces. Diagonal Gaussian policies, on the other hand, are appropriate for continuous action spaces. The Lux Environment utilizes discrete and bounded continuous action values within its action space (section 1.4; Figure 1.6), but for the purposes of simplification, we used a discretization technique known as **binning**. This approach significantly improves performance in on-policy optimization (Tang & Agrawal 2020), allowing action prediction in continuous domains using categorical policies. These policies serve as classifiers within the action selection mechanisms, utilizing a neural network to process inputs and ending with a linear layer that generates **logits** for each possible action. Logits represent the unnormalized scores output by the linear layer, which are then converted into a probabilistic distribution of actions through the softmax function.

A **trajectory** (Sutton & Barto 2018; Equation 1.3) represents a sequence composed of state-action pairs as experienced within the environment.

$$\tau = (s_0, a_0, s_1, a_1, \ldots). \tag{1.3}$$

Typically, the initial state of the environment is drawn from a **starting state distribution** (Sutton & Barto 2018; Equation 1.4).

$$s_0 \sim \rho_0(\cdot). \tag{1.4}$$

State transitions within an environment are dictated by the environment's inherent dynamics and depend on the previous action executed, represented by a_t. These transitions may follow a deterministic or stochastic pattern, as highlighted in (Sutton & Barto 2018; Equation 1.5).

$$s_{t+1} \sim P(\cdot|s_t, a_t). \tag{1.5}$$

Specifically, the Lux Environment runs under deterministic rules, as detailed in section 1.4, ensuring that each action directly results in a predictable environmental update. Moving forward, we will term trajectories as **rollouts**, adopting a standard nomenclature used in RL.

The reward function, represented by R has diverse notational variations, with a common formulation involving the agent's current state, the action executed in that state, and the subsequent state resulting from the action (Sutton & Barto 2018; Equation 1.6).

$$r_t = R(s_t, a_t, s_{t+1}) \tag{1.6}$$

The notation for reward is often simplified to depend solely on the state at timestep t, $R(s_t)$, or on both the current state and the recently executed action at timestep t, $R(s_t, a_t)$. The objective of the agent is to maximize the total accumulated reward throughout a rollout, denoted as $R(\tau)$ (Sutton & Barto 2018). However, given the variety of possible returns, this notation can be ambiguous. The first concept of return encountered in the literature is termed the **finite-horizon undiscounted return**, which aggregates the rewards collected within a predetermined rollout window (Sutton & Barto 2018; Equation 1.7).

$$R(\tau) = \sum_{t=0}^{T} r_t. \tag{1.7}$$

Mathematically, the reward may not always converge to a finite value. To address this, the concept of **infinite-horizon discounted return** is introduced, which applies a discount factor, γ to future rewards. This discounting ensures convergence under suitable conditions. When γ is set to 1, it results in the equivalent of an undiscounted return (Sutton & Barto 2018; Equation 1.8).

$$R(\tau) = \sum_{t=0}^{\infty} \gamma^t r_t. \tag{1.8}$$

The core goal of the agent is to adopt a policy that ensures the maximization of its expected return, regardless of the specific return metric or policy model used. In a deterministic environment, such as the one addressed in our research, the probability function $P(s_{t+1}|s_t, a_t)$ transforms into a deterministic relation $s_{t+1} = f(s_t, a_t)$. This modification indicates that state transitions are directly determined by the current state and action, simplifying the calculation of trajectory probabilities under a given policy π (Sutton & Barto 2018; Equation 1.9).

$$P(\tau|\pi) = \rho_0(s_0) \prod_{t=0}^{T-1} \pi(a_t|s_t). \tag{1.9}$$

Given that the transitions are deterministic, the expectation of the return, $J(\pi)$, remains (Sutton & Barto 2018; Equation 1.10):

$$J(\pi) = \int_{\tau} P(\tau|\pi) R(\tau) = \mathbb{E}_{\tau\sim\pi}\left[R(\tau)\right] \tag{1.10}$$

The optimization challenge in reinforcement learning involves identifying the **optimal policy** by selecting the argument π that maximizes the expected return (Sutton & Barto 2018; Equation 1.11).

$$\pi^* = \arg\max_{\pi} J(\pi), \tag{1.11}$$

with π^* being the optimal policy.

Value functions quantify expected rewards from given states or state-action pairs under defined policies. Value functions estimate the benefit of an agent being in a specific state and play a role, in some form, across nearly all reinforcement learning algorithms (Achiam 2018; AlMahamid & Grolinger 2021). The **On-Policy Value Function** calculates the expected reward for being in state s while following a specific policy π, essentially computing the expected reward across a trajectory τ (Sutton & Barto 2018; Equation 1.12).

$$V^{\pi}(s) = \mathbb{E}_{\tau\sim\pi}\left[R(\tau)|s_0 = s\right] \tag{1.12}$$

The **On-Policy Action-Value Function** estimates the value of starting in state s, taking an initial arbitrary action a, and thereafter adhering to policy π, providing a measure that parallels the On-Policy Value Function but starts with a specific action a (Sutton & Barto 2018; Equation 1.13).

$$Q^{\pi}(s,a) = \mathbb{E}_{\tau\sim\pi}\left[R(\tau)|s_0 = s, a_0 = a\right] \tag{1.13}$$

Optimal value functions signify the maximum expected return obtainable from a state s when following the optimal policy within an environment. Conversely, optimal action-value functions indicate the highest expected return achievable by executing an action a in state s and adhering to the optimal policy for all subsequent decisions. The optimal value function is defined as (Sutton & Barto 2018; Equation 1.14):

$$V^{*}(s) = \max_{\pi} \mathbb{E}_{\tau\sim\pi}\left[R(\tau)|s_0 = s\right], \tag{1.14}$$

while the optimal action-value function has the following form (Sutton & Barto 2018; Equation 1.15):

$$Q^{*}(s,a) = \max_{\pi} \mathbb{E}_{\tau\sim\pi}\left[R(\tau)|s_0 = s, a_0 = a\right]. \tag{1.15}$$

In value and action-value functions, we assume an **infinite-horizon discounted return** to ensure convergence. Without discounting, a finite horizon is required to avoid infinite returns, making the concept of expected return meaningless and impractical for RL tasks. In the undiscounted case, incorporating time as an argument is necessary to account for time-dependence. An equally important connection in literature is to define the value function in terms of the action-value function (Sutton & Barto 2018; Equation 1.16; Equation 1.17), highlighting their integral relationship. The value function using the action-value function is defined as:

$$V^{\pi}(s) = \mathbb{E}_{a\sim\pi}\left[Q^{\pi}(s,a)\right] \tag{1.16}$$

while the optimal value function using the optimal action-value function has the following form:

$$V^{*}(s) = \max_{\mathrm{a}}\left[Q^{*}(s,a)\right] \tag{1.17}$$

The Bellman Equations, crucial to reinforcement learning, establish a recursive linkage between the value of a specific state and the values of its ensuing states. They assert that the optimal policy at any stage matches the optimal policy at all future stages, effectively stating that the value of a state equals the immediate reward plus the expected value of next states under the optimal policy. This principle ensures the consistency and optimality of decision-making across time. Bellman equations use **on-policy** notation (Sutton & Barto 2018; Equation 1.18; Equation 1.19), calculating state values or action-values based on a specified policy π [1].

$$V^{\pi}(s) = \mathbb{E}_{\substack{a\sim\pi(\cdot|s)\\ s'\sim P(\cdot|s,a)}}\left[r(s,a) + \gamma V^{\pi}(s')\right] \tag{1.18}$$

$$Q^{\pi}(s,a) = \mathbb{E}_{s'\sim P(\cdot|s,a)}[r(s,a) + \gamma\, \mathbb{E}_{a'\sim\pi(\cdot|s')}\, Q^{\pi}(s',a')] \tag{1.19}$$

For both value iteration and policy iteration algorithms, the Bellman equations ensure convergence to the optimal policy (Sutton & Barto 2018), given enough iterations and under certain

[1] This notation sometimes leads to confusion with the concepts of on-policy and off-policy learning. In off-policy scenarios, though the equations appear similar, the policy π refers to the target policy, whereas data collection follows a different behavior policy μ.

conditions, such as a finite state space or the presence of a discount factor in infinite horizons. The Bellman equations for optimal value and action-value functions resemble the on-policy framework, yet differ in action selection. Instead of sampling actions a from a state-dependent probability distribution, the optimal framework selects the action that maximizes the future value (Sutton & Barto 2018; Equation 1.20; Equation 1.21).

$$V^*(s) = \max_a \mathbb{E}_{s' \sim P(\cdot|s,a)} \left[r(s,a) + \gamma V^*(s') \right] \tag{1.20}$$

$$Q^*(s,a) = \mathbb{E}_{s' \sim P(\cdot|s,a)} [r(s,a) + \gamma \max_{a'} \mathbb{E}_{a' \sim \pi(\cdot|s')} Q^*(s',a')] \tag{1.21}$$

In modern RL algorithms, the calculation of a so-called **advantage function** plays a crucial role in optimal decision making. The advantage function quantifies the benefit of selecting a particular action a in state s compared to the average result of pursuing other available actions in the same state, given adherence to policy π subsequently. The on-policy advantage function, $A^\pi(s,a)$, measures the expected benefit of choosing action a compared to following the policy's action distribution in state s. This calculation involves subtracting the value function, representing the average benefit of being in state s, from the action-value function for a specific action a, thus determining a relative improvement over other actions (Sutton & Barto 2018; Equation 1.22).

$$A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s) \tag{1.22}$$

Our research environment (section 1.4) can be formulated as a fully-observable **Markov Decision Process**, which is a formal notation frame work for an agent's environment. It is mathematically represented to indicate adherence to the Markov property, meaning that state transitionts do not rely on past history. At any given timestep t, the next state s_{t+1} is influenced by the current state s_t and the action a_t taken at that state. In our study, we model our environment as an MDP (Achiam 2018), represented by the tuple $\langle S, A, R, P, \rho_0 \rangle$, where:

- S is the set of all possible states, including map features, units, and resources.
- A encompasses the entire set of allowable actions, such as digging and moving.
- $R : S \times A \times S \to \mathbb{R}$ is the reward function, devised based on a specific reward scheme.
- $P : S \times A \times A_{adv} \to P(S)$ denotes the state transition probability function. This function is characterized as stochastic in adversarial environments. Transforming the setting to a cooperative scenario, or making the adversary passive, results in a deterministic transition probability of 1.
- ρ_0 represents the starting state distribution, determined by a controlled seeding method.

Modern RL algorithms fit into a diverse range of domains and applications. Our research focuses on **model-free methods**, which operate without an explicit environmental model, restricting the ability to predict state transitions and rewards. This absence prevents the utilization of lookahead or planning techniques to enhance policy learning through forward simulation. Consequently, model-free approaches rely solely on empirical interactions heavily compromising sample efficiency. Moreover, the transferability of models learned from empirical data to real-world scenarios is limited, often requiring substantial computational resources for effective adaptation.

In model-free learning, the agent cannot directly access the expected future rewards or the total distribution of the future states and rewards at the start of learning. Instead, these **distributions must be approximated** through various methods that explore potential states and rewards

within the environment. This concept is central to the multi-armed bandit problem (Sutton & Barto 2018), a form of statistical optimization where the agent must find the optimal balance between exploring the environment to accurately approximate the total distribution of rewards and states and exploiting the environment to maximize reward collection. This framework comprises an evolutionary multi-objective optimization challenge (Zitzler 2012). Ideally, an agent could achieve a perfect distribution understanding through the law of large numbers with infinite samples and negligible time. However, in practice, resources are finite, and the goal becomes optimizing the utilization of given resources.

In the case of **multi-armed bandits**, imagine a slot machine with n levers, each lever dispensing rewards from a different and unknown distribution. The algorithm's goal is to devise an optimal strategy that balances the initial exploration of each lever — to collect sufficient data for approximating the distribution of rewards — with the exploitation of this data to maximize accumulated rewards by prioritizing levers with higher payouts (Sutton & Barto 2018). The complexity of finding an optimal solution increases with the complexity of the reward distributions, particularly if they are heavily skewed, irregular, or multimodal, as these require more samples to approximate accurately. For example, consider a lever that typically yields no or minimal rewards but, on rare occasions, dispenses the largest jackpot in the country. This scenario would necessitate more extensive sampling compared to a lever offering moderate but consistent payouts, where the reward distribution might closely resemble a uniform distribution.

Another distinction between algorithms is concerned with what is being learned by the agent, which can be **policies**, **value functions**, *action-value functions* or the **model of the environment**. There are two main approaches to represent model-free reinforcement learning: **policy gradient methods** and **value-based methods**.

To properly understand policy gradient methods, it's important to differentiate between an **agent** and an **entity** in the environment. In scenarios with a single agent, the policy is the learning entity controlling the algorithm while following a policy (Sutton & Barto 2018). However, in multi-agent systems, it's necessary to distinguish between passive or heuristic entities and learning entities. The definition of an agent in multi-agent systems depends on how rewards are distributed across agents, whether based on global or local information, how experience is divided into multiple trajectories, and how action probabilities, entropies, and advantages are calculated. If an entity in the environment is passive or follows a pre-defined heuristic operation, it's not considered an agent, but rather a passive entity. If the same rewards are distributed to all agents based on a single trajectory using global observation, the agent is considered the **orchestrator**. Conversely, when rewards or other computations are allocated at the entity or group level, agents are referenced accordingly to those groups or individual entities. This approach may encompass all entities present on the map as agents, as rewards, advantages, and model updates are computed based on individual trajectories and at the individual level.

When dealing with multiple agents, the method of credit assignment must also be mentioned. In a cooperative scenario, rewards must reflect not only the individual's achievement but also the shared goals of the entire team in order to drive the agent towards the desired degree of altruistic behavior. By forgoing team-based rewards, the agents might prefer to act in their own self-interest, while not providing them with enough "selfish" rewards could result in their inability to learn basic functions due to the sparsity of the reward feedback. This phenomenon is called the credit assignment problem, and multiple approaches have been theorized for its mitigation. These include the utilization of counterfactual reward baselines (J. Foerster *et al.* 2017) and

forgoing the multi-agent aspect of the scenario entirely with the use of an orchestrator (Chen *et al.* 2023).

1.3.1 Policy Gradient Methods

Policy optimization algorithms explicitly parameterize the policy by employing a set of parameters denoted as θ. The objective of these algorithms is to optimize these parameters by optimization methods on the objective function, denoted as $J(\pi_\theta)$, which is the total expected return across a given trajectory τ (Achiam 2018). Optimization techniques may also focus on maximizing local approximations of this function. These algorithms operate on an on-policy basis, meaning they use data generated from the current policy's actions to calculate updates. The process includes learning a value function approximator, $V_\theta(s)$, for the on-policy value function, $V^\pi(s)$, which assists in guiding policy updates.

To optimize the objective function $J(\pi_\theta)$ via gradient ascent, it's essential to define the concept of the **policy gradient**, which represents the derivative of the expected reward with respect to the policy parameters θ. This leads to the formulation of the parameter update rule as follows (Achiam 2018; Equation 1.23):

$$\theta_{k+1} = \theta_k + \alpha \nabla_\theta J(\pi_\theta) \tag{1.23}$$

Policy gradient algorithms, including Trust Region Policy Optimization, Proximal Policy Optimization, Recurrent-PPO, and Masked-PPO, form the basis of our research experiments presented. These algorithms utilize distinct update mechanisms, ratios and gradient clipping methods to steer the policy toward optimality (Lehmann 2024). For implementation purposes, a generalized, computationally viable and numerically stable form of the update rule is adopted from vanilla policy gradient techniques. The derivation of the analytical gradient is beyond the scope of this study and is detailed in Sutton and Barto's book (Sutton & Barto 2018).

The complete derivation of the vanilla policy gradient results in an expression representing the expected cumulative sum of the policy's log probability gradients, each weighted by the trajectory's total reward, across all possible trajectories (Achiam 2018; Equation 1.24).

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \underbrace{R(\tau)}_{\text{total reward}} \right] \tag{1.24}$$

Given the principle of the law of large numbers (Athreya & Lahiri 2006), the expected form of the vanilla policy gradient can be estimated by aggregating a large volume of environment rollouts. By defining this set of rollouts as D, where the cardinality of the set is $|D| = N$, the formula for the vanilla policy gradient is accordingly updated for estimation (Achiam 2018; Equation 1.25).

$$\hat{g} = \frac{1}{|D|} \sum_{r \in D} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau) \tag{1.25}$$

State-of-the-art policy gradient algorithms adopt a refined version of this fundamental gradient update formula in their original implementations. The computability of the gradient of log probabilities of the policies with respect to the parameters, θ, denoted by $\nabla_\theta \log \pi_\theta(a_t|s_t)$ has to be ensured (Achiam 2018). As the sample size N increases, the sample mean more closely approximates the true mean, resulting in a sample estimate with significantly reduced variance.

In policy gradient methods, a key mathematical caveat involves the updates to the log probabilities of actions being directly proportional to $R(\tau)$ (Achiam 2018; Equation 1.24), which represents the sum of rewards obtained throughout the whole trajectory τ. This methodology may initially appear counterintuitive, as the objective is to condition agent behavior on the future outcomes of their actions, or the rewards accrued subsequent to a specific timestep t. By adjusting the vanilla policy gradient equation to utilize a **reward-to-go** formulation instead of the total reward function, the policy gradient becomes more stable, achieving lower variance in parallel (Achiam 2018; Equation 1.26).

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_t|s_t)\underbrace{\sum_{t'=t}^{T}R(s_{t'},a_{t'},s_{t'+1})}_{\text{reward-to-go}}\right] \tag{1.26}$$

The "reward-to-go" trick significantly enhances the calculation of policy gradients by addressing a specific optimization problem intrinsic in the vanilla policy gradient equation, known as the **Expected Grad-Log-Prob (EGLP) Lemma** (Achiam 2018). This lemma specifies that for a continuous probability distribution P_0 over a random variable x, when one integrates over x, differentiates both sides of the resulting equation, and then applies a log derivative trick, the outcome is the following (Achiam 2018; Equation 1.27):

$$\mathbb{E}_{x\sim P_\theta}[\nabla_\theta \log P_\theta(x)] = 0 \tag{1.27}$$

This equation shows that the product of the gradient of log probabilities with any function exclusively dependent on the current state t will also yield zero. This introduces the baseline function, denoted as $b(s_t)$, which, when added to or subtracted from the reward-to-go component in (Achiam 2018; Equation 1.26), does not affect the expected value of the equation.

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_t|s_t)\left(\sum_{t'=t}^{T}R(s_{t'},a_{t'},s_{t'+1}) - \underbrace{b(s_t)}_{\text{baseline}}\right)\right] \tag{1.28}$$

In policy gradient algorithms like PPO, TRPO, A2C, and VPG, the baseline function is set as the on-policy value function $V^\pi(s_t)$. Using $V^\pi(s_t)$ as a baseline improves stability and convergence but requires neural network estimation. This requires simultaneous updates to both the policy and the value function estimate, an approach found in **actor-critic** methods, where the objective is typically defined by the mean-squared-error (MSE) loss (Achiam 2018; Equation 1.29).

$$\phi_k = \arg\max_\phi \mathbb{E}_{s_t,\hat{R}_t\sim\pi_k}\left[\left(V_\phi(s_t) - \hat{R}_t\right)^2\right] \tag{1.29}$$

The final generalization of (Achiam 2018; Equation 1.24) can be expressed as the following:

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_t|s_t)\Phi_t\right], \tag{1.30}$$

where the choice for Φ_t may include **total reward** $(R(\tau))$, **reward-to-go** $\left(\sum_{t'=t}^{T}R(s_{t'},a_{t'},s_{t'+1})\right)$, **reward-to-go + baseline** $\left(\sum_{t'=t}^{T}R(s_{t'},a_{t'},s_{t'+1}) - b(s_t)\right)$ or be de-

termined by alternative approaches, such as the on-policy action-value function (Achiam 2018; Equation 1.31):

$$\Phi_t = Q^{\pi_\theta}(s_t, a_t), \tag{1.31}$$

or the advantage function with its refinement, **General Advantage Estimation (GAE)**, employed in algorithms such as PPO and its variants, due to the necessity of estimating the advantage, which cannot be directly calculated (Achiam 2018; Equation 1.32).

$$\Phi_t = A^{\pi_\theta}(s_t, a_t) = Q^{\pi_\theta}(s_t, a_t) - V^{\pi_\theta}(s_t) \tag{1.32}$$

1.3.2 Value-Based Methods

Value-based methods aim to estimate the values $V^\pi(s)$ or action-values $Q^\pi(s,a)$ of states under a policy π, adopting a strategy to determine these optimal values first (Achiam 2018). Consequently, these methods develop a deterministic policy optimized for exploitation by choosing actions that maximize expected returns, guided by choosing actions with respect to the greedy policy. RL literature, such as Sutton and Barto's book (Sutton & Barto 2018), heavily focus on explaining a lot these methods, but in our research we only utilized Q-function approximations with deep Q-learning, therefore we will mention the most notable value-based methods. We think its important to go trough these in a minimalistic way, since they help to build up intuititon for more complex approaches:

- **Value Iteration**: a dynamic programming method for finding either the optimal value function $V^*(s)$ or the optimal action-value function $Q^*(s,q)$ trough solving the Bellman Equations in an iterative way. It has been shown to converge if certain conditions are met (Della Vecchia *et al.* 2011). Requires the model of the world; in particular, knowing $P(s', r|s, a)$ (Sutton & Barto 2018).
- **Monte-Carlo Methods**: estimate state or action values by averaging the returns from many complete episodes, updating value estimates only at the end of each episode. These methods do not use partial updates or rely on existing value estimates, instead depending solely on the total actual rewards received. Q-tables are used to maintain Q-values troughout the learning process. This makes the methods impossible to scale to very large problems (Sutton & Barto 2018).
- **Temporal Difference Methods**: due to the high variance associated with Monte Carlo methods, TD-learning introduces **bootstrapping**, which updates Q-values using the immediate reward received and an estimate of future rewards, rather than relying solely on the total discounted return G. Two renowned methods fit into this category of algorithms: Q-Learning and SARSA (Sutton & Barto 2018).
- **n-step Methods**: essentially a combination of both trying to find a sweet spot between deep and shallow backups. Introduces the n-step SARSA, n-step Q-Learning algorithms (Sutton & Barto 2018).

Addressing major challenges in Q-learning, such as extremely poor scaling in large environments with vast state and action spaces, **function approximation methods** have been developed. Notably, linear function approximation adopts a statistical approach, while deep Q-learning leverages artificial neural networks to estimate **Q-values**. Our research primarily focuses on deep Q-learning, but a brief overview of linear function approximation is also beneficial for understanding the broader context of approximation methods in Q-learning.

In linear function approximation, the objective is to approximate a value by using a linear combination of features and weights. Specifically for Q-function approximation, this involves representing the state space through a well-defined set of features, each assigned a specific weight. Instead of directly storing state-action pairs, the approach uses feature vectors $f(s,a)$, where each vector has a dimensionality of $n \times |A|$. Here, n denotes the number of features, and $|A|$ represents the total number of possible actions (Miller 2023; Equation 1.33).

$$f(s,a) = \begin{bmatrix} f_1(s,a) \\ f_2(s,a) \\ \vdots \\ f_{n\times|A|}(s,a) \end{bmatrix} \tag{1.33}$$

A weight vector of the same size W is assigned to correspond with each feature-action pair. The final Q-value is then calculated as the linear combination of these feature vectors and weight vectors (Miller 2023; Equation 1.34).

$$Q(s,a \mid W) = f_1(s,a) \cdot w_1^a + f_2(s,a) \cdot w_2^a + \ldots + f_n(s,a) \cdot w_n^a \tag{1.34}$$

For effective weight updates in linear function approximation, it is crucial to start with a proper initialization technique and an update rule (Miller 2023; Equation 1.35). Given the linear nature of the model and the convexity of the optimization problem, these factors ensure convergence (Szepesvári 2010).

$$w_i^a = w_i^a + \alpha \cdot \delta f_i(s,a) \tag{1.35}$$

In deep Q-learning, the task of updating weights and engineering features for a statistical linear function approximator is done by an artificial neural network. This network automatically learns the features through its layers. A significant advantage of this approach is its versatility; the ANN can handle various types of environment representations, including images, gridmaps, videos, and text. The improved update rule becomes the following (Miller 2023; Equation 1.36):

$$\theta = \theta + \alpha \cdot \delta \cdot \nabla_\theta Q(s,a|\theta) \tag{1.36}$$

In deep Q-learning, methods like DQN improve stability by using two neural networks: a primary Q network for updating Q-values and a separate target network, whose weights are periodically synced with the Q network, to provide a stable reference for updates.

1.3.3 Evolution Strategies

Evolutionary algorithms are a specific subset of optimization techniques that find unique applications in reinforcement learning. Though evolutionary algorithms and RL are strictly categorized as distinct subfields, evolutionary strategies are useful for neural network search and optimization, enhancing methods such as deep Q-learning and policy gradient techniques. Despite being less sample efficient than conventional RL algorithms, evolutionary algorithms require less computational resources and are easily parallelizable, leading to improved processing speeds (Yanes Luis *et al.* 2021).

Evolutionary algorithms draw inspiration from **biological evolution**, utilizing processes such as reproduction, mutation, recombination, natural selection, and survival of the fittest. These mechanisms make them inherently profficient at exploration and optimization tasks. Among

these, the **genetic algorithm** (Holland & H. 1975) is a fairly well-known algorithm, simulating the process of natural selection inspired by Charles Darwin's theory. However, a significant challenge with genetic algorithms is the encoding of the problem space, which does not always ensure the discovery of the most optimal solution, especially in vast search spaces.

We utilized a specific domain of evolution, called **evolutionary artificial neural networks**, often termed Neuro-Evolution (Galvan & Mooney 2021). These methods optimize neural network architecture by evolving topologies, connections, activation functions, the number of hidden layers, and even weights until an optimal solution is reached. Neuro-Evolution is particularly valuable in scenarios where domain-specific knowledge is limited, making it challenging to construct an optimal network architecture through conventional means. In our environment, neuroevolution has been employed to systematically explore the optimization space by conducting randomized searches for an optimal policy.

1.4 Environment

The Lux AI Environment represents a 2D grid platform tailored for Multi-Agent Reinforcement Learning (MARL) research (Chen *et al.* 2023), designed to tackle challenges in multi-variable optimization, resource acquisition, and allocation within a competitive 1v1 setting. Beyond optimization, proficient agents are tasked with adeptly analyzing their adversaries and formulating strategic policies to gain a decisive advantage (Tao, Pan, *et al.* 2023a). The environment is fully observed by all agents.

1.4.1 Map

The world of Lux is represented as a **2D grid**. Coordinates increase east (right) and south (down). The map is always a square and has a variable size of 32, 48, 64 or 128 tiles. The (0, 0) coordinate is at the top left. The map has various features, including raw resources (Ice, Ore), refined resources (Water, Metal), units (Light, Heavy), Factories, Rubble, and Lichen. Figure 1.3 shows two possible generations of the game environment, visualizing different block types and resources. In our research, we utilized Lux AI S2 version 2.2.0 as the engine for our experiments (Tao, Q. Li, *et al.* 2023).

1.4.2 Early Game

Each player will start the game by bidding on factory placement order, then alternating placing several factories and specifying their starting resources. Factory placement policies play a crucial role in speeding up the learning process within the game environment. By strategically situating units closer to resources, these policies establish a less cluttered and more conducive proximity environment, thereby promoting faster resource collection. Additionally, the introduction of bidding mechanisms enhances the competitive aspect of the game, compelling players to outbid one another for optimal factory spawn locations. Players are given starting resources for the bidding process. Figure 1.4 shows map states after concluding the **factory placement** and **bidding tasks**.

[2]The maps were generated using seeds 560 and 788.
[3]Advanced specifications available on GitHub.

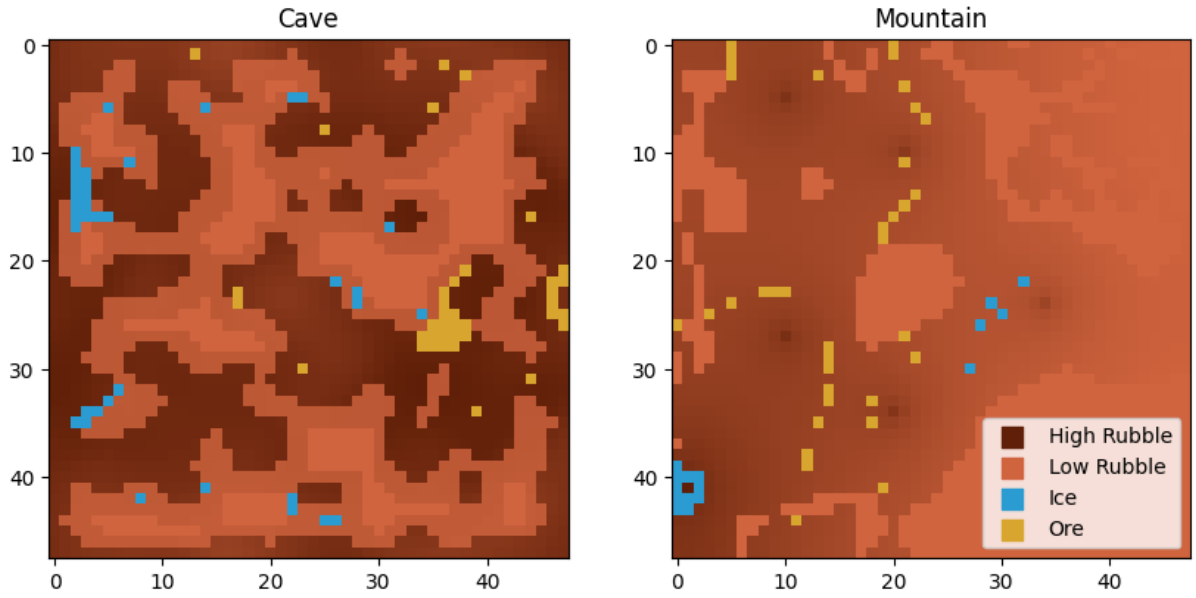


Figure 1.3: Visual Representation of Lux 2D map types which exhibit unique layouts, each generated based on specific seeds [2]. In cave maps, resources are situated inside ravines, while mountain maps, presenting a more challenging terrain, hide resources beneath extensive rubble.

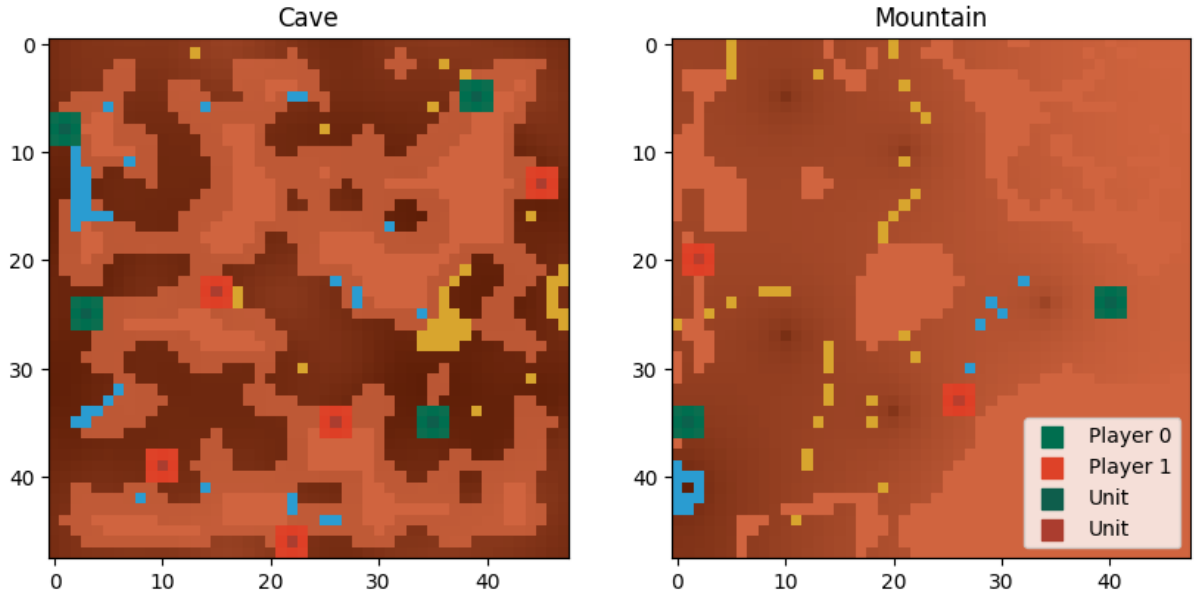


Figure 1.4: Visual representation of a random factory placement policy on the two generated maps presented on Figure 1.3. For further restrictions on factory placement and bidding, please refer to the more in-depth documentation [3].

1.4.3 Resources

The two kinds of raw resources shown in Figure 1.3, Ice and Ore, can be refined by a factory into Water or Metal respectively. The raw resources are collected by Light or Heavy units and then dropped off once a unit transfers them to a friendly factory, which then automatically converts them into refined resources at a constant rate. Refined resources are used for growing Lichen, powering factories as well as building more units. For detailed conversion rates and restrictions on transferring resources, please refer to the documentation (Tao, Pan, *et al.* 2023a).

1.4.4 Actions

Units and factories can perform actions at each turn, given certain conditions and enough power to do so. In general, all actions are simultaneously applied and are validated against the state of the game at the start of each turn (Chen *et al.* 2023). Every turn, players can give an action to each factory and a queue of actions to each unit. Units always execute actions from an action queue, which is limited to a size of 20, while factories directly execute actions. Players can choose to repeat actions for n times, which further complicates the action queue. Submitting a new action queue for a robot requires the robot to use additional power to replace it's current action queue. It costs an additional 1 power, for light units and an additional 10 power, for heavy units. For further information regarding action queue updates and repeat handling, please, refer to the documentation (Tao, Pan, *et al.* 2023a). There are six possible action types that are simplified for easier understanding on Figure 1.5.

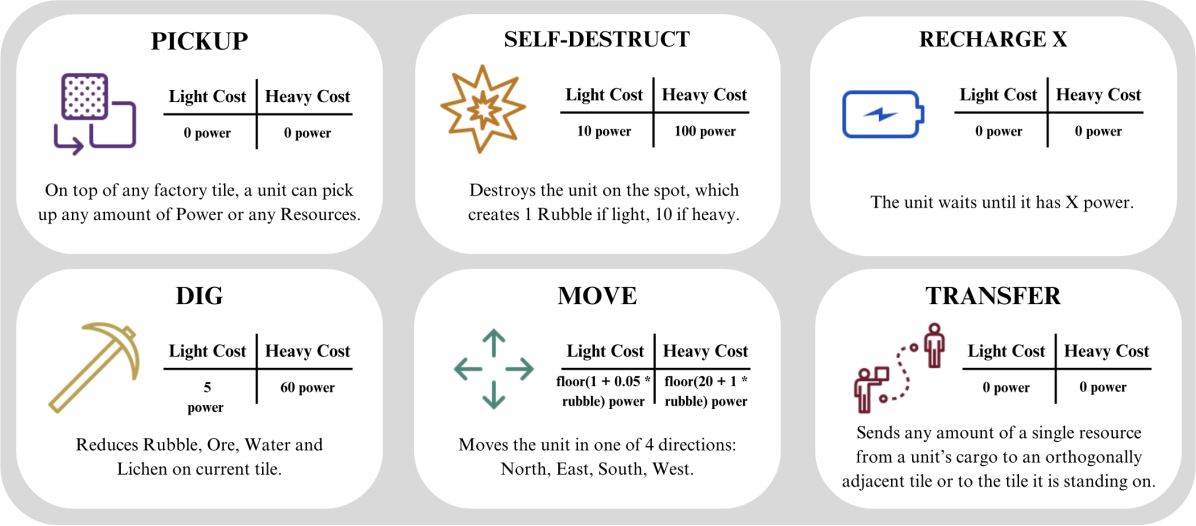


Figure 1.5: Simplified representation of all possible actions available in the Lux environment. Additional limits and restrictions are applied in the implementation of the engine (Tao, Pan, *et al.* 2023a).

A factory is a building that takes up 3×3 tiles of space. Units created from the factory will appear at the center of the factory. Factories can assume passive roles with fixed heuristic action generation, alongside active roles where actions are learned. The action vectors for both factories and units are 1D vectors. Given n units and m factories, predicting action vectors yields matrices of $n \times 6$ and $m \times 1$, respectively. The response template to the **Lux engine** conforms to the format illustrated in Figure 1.6.

Unit Action Vector	action_type	direction (move, transfer)	resource_type (ore, water, metal, ice, power)	resource_amount (transfer, pickup)	repeat	time_to_execute
unit_1 (move up and repeat 5 times)	0	4	0	0	5	5
unit_2 (dig and repeat 1 time)	3	0	0	0	1	1
unit_3 (pick up 250 power from factory once)	2	0	4	250	0	0
		x N				
unit_N (transfer 150 ore to the right tile once)	1	2	1	150	0	0

Factory Action Vector	action_type (light/heavy unit, water lichen)
factory_1 (create one light unit)	0
factory_2 (create one heavy unit)	1
factory_3 (create one heavy unit)	0
	x M
factory_M (water lichen)	2

Figure 1.6: Visual example of possible action vectors for both units and factories.

1.4.5 Rubble Dynamics

Each square on the map has a rubble value, which affects how difficult that square is to move onto. Rubble value is an integer ranging from 0 to 100. Rubble can be removed from a square by a light or heavy unit by executing the dig action while occupying the square. This environment also has unit collisions. Units which move onto the same square on the same turn can be destroyed and add rubble to the square according to a list of extensive rules in the documentation (Tao, Pan, *et al.* 2023a).

1.4.6 Night Cycles

Night cycles are introduced as a complexity measure in the Lux Environment to enhance dynamism and continual change. The environment alternates between **day cycles**, which last 30 steps, and **night cycles**, which are 20 steps long. This alternation continues until the episode concludes at 1000 steps. During night cycles, the recharging capability of units is stopped, although factories continue to produce power without change (Tao, Pan, *et al.* 2023a; Chen *et al.* 2023).

1.4.7 Win Condition

A game can be resolved in four possible ways:

- In the event that all factories belonging to a player explode, the opposing player is declared the winner.
- If both players lose their last factories in the same step, the game ends in a draw.
- If each player maintains at least one operational factory, victory in the Lux AI environment is determined by the quantity of Lichen tiles under their control. Lichen watering serves a dual purpose: establishing victory conditions and generating additional power for factories. Due to the intricate nature of power calculations and lichen watering dynamics, we advise consulting the detailed documentation for further understanding (Tao, Pan, *et al.* 2023a).
- If the quantity of owned Lichen tiles match, the game concludes in a draw.

Chapter 2

Methods

In this chapter, we would like to introduce the different methods utilized in our experiments. We will begin in section 2.1 with the specification of our initial trials in a simplified **single unit** environment, where we could study different reinforcement learning algorithms without the complexity of multiple agents. Next, in section 2.3 and section 2.4 we present two different approaches towards multi-agent control, namely **monolithic** and **hybrid** architectures. The section on the **hybrid approach** also contains the definition of our novel **trajectory separation** technique.

2.1 Single Unit Testbench

We started with a benchmarking study to assess the performance of various RL algorithms within a standardized environment to identify the optimal algorithm for our purposes. We utilized **OpenAI's Gym** (not Gymnasium at the time) package (Brockman *et al.* 2016) for compatibility with the Lux package and **PettingZoo** (J. Terry *et al.* 2021) for environment parallelization. To access state-of-the-art RL algorithms, we conducted our simple benchmarking study using **Stable Baselines 3** (Raffin *et al.* 2021), a reliable collection of reinforcement learning algorithms implemented in **PyTorch**. This toolkit provided us with a unified framework for all included algorithms and offered subprocess-level vectorization through Python's **Ray** (Ray/RLlib) framework (Moritz *et al.* 2018), **TensorBoard** support (Abadi *et al.* 2016), and easy out-of-the-box usability. Another significant advantage of Stable Baselines 3 is the fidelity of its implementation to the original specifications in the respective research papers, which meant that only minimal parameter adjustments were necessary for usage. Due to hardware constraints, specifically the availability of only eight logical CPU cores on our local machine, we were limited to running a maximum of eight parallel environments through subprocess-level vectorization.

2.1.1 Heuristic Bidding and Factory Placement

For our testbench, we simplified the problem space to focus on a **single-unit scenario**. This involved implementing a straightforward factory placement policy, a basic bidding system, and a simplified controller and action space for our agent. We also employed substantial **reward shaping** to focus the agent's learning on a specific task, which was then used for evaluation.

The factory placement policy utilized a comprehensive **heuristic search** for simplicity, gathering potential spawn positions and identifying areas with high Ice presence. By searching through

the ideal locations iteratively, the policy strategically positioned the factory adjacent to Ice clusters within a 3×3 grid. To completely remove the impact of bidding dynamics in adversarial settings, we adopted a strategy of bidding the minimum required resources while ensuring the opposing agent bid zero. This approach guaranteed us that we started factory placement with the maximum amount of resources possible [1].

2.1.2 Actions

In our study, we simplified the game dynamics by converting the factory into a non-learning, heuristic entity. It produced a single Heavy unit at the start of the episode, after which it played no further role, effectively circumventing complications associated with varying entity counts in reinforcement learning settings. Furthermore, we disabled the construction of Lichen, as its contribution to performance metrics — the agent's efficiency in exploring the map and securing necessary resources for the factory — was negligible. Here, therefore, when we use the term **agent**, we specifically mean the policy used by the Heavy unit present on the map. This approach, by confining our action space to a singular Discrete action space for one unit, made integration with Stable Baselines 3 easier by ensuring a consistent, discrete-variable-only action space.

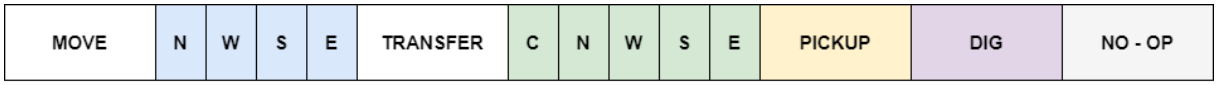


Figure 2.1: The action space inside our controller included movements in four cardinal directions, transfers to four cardinal directions and center [2], pickup [3], digging, and a no-operation action for debugging.

Figure 2.1 illustrates the action space adopted for our benchmarking analysis. We excluded certain operations available in the engine, including self-destruct, recharge, action queues (planning), and the transfer of resources other than Ice. The chosen action needed to be formatted to meet the specifications required by the Lux Engine, as detailed in Figure 1.6.

2.1.3 Observation

Our observation feature space was also designed with simplicity in mind, consisting of only a one-dimensional vector tailored specifically to the task of Ice mining. We intentionally excluded complex data types such as image maps, text, and extensive global information and conducted a small focused **ablation study** to make sure the most significant features are present in the input space of the model. As shown in (Figure 2.2), the features can be broken down into three components. The first section is the unit vector, which contains information about the state of the agent. Namely, the agent's position, class, team, and cargo data. The other two categories are observations about the environment itself. The features included here are vectors pointing towards the closest factory or ice tiles.

[1] Our goal was to neutralize the competitive element of the game for this benchmarking study, focusing instead on assessing algorithms through their proficiency in exploration and resource gathering rather than their ability to outperform an adversary.

[3] Transfer amount was set to the maximum available resource in cargo, circumventing binning or continuous bounded variables.

[3] The same rule applied here as in the previous footnote.

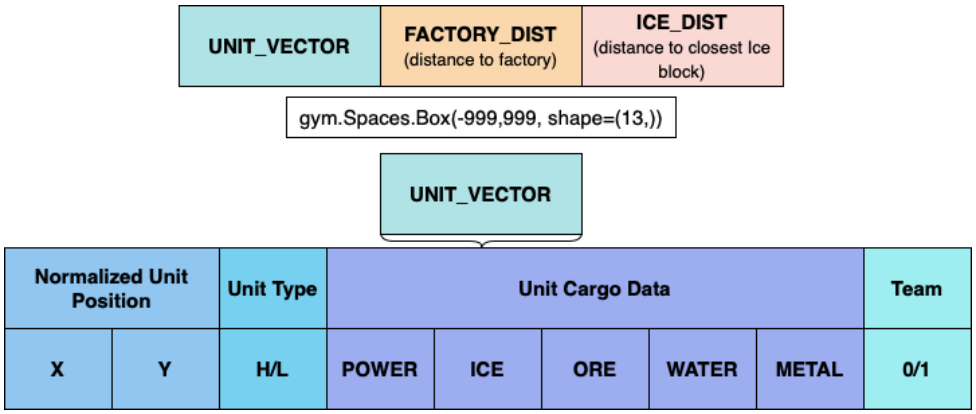


Figure 2.2: The feature space specifically optimized to maximize Ice collection.

2.1.4 Network Architecture

Our neural network architecture was also designed with efficient computation and speed in mind, incorporating a small **multilayer perceptron (MLP)** with two hidden layers. Each layer uses one **linear projection** followed by a **ReLU nonlinearity**. The final output layer is a linear projection that maps to the action space of the environment. We evaluated eight state-of-the-art reinforcement learning algorithms: ARS, A2C, DQN, PPO, QR-DQN, R-PPO, TRPO, and MPPO, each utilizing distinct learning approaches and objectives for the agent. Some algorithms employ an actor-critic approach with distinct policy and value heads learned concurrently, while algorithms like DQN and QR-DQN focus on trying to estimate Q-values in different ways for state-action pairs. In contrast, ARS does not learn values, Q-values, or policies directly.

On-policy actor-critic algorithms often employ **separate feature extractors** for the actor and critic components when utilizing simple network architectures like MLPs. This approach is computationally manageable even with relatively small state spaces and has been shown to yield better results (Andrychowicz *et al.* 2020). However, for more complex models that require a CNN backbone, a shared network architecture should be used since it enables the sharing of critical features between the actor and critic heads while significantly reducing computational overhead.

Value-based methods and ARS utilized **shared feature extractors**, as they focused solely on learning only Q-values, quantiles, and policy, without the need for distinct learning objectives such as separate value or policy heads. However, not all value-based methods follow this pattern. For instance, Double Deep Q-Learning (van Hasselt *et al.* 2015) employs two distinct networks: a primary online network for continuous learning and decision-making, and a separate target network, which is a clone of the primary network. The target network serves as a stable benchmark and is updated less frequently than the primary network.
It is important to note that while Stable Baselines 3 adheres to the specifications outlined in original research papers (Raffin *et al.* 2021), these documents often omit significant implementation details, necessitating cautious interpretation of results (S. Huang, Dossa, Raffin, *et al.* 2022). Moreover, Stable Baselines 3 incorporates significant, occasionally redundant, code overhead. This issue became more apparent when we used a single-file implementation of the

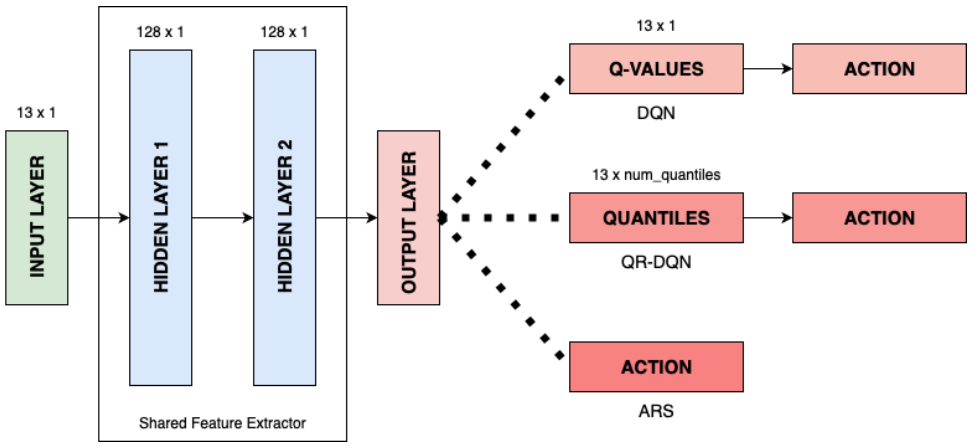


Figure 2.3: A high-level overview of the neural network layout that employs shared feature extractors used for value-based methods: DQN, QR-DQN, ARS. [4].

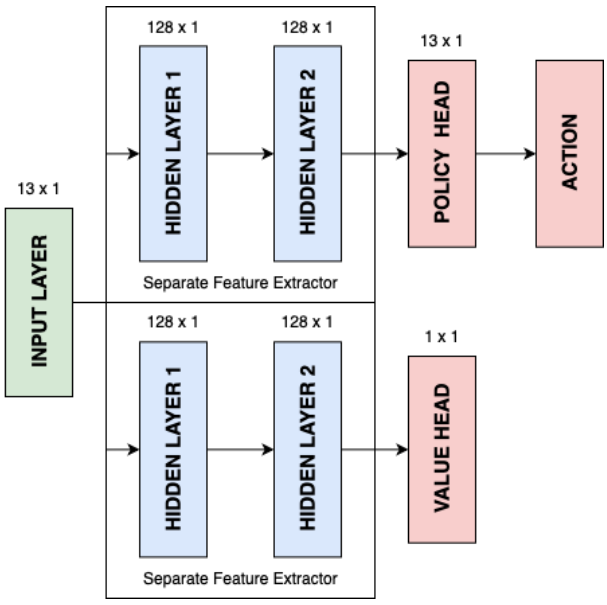


Figure 2.4: A high-level overview of the neural network layout that employs separate feature extractors used for actor-critic-based methods: A2C, PPO, TRPO, M-PPO, R-PPO. [5].

PPO algorithm (S. Huang, Dossa, C. Ye, *et al.* 2022), to which we introduced several optimization techniques. For this benchmark study, we are continuing with the SB3 baselines.

2.1.5 Reward Function

Lastly, our reward system was designed with heavy shaping to incentivize two specific behaviors. It encouraged the agent to increase Ice collection and motivated the factory to produce more Water than in the previous round. **This 1-step lookback** update-based reward system proved to be an effective method for encouraging targeted behaviors within our model.

$$r_t = \frac{\Delta\, ice\_dug}{100 + \Delta\, water\_produced} \tag{2.1}$$

2.1.6 Trust Region Policy Optimization (TRPO)

Policy optimization techniques aim to make significant updates to the policy, but this approach has inherent limitations. Specifically, Vanilla Policy Gradient (VPG) methods lack regularization mechanisms, gradient clipping, or specific thresholds for policy updates (Achiam 2018). This absence often results in a phenomenon known as policy collapse, primarily due to the high variance typically associated with policy gradient methods (Dohare *et al.* 2023). In classical neural network architectures, updating the model's parameters is straightforward because the loss function guides the adjustment of weights based on the deviation from known labels in the training dataset. However, in reinforcement learning, especially in model-free scenarios, this process becomes more complex. There is no explicit ground truth action for each state; instead, weight adjustments are guided by reward signals, which are often sparse and delayed, making it challenging to directly correlate actions to desired outcomes. This implies that in almost every deep RL task, there are some updates that push the policy gradients toward suboptimal directions. If such a negative update is relatively large, it can lead to a significant degradation of the policy, a situation from which the algorithm might not recover. This scenario is commonly referred to as **policy collapse**.

Trust Region Policy Optimization (J. Schulman, Levine, *et al.* 2015) was the first among policy gradient methods to implement a hard limit on gradient updates, based on the Kullback-Leibler Divergence between the probability distributions of actions before and after the updates. This constraint ensures that the model's parameters remain close in the parameter space, thus helping the model mitigate potentially detrimental updates. The parameter update rule looks like the following (J. Schulman, Levine, *et al.* 2015; Equation 2.2):

$$\theta_{k+1} = \arg\max_{\theta} L(\theta_k, \theta) \quad \text{w.r.t.} \quad D_{KL}(\theta||\theta_k) \leq \delta, \tag{2.2}$$

where $L(\theta_k, \theta)$ is defined as the **surrogate advantage function**, which measures how much better does π_θ perform over π_{θ_k} (J. Schulman, Levine, *et al.* 2015; Equation 2.3).

[5]It's important to note that in our implementation of Recurrent Proximal Policy Optimization (R-PPO), we opted for two LSTM layers with a hidden size of 128, instead of using traditional linear projection layers due to the recurrent nature of the algorithm.

$$L(\theta_k, \theta) = \hat{\mathbb{E}}_{s,a\sim\pi_{\theta_k}} \left[\underbrace{\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}}_{\text{importance sampling ratio}} \hat{A}^{\pi_{\theta_k}}(s,a) \right] \tag{2.3}$$

The update in Equation 2.2) is not straightforward to compute in an explicit way. Therefore, TRPO approximates the impact of a small change in policy parameters using a first-order Taylor series expansion. It linearly approximates how the objective function changes with a small change in parameters, constrained by a second-order Taylor expansion of the KL divergence to ensure the change is not too large (J. Schulman, Levine, *et al.* 2015; Equation 2.4).

$$\theta_{k+1} = \arg\max_\theta g^T(\theta - \theta_k) \quad \text{w.r.t.} \quad \frac{1}{2}(\theta - \theta_k)^T H(\theta - \theta_k) \leq \delta \tag{2.4}$$

The above optimization problem is addressed using Lagrangian duality (Knowles 2010), but potential inaccuracies from the Taylor approximation may not meet the KL divergence constraint, needing for a backtracking line-search for correct updates. Due to the computational intensity of directly calculating and storing the inverse Hessian, the conjugate gradient method (Wikipedia 2024a) is employed for approximation.

Algorithm 1 Trust Region Policy Optimization (J. Schulman, Levine, *et al.* 2015)

1: **Input:** Initial parameters for both the value, ϕ_0, and policy functions, θ_0.
2: **Hyperparameters:** KL-divergence threshold, δ, number of backtracks, K, and the backtracking coefficient, α.
3: **for** $k = 0, 1, 2, \ldots$ **do**
4: Collect rollouts, D_k, using current policy π_k.
5: Compute the rewards-to-go term $\hat{R}_t$.
6: Estimate the advantage function, $\hat{A}_t$, using any general advantage estimator.
7: Calculate the policy gradient based on Equation 1.25.
8: Utilize the conjugate gradient method to calculate:

$$\hat{x_k} \approx \hat{H}_k^{-1} \hat{g}_k$$

9: Use backtracking line search to update the policy:

$$\theta_{k+1} = \theta_k + \alpha^j \sqrt{\frac{2\delta}{\hat{x_k}^T \hat{H}_k \hat{x_k}}} \hat{x_k}$$

where j is the smallest possible integer value that satisfies the KL constraint.
10: Approximate the value function $V_\phi(s_t)$ by mean-squared error:

$$\phi_{k+1} = \arg\min_\phi \frac{1}{|D_k|T} \sum_{\tau\in D_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t\right)^2$$

via a gradient descent algorithm.
11: **end for**

2.1.7 Advantage Actor Critic (A2C)

The Advantage Actor-Critic is a family of algorithms, including Asynchronous Advantage Actor Critic (A3C) and Advantage Actor Critic (A2C), and were developed by the Deepming team at Google (Mnih, Badia, *et al.* 2016). A3C employs multiple parallel actor-learners that asynchronously update a global network, introduced one year after Trust Region Policy Optimization (TRPO). A3C, notable for its asynchronicity, employed **fixed-length episode segments** for both updates and advantage function calculations and it also utilized a shared feature extractor for the value and policy heads. As a highly influential algorithm at its time, A3C underwent extensive benchmarking across various Gym environments. These tests aimed to determine whether its asynchronous nature genuinely enhanced performance or if apparent improvements were due to unrelated factors, such as optimal hyperparameter settings or variability introduced by initialization and seeding techniques. One year after A3C's release, OpenAI conducted a comprehensive benchmark study on A3C, its synchronous counterpart A2C, and the ACKTR algorithms (Y. Wu *et al.* 2017). The study concluded that the asynchronicity and the associated noise did not enhance the performance of A3C. Surprisingly, it was found that A2C, which employs the same architecture but synchronizes **actor-learner** updates to a global network, performed comparably to A3C. Notably, in A2C, the actor-learners wait for each other and update the central network by averaging over the learners, a process that benefits from GPU parallelization. This contrasts with the original design of A3C, which was optimized for running on CPU threads to accelerate training.

The Advantage Actor Critic (A2C) algorithm computes its loss function using the advantage function, necessitating estimates of both state values and state-action values (Equation 1.22). This requirement may seem atypical as it implies the need for separate networks: one to estimate state values and another for action-value estimations.

$$L^{A2C}(\theta) = \hat{\mathbb{E}}_t \left[\log \pi_\theta(a_t|s_t) \hat{A}_t \right] \tag{2.5}$$

From (Mnih, Badia, *et al.* 2016; Equation 2.5), it is clear that the advantage function, $\hat{A}_t$ is simply an approximation of the actual true advantage. However, this can be reformulated to include only the value functions, using the Bellman Equations (Equation 1.19). According to these equations, the expected return of taking action a_t in state s_t can be broken down into the immediate reward r_{t+1} plus the discounted future returns, as estimated by the value function V_ϕ^π at the subsequent state $s(t+1)$ (Mnih, Badia, *et al.* 2016; Equation 2.6).

$$\hat{A}^\pi(s_t, a_t) = r(s_{t+1}, a_{t+1}) + \gamma V_\phi^\pi(s_{t+1}) - V_\phi^\pi(s_t) \tag{2.6}$$

The seminal A2C paper also introduced n-step Actor-Critic learning, which updates both the policy and the value function using an **n-step lookahead**. This method complicates advantage estimation by requiring the calculation of the Temporal Difference **(TD) target**, $r(s_{t+1}, a_{t+1}) + \gamma V_\phi^\pi(s_{t+1})$, across n steps, from which the value of the state, at timestep t, under policy π, parameterized by ϕ is subtracted (Mnih, Badia, *et al.* 2016; Equation 2.7).

$$\hat{A}^\pi(s_t, a_t) = \underbrace{\sum_{i=0}^{n-1} \left(\gamma^i r(s_{t+i}, a_{t+i}) + \gamma^n V_\phi^\pi(s_{t+n}) \right)}_{\text{n-step return}} - V_\phi^\pi(s_t) \tag{2.7}$$

In the official paper, the value head of the network, characterized as an n-to-1 linear projection connected to a CNN backbone is updated by minimizing the squared difference between the

bootstrapped n-step return and the predicted value, which helps approximate $V_\phi(s_t)$ (Mnih, Badia, *et al.* 2016; Equation 2.8).

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{2} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2 \tag{2.8}$$

Additionally, as a method of regularization, entropy is added to the policy π to enhance exploration rates, a technique shown to improve performance during exploration phases (Williams & Peng 1991).

$$\nabla_\theta L^{A2C}(\theta, \phi) = \hat{\mathbb{E}}_t \left[\nabla_\theta \log \pi_\theta(a_t|s_t)(R_t - V_\phi(s_t)) + \underbrace{\beta \nabla_\theta H(\pi_\theta(s_t))}_{\text{entropy}} \right] \tag{2.9}$$

As detailed in Mnih, Badia, *et al.* 2016; Equation 2.7, the final gradient calculation for the policy head enables us to optimize of parameters of the policy, θ, using various techniques, including SGD, RMSProp or Adam.

Algorithm 2 Advantage Actor Critic (A2C) - One Actor (Mnih, Badia, *et al.* 2016)

Input: Initial parameters for both the value, ϕ_0, and policy functions, θ_0.
Hyperparameters: Number of lookahead steps, n, the strength of the entropy, β, and the global shared counter, T.
repeat
    Set back gradients to 0.
    Synchronize thread-specific parameters for correct global updates.
    Set thread step counter, t, to 1.
    repeat
        Perform action according to current policy $\pi_\theta(a_t \mid s_t)$ and receive reward r_t
        $t = t + 1$
        $T = T + 1$
    until current state is terminal or t_{max} is reached.
    if state is terminal **then**
        $R = 0$
    else
        $R = V_\phi(s_t)$
    end if
    for $i = \{t-1, \ldots, t_{\text{start}}\}$ **do**
        $R = r_i + \gamma R$
        Accumulate gradients using Equation 2.8.
        Accumulate gradients of Equation 2.9 using SGD or RMSProp.
    end for
    Perform asynchronous update of both θ and ϕ.
until $T > T_{\text{max}}$

Interestingly, Proximal Policy Optimization (PPO) performs similarly to A2C when limited to a single training epoch (S. Huang, Kanervisto, *et al.* 2022). In this setting, $\pi_{\theta_{old}}$ and π_θ remain unchanged during the epoch, which means the clipping operation, a key feature in PPO's loss calculation, does not fire. As a result, PPO's loss function becomes identical to that of A2C,

leading to the same gradient if the data input is the same. However, this observation should be taken with a grain of salt. In practical implementations, achieving perfect determinism in the code to ensure log ratios equal one is impossible.

2.1.8 Proximal Policy Optimization (PPO)

Proximal Policy Optimization extends the principles established by TRPO, with the aim of maximizing the magnitude of policy updates while ensuring they remain under certain controlled limits (J. Schulman, Wolski, *et al.* 2017). PPO distinguishes itself from TRPO by moving away from the use of hard KL-divergence constraints and second-order approximation methods, introducing two new approaches for computing policy gradients instead. The first variant, **PPO-Penalty**, incorporates a dynamically adjusted KL-divergence penalty coefficient into the objective function, deviating from TRPO's fixed threshold approach. However, despite its flexibility, PPO-Penalty has not demonstrated any performance improvements over TRPO and, thus, will not be further discussed. In subsequent references to PPO, we will specifically refer to the more performant variant, **PPO-Clip**. PPO-Clip lacks a KL-divergence term in its objective function and employs a **clipping mechanism** that effectively limits the distance between old and updated policies.

PPO-Clip updates its policy using a formulation similar to that in Equation 2.2, where the loss function employs a surrogate advantage function represented by the ratio $r_t(\theta, \theta_k)$. This ratio, commonly referred to as the importance sampling ratio in the literature, measures the improvement of the current updated policy over the old policy. By applying this formulation to the TRPO loss function, (J. Schulman, Wolski, *et al.* 2017; Equation 2.3), we get the following equation:

$$L^{PPO}(\theta_k, \theta) = \hat{\mathbb{E}}_{s,a\sim\pi_{\theta_k}}\left[r_t(\theta)\hat{A}^{\pi_{\theta_k}}(s,a)\right] \tag{2.10}$$

Without constraints, this loss function results in excessively large policy updates. To address this, PPO-Clip incorporates a **clipping mechanism** on the probability ratio (J. Schulman, Wolski, *et al.* 2017; Equation 2.11) [6].

$$L^{PPO}(\theta_k, \theta) = \hat{\mathbb{E}}_{s,a\sim\pi_{\theta_k}}\left[\min(r_t(\theta, \theta_k)\hat{A}^{\pi_{\theta_k}}(s,a),\ \text{clip}(r_t(\theta, \theta_k), 1-\varepsilon, 1+\varepsilon)\hat{A}^{\pi_{\theta_k}}(s,a)\right] \tag{2.11}$$

This complex- and daunting-looking loss function is surprisingly simple while effectively limiting the magnitude of policy updates in PPO-Clip. The first term within the empirical estimate is derived from the original loss functions used in both PPO and TRPO (J. Schulman, Wolski, *et al.* 2017; Equation 2.10), incorporating a modification to the surrogate objective function. The clipping mechanism limits the importance sampling ratio to the interval $[1-\varepsilon, 1+\varepsilon]$. If the ratio exceeds these bounds, it is clipped and then multiplied by the advantage estimate, calculated using methods from TRPO and A2C that involve the Bellman Equations and n-step returns (Equation 2.7). The min operation at the end establishes a conservative lower bound on the unclipped objective, effectively ignoring improvements in the probability ratio that would enhance the objective and only incorporating changes that would degrade it. There is a more intuitive approach to the loss function that simplifies the concept and helps build an understanding of the clipping mechanism (Achiam 2018; J. Schulman, Wolski, *et al.* 2017; Equation 2.12):

[6] It should be noted that the final objective function of PPO also incorporates an entropy term, $H(\pi_\theta(s_t))$, to increase exploration.

$$L^{PPO}(\theta_k,\theta) = \hat{\mathbb{E}}_{s,a\sim\pi_{\theta_k}}\left[\min(r_t(\theta,\theta_k)\hat{A}^{\pi_{\theta_k}}(s,a),\, g(\varepsilon,\hat{A}^{\pi_{\theta_k}}(s,a))\right], \tag{2.12}$$

where g takes the following form (Achiam 2018; J. Schulman, Wolski, *et al.* 2017; Equation 2.13):

$$g(\varepsilon,\hat{A}) = \begin{cases} (1+\varepsilon)\hat{A} & \text{if } \hat{A} \geq 0 \\ (1-\varepsilon)\hat{A} & \text{if } \hat{A} < 0 \end{cases} \tag{2.13}$$

When examining the case where the advantage estimate is positive, the term g becomes $(1+\varepsilon)\hat{A}$, suggesting that the advantage supports a positive direction in the surrogate objective. Since $\hat{A}$ appears in both terms within the **min** function, we can factor it out, simplifying the expression to: $\min(r_t(\theta,\theta_k),(1+\varepsilon))$. This indicates that if $\pi_\theta(a|s)$ exceeds $(1+\varepsilon)\pi_{\theta_k}(a|s)$, the **min** function fires, adding a ceiling of $(1+\varepsilon)\hat{A}$ on the policy update. Similarly, if the advantage estimate $\hat{A}$ is negative, the objective will only increase if the selected action becomes less likely in the future, meaning a decrease in $\pi_\theta(a|s)$. When factoring out the negative advantage estimate, care must be taken due to the negative sign inside $(1-\varepsilon)$. This detail converts the **min** function to a **max** function. This change imposes a ceiling of $(1-\varepsilon)\hat{A}$ on the policy updates, effectively limiting how much the policy can adjust in response to actions that degrade the final objective.

Surprisingly, heavily adopted implementations of PPO integrate both clipping mechanisms and KL-Divergence thresholds to maintain controlled policy updates (S. Huang, Dossa, Raffin, *et al.* 2022). It has been demonstrated in various instances that clipping alone is insufficient to limit excessive updates to the model (Ernestus *et al.* 2019). If the KL-Divergence exceeds a predefined limit, the update process is stopped early. In practice, PPO utilizes **fixed-length trajectory segments**, where a single learner collects a specified number of steps (N) in multiple environments (M) before proceeding to update the model. This structured approach categorizes the learning process into two parts: a **rollout phase** and a **learning phase**. For updates, PPO utilizes **minibatches** randomly sampled from the rollout batch, allowing certain steps to be sampled multiple times. Additional implementation details, not specified in the seminal paper, include the necessity for clipping the value function as well (Engstrom *et al.* 2020) and the application of **clip range annealing**, similar to learning rate annealing.

2.1.9 Recurrent Proximal Policy Optimization (R-PPO)

R-PPO (Pleines *et al.* 2022; Raffin *et al.* 2021), a recurrent variant of the Proximal Policy Optimization (PPO) algorithm, integrates **LSTM layers** to establish a recurrent policy framework. This enables the policy to consider not only the current state but also the historical information from the agent's past interactions, encapsulated in the hidden state h_t at each timestep. This adaptation makes the action selection at each timestep dependent on both the state s_t and h_t. This architecture is particularly advantageous in environments characterized by partial observability, or **Partially Observable Markov Decision Processes (POMDPs)**, where the recurrent nature allows the policy to access a more extensive portion of the observational history from an experience replay. Additionally, R-PPO is well-suited for longer or continuous tasks that are not episodic, as recurrence can capture **temporal dependencies** across extended periods. Recurrence was not initially introduced in policy gradient methods but was first explored in Deep Q-Networks (DQN) (Andrychowicz *et al.* 2021). These networks experimented with the ways in which episodes and their corresponding hidden states are sampled from an experience replay buffer.

Algorithm 3 PPO-Clip (J. Schulman, Wolski, *et al.* 2017)

Input: Initial parameters for both the value, ϕ_0, and policy functions, θ_0.
Hyperparameters: KL-divergence threshold, δ, and the clip threshold, ε.
for $k = 0, 1, 2, \ldots$ **do**
    Collect rollouts, D_k, using current policy π_k.
    Compute the rewards-to-go term $\hat{R}_t$.
    Estimate the advantage function, $\hat{A}_t$, using any general advantage estimator.
    Update the policy by maximizing the clipped PPO objective:

$$\theta_{k+1} = \arg\max_{\theta} \frac{1}{|D_k|T} \sum_{\tau \in D_k} \sum_{t=0}^{T} \min\left(r_t(\theta, \theta_k)\hat{A}^{\pi_{\theta_k}}(s_t, a_t),\ g(\varepsilon, \hat{A}^{\pi_{\theta_k}}(s_t, a_t))\right),$$

    using optimization techniques such as SGD, RMSProp, or Adam.
    Approximate the value function $V_\phi(s_t)$ by mean-squared error:

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|D_k|T} \sum_{\tau \in D_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t\right)^2$$

    via a gradient descent algorithm.
end for

In terms of implementation, most approaches (S. Huang, Dossa, Raffin, *et al.* 2022) employ a convolutional neural network backbone linked to an LSTM or a dual-stream transformer. The outputs are then processed through either shared or separate feature extractors for the value and policy heads of the PPO. Some implementations also feature a shared linear layer before diverging into two distinct extractors. The loss functions of PPO and its recurrent variant, R-PPO, are largely similar, with a key difference in the importance sampling ratio. In R-PPO, both π_θ and π_{θ_k} depend on the hidden state of the recurrent layer, expressed as: $\pi_\theta(a_t|s_t, h_t)$ (Pleines *et al.* 2022; Equation 2.14).

$$r_t(\theta, \theta_k) = \frac{\pi_\theta(a_t|s_t, h_t)}{\pi_{\theta_k}(a_t|s_t, h_t)} \tag{2.14}$$

Incorporating LSTM layers into a model significantly modifies the processing of rollout data, modifies the loss function (Pleines *et al.* 2022; Equation 2.14), and requires adjustments to the forward pass. Unlike the original PPO, which processes minibatches of potentially truncated data from the rollout buffer due to predetermined trajectory lengths, LSTM necessitates orderly processing of sequences to maintain temporal dependencies (S. Huang, Dossa, Raffin, *et al.* 2022). This requires that rollout data be segmented into episodes, which are then divided into smaller sequences of fixed lengths. Given that episode lengths are different, maintaining a fixed **sequence length**, unless it matches the lengths of episodes, can lead to padding shorter episodes with many zero-filled steps. This padding introduces substantial noise into the LSTM's input sequences, complicating the learning process and necessitating a masking procedure for loss calculation.

Correctly managing the hidden state is crucial when incorporating LSTMs into a model. The output hidden state from one sequence must serve as the input hidden state for the subsequent sequence, a process known as **truncated backpropagation through time** (Tallec & Ollivier 2017). Additionally, it is more efficient to process the entire batch of data through the network's

backbone before segmenting it for the LSTM layers. After processing through the recurrent layers, the data must be reshaped once again to feed into the final linear projection layers for both the policy and value heads.

There is an ongoing debate about the benefits of refreshing advantages and hidden states before each minibatch epoch in policy gradient methods. Research indicates that recalculating advantage estimates prior to each epoch can significantly enhance the performance of these methods (Andrychowicz *et al.* 2021). Similarly, studies suggest that updating hidden states before collecting experiences and recalculating them before each minibatch epoch can notably improve the performance of R-PPO (Kapturowski *et al.* 2019; Gupta 2021). However, other sources contend that recalculating advantages before every minibatch epoch does not yield substantial improvements (Pleines *et al.* 2022).

2.1.10 Masked Proximal Policy Optimization (M-PPO)

Action masking is a significantly underutilized tool in Reinforcement Learning that enables a learning agent to avoid sampling invalid actions from the action space distribution (S. Huang & Ontañón 2022). These invalid actions, determined by the constraints of the environment, can significantly reduce the effective size of the action space. For example, OpenAI's Dota AI (OpenAI *et al.* 2019) operates within an action space that has over 1.8 million dimensions, where a substantial number of actions—such as moving to invalid spaces, buying items without sufficient in-game currency, or using spells without the required resources—are infeasible. Action masking can effectively reduce this to a small fraction. Despite its simplicity, action masking is rarely discussed in academic literature, with only a few seminal papers addressing it (Vinyals *et al.* 2017; Johnson *et al.* 2016). Invalid action masking produces a valid policy gradient in policy optimization methods.

Invalid action masking can be implemented through various methods, each differing in scalability and effectiveness (Achiam 2018). One approach is to assign **negative penalties** to invalid actions, which can exponentially increase the hyperparameter space with the number of invalid actions. This method typically yields unfeasible reward shaping requirements and is generally not recommended (Hou *et al.* 2023; Dietterich 1999). More efficient techniques include **true** and **naive invalid action masking**, which directly prevent the selection of infeasible actions, improving the sample efficiency of an RL algorithm.

The former operates by modifying the output logits of infeasible actions before converting these logits into a probability distribution from which actions are sampled. Specifically, the logits for invalid actions are set to an extremely large negative value (Figure 2.5), effectively causing the softmax function to assign a near-zero probability to these actions (S. Huang & Ontañón 2022). However, this approach should be approached with caution due to potential issues with seeding and different implementations of categorical distributions in PyTorch or Tensorflow and inherent numerical uncertainties, which can result in this process not being entirely deterministic in masking out actions. The naive invalid action masking method employs a renormalized policy that ensures no invalid action can be chosen. However, it updates the gradient based on the unnormalized policy, which still includes the invalid actions. This method is termed "naive" because it does not alter the logit values of invalid actions; instead, it simply renormalizes the probabilities across the valid actions in the policy.

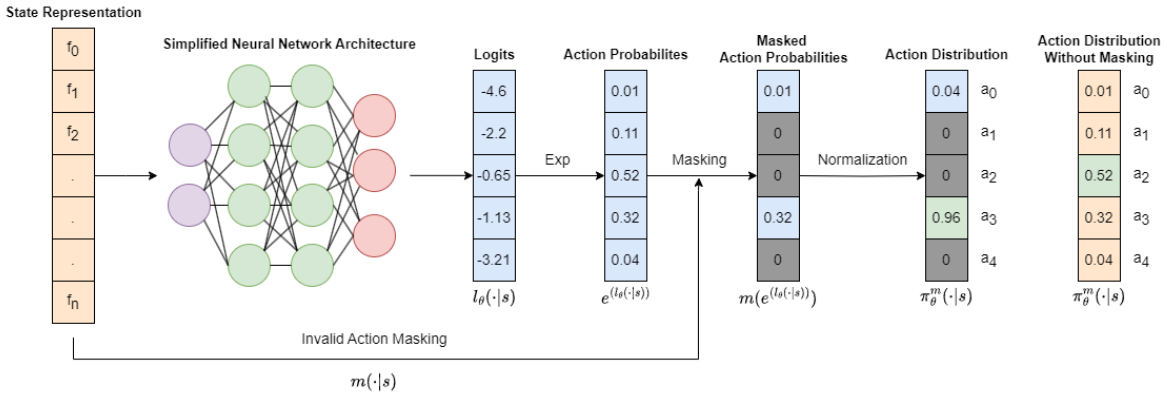


Figure 2.5: A high-level overview of how invalid action masking affects the final action distribution. Note that logits here represent the log probabilities of actions and are sent through a masking procedure where invalid actions are filtered out. After masking, exponentiation is applied to convert these log probabilities back into raw probabilities. Subsequently, normalization is performed to transform these probabilities into a valid probability distribution.

Studies have demonstrated that both true and naive invalid action masking methods perform comparably up to a certain point, with minor differences in convergence times (S. Huang & Ontañón 2022). Convergence is context-specific, typically defined by the learning agent consistently reaching a predetermined performance threshold. However, naive invalid action masking has some drawbacks, such as susceptibility to KL divergence explosion, which can result in significantly larger policy updates compared to true invalid action masking. Additionally, naive invalid action masking is sensitive to increases in environment size; for example, scaling up the size of a grid world map can significantly impair performance.

Now that we have explored the intricacies behind action masking, what exactly is M-PPO, and how does it function in practice? Simply put, M-PPO modifies the standard PPO algorithm by incorporating **domain knowledge** into action selection, thus guiding gradient updates faster towards a global optimum. M-PPO is essentially a straightforward extension of the original PPO algorithm. At first glance, it may seem like a "cheat code," and indeed, it could potentially be used as such. Theoretically, one could build the heuristics behind the invalid action masking within a system to render the algorithm completely deterministic, adhering to a **logic-based approach**. However, such modifications are purely hypothetical and would likely hold little practical value or relevance in real-world applications.

2.1.11 Deep Q Network (DQN)

The fundamental concept behind Deep Q-Networks (DQNs) builds on the principles of classical Q-Learning, aiming to estimate the action-value function using the Bellman Equations to ensure convergence through value iteration (Mnih, Kavukcuoglu, *et al.* 2013; Raffin *et al.* 2021). However, this approach often struggles in practice due to issues like scalability and limited generalization. Value iteration and Q updates, which rely on sequences of actions in the environment, introduce a temporal sequence bias. This bias persists even in function approximators, whether they are linear or neural networks, that attempt to estimate the action-value function through parameterized approximations. DQNs are trained using neural network function approximators by minimizing a specific loss function (Mnih, Kavukcuoglu, *et al.* 2013; Equation 2.15).

$$L^{DQN}(\theta_k, \theta) = \mathbb{E}_{s,a\sim\pi_{\theta_k}}\left[\left(\mathbb{E}_{s'\sim\varepsilon}\left[r + \gamma \max_{a'} Q(s', a'|\theta_k)\right] - Q(s, a|\theta)\right)^2\right] \tag{2.15}$$

In Deep Q-Networks (DQNs), the target value within the loss function, which might seem counterintuitive in classical supervised learning settings where targets are predefined, is dynamically determined. The target for a DQN is composed of the immediate reward received from taking action a in state s plus the discounted future rewards, calculated using the latest frozen network parameters while taking the greedy action. This creates the **target Q-value**. The loss function then measures the squared difference between the **predicted Q-value** by the network and this target Q-value, effectively guiding the network's learning process.

Here, it is necessary to introduce how DQNs address the temporal bias in Q-Learning by incorporating an experience replay buffer (Fedus *et al.* 2020). This buffer stores agent experiences at each timestep as $e_t = (s_t, a_t, r_t, s_{t+1})$ in a dataset, denoted D. To combat sequence bias and improve generalization, DQNs update Q-functions using minibatches randomly sampled from these stored experiences. Additionally, a new replay buffer has been introduced called the **Prioritized Replay**, which enhances this mechanism by selectively sampling experiences where there is a significant distance between actual and expected rewards, thus correcting the inaccurate reward assumptions of the model.

Usually, like with any other state-of-the-art algorithm, DQNs optimize the loss function in minibatches from experience replay, thus using stochastic gradient descent methods. Q-learning is **off policy**, meaning it follows and **e-greedy behavior policy** and learns about the greedy policy. This makes the gradient of the loss as the following (Mnih, Kavukcuoglu, *et al.* 2013; Equation 2.16):

$$\nabla_\theta L^{DQN}(\theta_k, \theta) = \mathbb{E}_{s,a\sim\pi_{\theta_k};s'\sim\varepsilon}\left[\left(r + \gamma \max_{a'} Q(s', a'|\theta_k) - Q(s, a|\theta)\right)\nabla_\theta Q(s, a|\theta)\right]. \tag{2.16}$$

The team behind DeepMind and the seminal paper did not provide a detailed description of the hyperparameters used nor any substantial information on hyperparameter annealing or detailed, statistically significant evaluation metrics. Instead, they relied on reward as the primary performance metric during evaluation phases (Mnih, Kavukcuoglu, *et al.* 2013). However, using reward as a metric can be problematic since the reward system itself is a hyperparameter. It is susceptible to abrupt changes and large deviations that may not accurately reflect the agent's actual performance.

OpenAI has found that **epsilon annealing** is essential for early exploration and later exploitation in the training processes (Dhariwal *et al.* 2017). Additionally, DQNs should use a version of **Huber Loss** (Wikipedia 2024b), a smoothed mean-squared error loss that clips the multiplicative term during gradient computation, a method that closely resembles the regularization strategies utilized in TRPO and PPO through setting thresholds or clipping gradients to prevent excessive updates. Deep Q Networks led to the development of two advanced variants, Double Q Learning and Dueling DQNs. Although these versions build on the original DQN framework, our testing focused solely on the original algorithm, and therefore, we will not discuss these variants further.

Algorithm 4 Deep Q-learning with Experience Replay (Mnih, Kavukcuoglu, *et al.* 2013)

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights
for episode $= 1$ to M **do**
    Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
    for $t = 1$ to T **do**
        With probability ε select a random action a_t
        otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
        Execute action a_t in emulator and observe reward r_t and image x_{t+1}
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
        Set $y_j = r_j$ for terminal ϕ_{j+1}
        Set $y_j = r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta)$ for non-terminal ϕ_{j+1}
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$
    end for
end for

2.1.12 Quantile Regression Deep Q Network (QR-DQN)

Quantile regression introduces a distributional approach to DQNs, shifting from modeling the mean or expected return in a supervised setting to learning the **full distribution of values** (Dabney *et al.* 2017; Raffin *et al.* 2021). This method explicitly models a distribution over returns rather than merely estimating the mean. In the seminal paper, the authors represented the returns as a random variable Z, whose expected value corresponds to the Q-value. In their research, the authors demonstrated that for a fixed policy, the Bellman operator applied to a value distribution is a contraction in the maximal form of the **Wasserstein distance**. This led to the development of the algorithm known as **C51** (Bellemare *et al.* 2017). However, employing the Wasserstein metric in loss calculations has proven to be practically infeasible because stochastic gradient methods cannot be directly applied. To overcome parts of this, C51 approximates the distribution by parameterizing N fixed locations and adjusting the probabilities at these points to approximate the true distribution of the random variable.

QR-DQN fixes the above approach by constraining the probabilities to a uniform distribution and adjusting only the locations, or **quantiles**, of the random variable Z. These quantiles act as dividing points within the range of the probability distribution over returns. Like its predecessor, QR-DQN employs the Wasserstein distance (Dabney *et al.* 2017). It stochastically adjusts the locations of these quantiles to minimize the Wasserstein distance between the approximate distribution and the target distribution.

To parameterize a model using parameters θ and a quantile distribution Z_θ, each action pair is mapped to a uniform distribution supported on $\theta(x,a)$. This setup results in the following notation (Dabney *et al.* 2017; Equation 2.17):

$$Z_\theta(x,a) = \frac{1}{N} \sum_{i=1}^{N} \underbrace{\delta_{\theta_i}(x,a)}_{\text{Dirac}}. \tag{2.17}$$

In the context of QR-DQN, when the Bellman update is projected onto the approximation space, it typically loses its contractive properties. However, it was demonstrated that the distributional

Bellman update, when projected onto a parameterized quantile distribution in QR-DQN, retains its **contraction characteristics** (Bellemare *et al.* 2017). Unlike traditional DQN that uses Huber loss, QR-DQN adapts this to **Quantile Huber** loss to accommodate the distributional approach. The original Q-function update is modified by employing the distributional Bellman optimality operator, resulting in the updated formulation as follows (Dabney *et al.* 2017; Equation 2.18):

$$\mathrm{T}Z(x,a) = R(x,a) + \gamma Z(x',a'), \quad x' \sim P(\cdot|x,a), \quad a' = \arg\max_{a'} \mathbb{E}_{z\sim Z(x',a')}[z]\,. \tag{2.18}$$

In QR-DQN, the action selected for the next state is determined by the mean of the next state-action value distribution. In terms of neural network architecture, compared to traditional DQN, the primary modification required is the resizing of the output layers to accommodate $|A| \times N$, where $|A|$ represents the number of actions and N represents the number of quantiles, which is a hyperparameter. Additionally, the loss function is updated from the standard Huber loss to a Quantile Huber loss to reflect the distributional approach (Yang *et al.* 2020).

2.1.13 Augmented Random Search (ARS)

Augmented Random Search (ARS) addresses the **reproducibility crisis** in reinforcement learning, which has seen models become increasingly sample-efficient yet complex and challenging to replicate (Mania *et al.* 2018; Raffin *et al.* 2021). ARS simplifies this approach by being a straightforward policy parameter search algorithm that incorporates basic techniques to enhance its effectiveness in control tasks. It is designed to overcome challenges such as initialization issues and sensitivity to random seeding (Henderson *et al.* 2019), making it a robust alternative to more complex algorithms.

Augmented Random Search (ARS) is a **derivation-free** algorithm, closely related to evolutionary algorithms in its operation. Each update in ARS is scaled by the standard deviation of the rewards collected during that update step. System states are normalized using only estimates of their mean and standard deviation. Unlike many other reinforcement learning strategies, ARS does not utilize neural networks. Instead, it operates with a simple linear policy (Mania *et al.* 2018).

Basic Random Search (BRS) (Mania *et al.* 2018), the predecessor to Augmented Random Search (ARS), explores the parameter space rather than the action space. BRS selects a random direction on the sphere within the parameter space and optimizes the function along this chosen direction. The updates directions are calculated as follows (Mania *et al.* 2018; Equation 2.19):

$$\frac{r(\pi_{\theta+\nu\delta},\xi_1) - r(\pi_{\theta-\nu\delta},\xi_2)}{\nu}, \tag{2.19}$$

Where ν is a positive real number and δ is a zero-mean Gaussian, the update function in Basic Random Search (BRS) is unbiased with respect to the parameters when ν is small. Additionally, using minibatch updates can also reduce variance. This approach results in a smoothed objective function. Augmented Random Search enhances BRS by implementing larger updates to the parameters, scaling the steps with the standard deviation of the rewards collected in each iteration. ARS also improves efficiency by discarding **perturbation directions** that yield the least improvement in reward.

Algorithm 5 Augmented Random Search (Mania *et al.* 2018)

Hyperparameters: step-size α, number of directions sampled per iteration N, standard deviation of exploration noise ν, number of top-performing directions to use b.
Initialize: Set M_0, μ_0, $\Sigma_0 = I_n \in \mathbb{R}^{n\times n}$ and $j = 0$
while ending condition not satisfied **do**
    Sample $\delta_1, \delta_2, \ldots, \delta_N$ in $\mathbb{R}^{p\times n}$ with i.i.d. standard normal entries.
    Collect $2N$ rollouts of horizon H and their corresponding rewards using the $2N$ policies:

$$\text{V1:} \begin{cases} \pi_{j,k,+}(x) = (M_j + \nu\delta_k)x \\ \pi_{j,k,-}(x) = (M_j - \nu\delta_k)x \end{cases} \qquad \text{V2:} \begin{cases} \pi_{j,k,+}(x) = (M_j + \nu\delta_k)x\ \mathrm{diag}(\Sigma_j)^{-1/2}(x - \mu_j) \\ \pi_{j,k,-}(x) = (M_j - \nu\delta_k)x\ \mathrm{diag}(\Sigma_j)^{-1/2}(x - \mu_j) \end{cases}$$

    Sort the directions δ_k by $\max\{r(\pi_{j,k,+}), r(\pi_{j,k,-})\}$ and by $\pi_{j,(k),+}$ and $\pi_{j,(k),-}$ the corresponding policies.
    Make the update step:

$$M_{j+1} = M_j + \frac{\alpha}{b\sigma_R} \sum_{k=1}^{b} \left[r(\pi_{j,(k),+}) - r(\pi_{j,(k),-}) \right] \delta_{(k)}$$

    Set μ_{j+1}, Σ_{j+1} to be the mean and covariance of the $2NH(j+1)$ states encountered from the start of training.
    $j \leftarrow j + 1$
end while

Since these algorithms are designed to work with linear policies, they are primarily suited for **control tasks**. However, we sought to assess their performance in more complex environments, such as Lux, particularly in discrete problem settings. Our findings support the assertion made in the seminal paper that these algorithms do not perform well in discrete tasks or within more complex environmental representations.

2.1.14 Training

We ensured uniformity across all training environments for the algorithms discussed by adhering to the following paradigms:

- Each algorithm was tested across **three trials**, training each from scratch.
- A **specific seed** was manually set for all algorithms and the SB3 torch configuration to maintain consistency across runs [7].
- We utilized 16 parallel environments running on threads, a choice dictated by the limitations of our local hardware, which consists of 8 physical cores with Intel's hyperthreading, allowing for 16 logical cores.
- Each environment was configured with a **fixed rollout size of 4000** steps, and each episode was permitted to extend up to 1000 steps without any truncation. This setup resulted in a total of $4000 \times 16 = 64.000$ rollout steps, after which the model transitioned into the training phase.
- The enemy player was configured as a passive entity without decision-making capabilities; their water count was set to a high level to prevent their factory from being destroyed.

Consequently, during each training step, the model was updated using data from only one team.
- The predefined hyperparameter sets from the original stable baselines repository were used for all algorithms. We adjusted the batch size to 800 for all algorithms except ARS (which does not use batch updates) to facilitate larger updates. Furthermore, the number of steps per environment per update was standardized to 50 for all policy gradient methods.
- A shared feature extractor architecture was employed across all algorithms, consisting of an MLP with two linear layers of size 128 and two ReLU activation layers.
- All training sessions were conducted within an isolated Docker environment, ensuring that memory usage did not exceed 8GB in any case.
- All trials were conducted up to a total of **3.000.000 steps**.

2.1.15 Evaluation

Our objective in evaluating these novel algorithms was to select one of the eight for scaling up and further assessments across various agent control levels. We aimed to identify an algorithm that aligns best with our specific problem, offers quick performance to speed up research progress, and delivers promising metrics in our benchmark study. The algorithms underwent continuous online evaluation, with the agent interacting with the environment in alternating training and evaluation phases. The criteria for the evaluation phases were as follows:

- Every 24k steps, four parallel environments were instantiated with a **limit of 1000 maximum episode steps**, the usual game length in the Lux environment.
- After each evaluation phase, the model state was saved. If it performed better than the current best, the evaluation step was updated and saved.
- The same stochastic policy used during the training phase was employed for the evaluation phase, meaning that actions were chosen probabilistically rather than greedily at each step.
- Five evaluation episodes were conducted at every phase, resulting in 4×5 full runs of evaluation episodes to 1000 maximum episode steps.
- **Quantitative evaluations** utilized metrics that assessed the agent's performance in the environment, such as *icedug* and *waterproduced*, and how quickly the model reached an average episode length of 1000. This was considered the convergence threshold, indicating the agent had adapted an optimal policy where a single heavy unit could sustain the factory for 1000 steps. These metrics were calculated as the average over the 4×5 environments.
- The final quantitative results were averaged over all three trials, meaning the plotted results represent an average over $3 \times 4 \times 5$ evaluation episodes at every 24.0000 step interval.
- Speed (**step per second**) was used as an auxiliary metric to measure what could be expected from the algorithms if scaled up.

Our analysis showed conclusive results, identifying Masked-PPO as the most effective algorithm across several metrics: it achieved convergence in the fewest steps, exhibited the fastest

[7] The three trial seeds used across training were: **42, 69** and **999**

performance among the algorithms that actually converged, and its application in multi-agent tasks has been widely documented. Its design allows for straightforward CPU and GPU parallelization due to fixed rollout lengths and minibatch updates. Consequently, Masked Proximal Policy Optimization (M-PPO) will be the primary focus of our subsequent research, while other algorithms may be revisited in future studies.

2.2 Multi Agent Environment

Having **more than one non-heuristically controlled entity** brings up the question: What do we call an agent? The team of entities, or the entities that make up that team? The Lux Environment (section 1.4) presents a unique challenge in multi-agent control due to its diverse entity types: factories and units. These entities have distinct action spaces and characteristics. Units can be further categorized into light and heavy units, each with different strengths and weaknesses. Units are created by the factories, which further complicates the situation since the **number of entities depends on the entities' actions** themselves, and these numbers are **constantly changing**.

In our work, we explored various agent-control approaches, mainly **centralized** architectures, where a single pass through the model is responsible for selecting actions for the entire team, turning the multi-agent scenario into a single-agent problem where each controllable entity becomes part of a complex composite action space. One of such implementations is our **monolithic approach** (section 2.3), where the agent makes decisions for every entity based on a global observation. However, as we will later discuss, this method requires the agent to **interpret the entire game state**, which is particularly challenging in highly dynamic environments such as Lux. The complexity of this environment highlights the limitations of existing approaches and the need for a new, more adaptable solution.

Instead of using a single pass through a centralized model, another option would be to generate actions for every entity individually, based on their own observations. Generating a single entity's action based on a much smaller feature set reduces the required complexity of the network. Furthermore, we can treat each entity as a **separate, parallel environment trajectory**, increasing the train examples collected from one step of the environment. Solutions for PettingZoo (J. Terry *et al.* 2021) multi-agent environments have been successfully trained with similar approaches using SuperSuit (J. K. Terry *et al.* 2020). However, **changing agent numbers** poses a problem for such implementations. In our case, factories can explode throughout the game, and units are frequently destroyed or created, making existing multi-agent training solutions obsolete for our use case because they assume a constant number of agents throughout the episode. Additionally, training with on-policy methods requires **careful tracking of their trajectories** and creates **constantly changing batch sizes** if we treat every agent as a separate observed step. The Lux Environment can have hundreds of units active simultaneously. Storing observations and calculating actions for each unit and factory **hinders performance at both training and inference**. For these reasons, we have not explored full multi-agent control. Instead, we used what we call a **hybrid approach** (section 2.4), which uses a **centralized decision-maker** while allowing **individual control** of units and factories based on both **local and global information**.

2.3 Monolithic Approach

In our monolithic solution, we introduce a **central decision-maker** agent, serving as the exclusive learning entity with oversight over entities that provide information but lack **direct influence** on the whole learning process. This framework enables the central decision-maker to supervise all units on the map concurrently, leveraging a global trajectory and reward system for updates. To tackle the challenge of fluctuating numbers of learning agents (Piccoli 2023; J. K. Terry *et al.* 2020), such as factories or units, we implemented a **single-trajectory approach** for each episode. This method ensures fixed episode lengths, independent of factors like unit deaths or creations, resulting in a singular termination flag at the conclusion of each episode.

In our context, **direct influence** refers to the scenario where the actions of a single entity directly impact the central decision-maker. However, in the case of the monolithic approach, this direct influence does not occur. Instead, the central brain only perceives global changes in the environment without attributing them to specific agents. We refer to this as **indirect influence** on the learning agent. This distinction significantly simplifies the functioning of the learning agent. For instance, when there is a positive change in the environment, such as ice collection, it is reinforced without the need to identify which agent was responsible for the action.

In our approach, we utilize a single actor and a single critic, forming a **single-task learning setup** (Eysenbach *et al.* 2023; Gai & Wang 2024; Mysore *et al.* 2022) that optimizes a unified global objective. The single actor generates actions for every pixel on the map, creating a global action tensor that encompasses the entire map. From the raw observations, we filter out pixels containing units and factories to form a tensor of shape $M \times |A|$, where M is the number of factories and units, and $|A|$ represents the size of the action space. Similarly, the single critic approximates the singular value of the entire game state. This is what we refer to as a **monolithic framework** for PPO: it includes global observations, a single decision maker, a single global trajectory for every episode, a global reward system, a single actor generating a global action tensor, and a single critic evaluating the global state of the game.

2.3.1 Environment

There were minor changes to the environment and learning setups compared to section 2.1. We expanded the action space to include the **recharge action** for units and **three additional factory actions**: light unit building, heavy unit building, and lichen watering. This adjustment increased the original discrete action space (subsection 2.1.2) from 12 to 16. Additionally, we implemented more rigorous action masking, which restricted units from transferring resources to enemy units and factories, prevented out-of-bounds movements, and limited factory unit generation.

We reduced the episode length during the training phase to 256 steps to **fit a broader range of data within the same number of training examples**. This adjustment was driven by the observation that the environment dynamics stabilize after a certain number of steps, especially when the goal is to sustain the factories. Given the relatively straightforward strategy of mining ice and transporting it to the factory, the mechanics remain largely consistent throughout the match. By shortening the episode length during training, we were able to parallelize more environments, resulting in accelerated training.

2.3.2 Heuristic Bidding and Factory Placement

In our experiments, we aimed to **minimize variance between environments** by employing heuristics for the bidding and factory placement phase (subsection 1.4.2). Given that bidding reduces starting resources, leading to faster factory depletion, we consistently bid with zero. To optimize ice supply to factories, we identified locations where the **adjacency of ice to factories is maximal**. **Gaussian filtering** (Wikipedia 2024c) techniques were applied to the board to smooth out data, emphasizing nearby points and reducing noise. Gaussian filtering is a method that smooths out data, like images or, in our case, an observed matrix of the board, by giving more importance to nearby points in a **bell-shaped distribution**, reducing noise and sharp transitions. This effectively causes the values on the board to spread out in an area, indicating how far away a tile is from other relevant tiles. After applying Gaussian filtering, the smoothed boards are weighted and summed pixel-wise to obtain final scores for each tile. Placement selection is then performed **greedily**, with the factory placed on the tile with the highest score among the currently valid placement locations.

2.3.3 Actions

Utilizing a single actor, we compressed our action space into a single discrete space, accommodating both unit and factory actions. To ensure precise probability calculations and to restrict factory actions for units, and vice versa, we **employed invalid action masks**. This essentially led to a theoretical **splitting of the action space** into two categories: factory actions and unit actions. Factory actions were masked out for all units, and unit actions were masked out for all factories, with the exception of the no-operation action, which was available to both entity types. Restricting illegal actions within such vast action spaces is crucial for expedited learning. For instance, by forbidding the transfer action when no cargo is available on the unit or when the unit is not adjacent to another unit, or when the factory is not adjacent to the unit, we effectively reduced the range of possible unit actions from 13 to 8 (including move in 4 cardinal directions, pickup, dig, recharge, and no-operation) (subsection 1.4.4). Although we did not implement a more complex masking system to limit repeated actions or prevent agents from overlapping, as it would introduce excessive logic and determinism into the game, we aimed for the central decision maker to **learn these patterns** rather than explicitly instructing it on all possible negative edge cases. Lichen watering was also excluded for all factories to better align them with the task of preserving their longevity, as watering lichen consumes water, thereby reducing factory lifespan. For global control, our final action space was of size $[16, 48, 48]$, where each pixel on the map had an assigned action vector, as illustrated in Figure 2.12.

2.3.4 Observation

The observation features were also overhauled from the single-unit testbench (subsection 2.1.3), aiming to create a compact observation space while still providing the model with all relevant information. These features can be categorized into four groups: **global features, map features, unit features, and factory features**. Global features consist of scalar values that are tiled to the shape of the map for concatenation with other features. Map, unit, and factory features are all

[8] The seed used to generate the cave-style map was set to 420.

[9] The possible spawn positions are represented by a mask matrix where True values indicate valid spawn positions and False values indicate invalid ones. Invalid positions for factories include those on top of ore blocks, ice blocks, or the very edges of the map. This information is provided and calculated by the Lux AI engine.

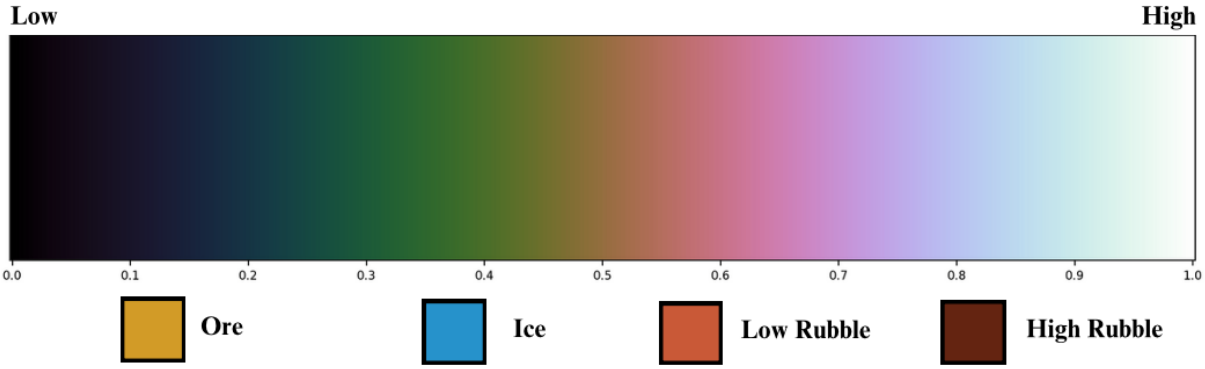


Figure 2.6: The legend image provided below accompanies subsequent images. It illustrates the original starting map of the Lux environment. The gradient color scheme represents the presence of elements, with lighter colors indicating lower presence and darker colors indicating higher presence. The color mapping for ore, ice, and rubble remains consistent with previous representations (Figure 1.3).

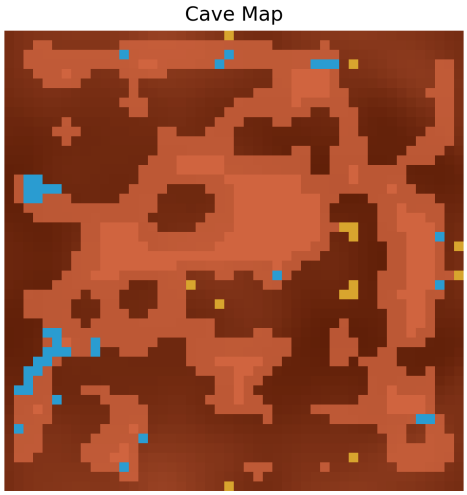


Figure 2.7: Image of the original starting seed of the map without any factories placed on it, generated as the initial step before the bidding phase [8].

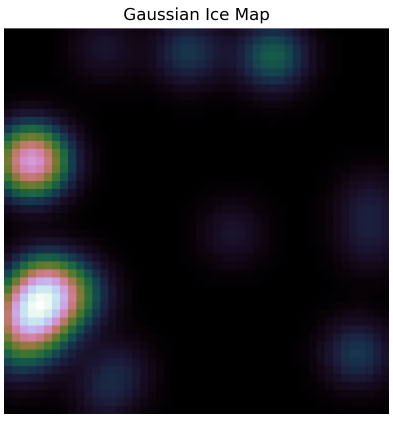


Figure 2.8: High presence of ice calculated using our Gaussian filter. Lighter values indicate high-density ice areas.

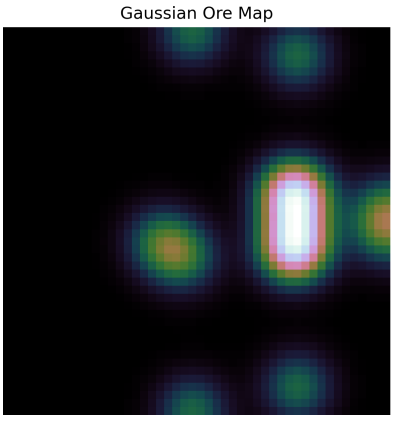


Figure 2.9: High presence of ore calculated using our Gaussian filter. Lighter values indicate high-density ore areas.

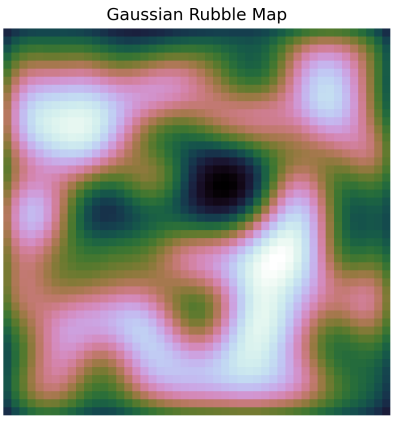


Figure 2.10: High presence of rubble calculated using our Gaussian filter. Lighter values indicate high-density rubble areas.

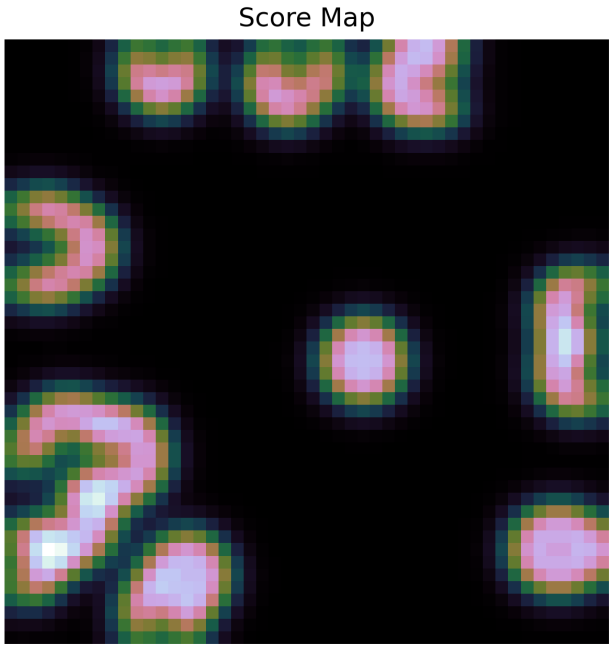


Figure 2.11: The output of the Gaussian filter algorithm applied to the ore, ice, and rubble maps weighted with possible spawn position maps [9]. Lighter areas indicate the best possible spawn locations.

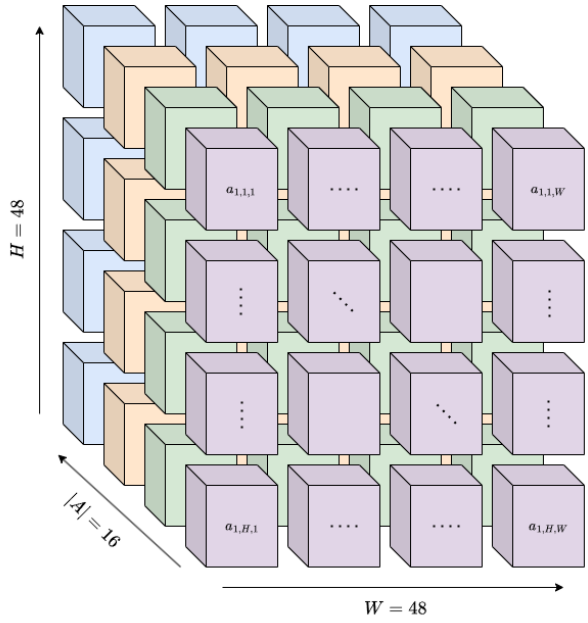


Figure 2.12: A comprehensive representation of an action tensor is depicted on the left, alongside its masked version utilizing invalid action masking. Here, **masked out** refers to actions that are deemed invalid in the environment given a state s. Similarly, colored boxes indicate the same actions across different pixels of the environment. This representation has limitations, as it shows each pixel on a 4×4 grid with 4 possible actions each. From the illustration, it is also visible that invalid action masking effectively reduces the exploration space of the environment for the learner agent.

shaped **H x W**, where **H** represents the game board's height and **W** its width, describing the tile of the board, units, and factories respectively. Features are represented as 32-bit floating-point numbers. Boolean flags are converted to zeros and ones, while other values are normalized to a **[-1, 1] range** to aid convergence. All features are concatenated as channels into a single tensor within the model. See Table 2.1 for a complete list of features.

The model additionally receives two matrices, each cell containing the ID of the unit or factory occupying that position on the map, representing the entity's team affiliation. These matrices do not alter entity behavior but are utilized during training for reward assignment, critic values, log probabilities, and entropy values. Further details on reward assignment will be provided in subsection 2.4.5.

Feature Type	Feature Name	Shape	Value Range
Global	Step	Scalar	[-1, 1]
Global	Daytime	Scalar	{0, 1}
Map	Friendly Factory	H x W	{0, 1}
Map	Ice	H x W	{0, 1}
Map	Ore	H x W	{0, 1}
Map	Rubble	H x W	[-1, 1]
Map	Friendly Unit	H x W	{0, 1}
Map	Enemy Unit or Factory	H x W	{0, 1}
Unit	Heavy Unit	H x W	{0, 1}
Unit	Power in Battery	H x W	[-1, 1]
Unit	Ice in Cargo	H x W	[-1, 1]
Unit	Ore in Cargo	H x W	[-1, 1]
Factory	Power in Factory	H x W	[-1, 1]
Factory	Ice in Factory	H x W	[-1, 1]
Factory	Water in Factory	H x W	[-1, 1]
Factory	Ore in Factory	H x W	[-1, 1]
Factory	Metal in Factory	H x W	[-1, 1]

Table 2.1: Table containing the complete list of observation features, along with their shape and value range.

2.3.5 Residual Network

Residual Networks were developed as a response to the increasing complexity of model depth in image recognition tasks. Progressing from AlexNet (Krizhevsky *et al.* 2012) to VGG16 (Simonyan & Zisserman 2015), GoogLeNet (Szegedy *et al.* 2014), and finally to ResNet-50 (K. He *et al.* 2015a), which has a depth of 50 layers, ResNets introduced a crucial feature called **residual** or **skip connections** (Figure 2.13). These connections allow layers to receive information not only from adjacent layers but also from earlier layers via direct links, creating a parallel path for more effective information flow. Highlighted in the seminal paper by the Microsoft Research team, ResNets, despite their lower complexity relative to VGG-16, achieved an 8x increase in depth with 152 layers and recorded a 3.57% error rate on the ImageNet dataset (K.

He *et al.* 2015a). This architecture's efficiency comes from its ability to enable later layers to learn only the **residual differences** from their inputs rather than the entire function mapping. Specifically, if the desired output $H(x)$ is defined as $H(x) = F(x) + x$, where $F(x)$ represents the residual mapping to be learned, thus the network effectively learning an identity function when $F(x) = 0$, i.e., when there is no difference between x and $H(x)$.

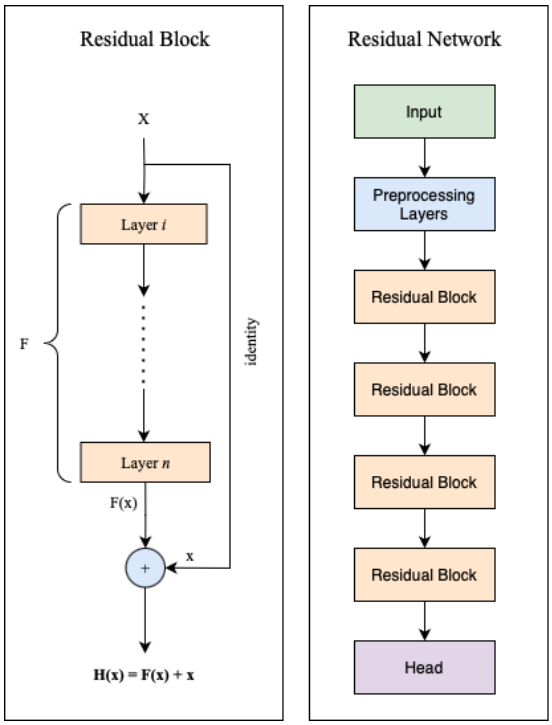


Figure 2.13: A representation of a residual block and its utilization in Residual Networks, which usually include preprocessing layers, which can vary by input type. For images, this often involves convolution layers coupled with max pooling and ReLU activations. The network's head typically consists of a sequence of linear projection layers.

Residual connections address the challenge posed by extremely deep networks, where during the lengthy backpropagation process, the gradients of the loss functions can diminish to the point of vanishing. This premature vanishing effectively halts the learning process. By implementing residual connections, there is improved gradient flow throughout the network, which not only counters the **vanishing gradient problem** but also helps prevent overfitting and deterioration of the model's performance. Residual networks address the gradient vanishing problem by functioning as an ensemble of shallow networks, which helps to avoid the issues associated with gradient explosions rather than directly solving them. This architectural approach is particularly beneficial in reinforcement learning tasks where the environment requires complex representations, enabling deeper networks to learn more effective mappings from inputs to outputs.

2.3.6 Leaky ReLU

Leaky Rectified Linear Unit (Leaky ReLU) (Xu *et al.* 2015) is a variant of the popular Rectified Linear Unit (ReLU) (Agarap 2019) activation function. While ReLU sets negative values to zero, Leaky ReLU **introduces a small, non-zero slope for negative inputs**. Unlike ReLU, the

slope coefficient in Leaky ReLU is pre-defined and remains fixed throughout training, rather than being learned. This activation function is particularly useful in scenarios where **sparse gradients** are a concern, such as training generative adversarial networks or actor-critic networks in reinforcement learning. In these contexts, maintaining non-zero gradients for negative inputs helps prevent the issue of "dying ReLU" (Lu 2020).

By allowing gradients to propagate even for negative inputs, Leaky ReLU effectively prevents them from vanishing, promoting more stable learning. Moreover, it **pairs well with** techniques like **orthogonal weight initialization**. It's important to note that the choice of activation function and weight initialization method can significantly impact the training dynamics and performance of neural networks. Therefore, careful consideration of these factors is essential for achieving optimal results in various tasks and domains (Datta 2020; S. Huang, Dossa, Raffin, *et al.* 2022).

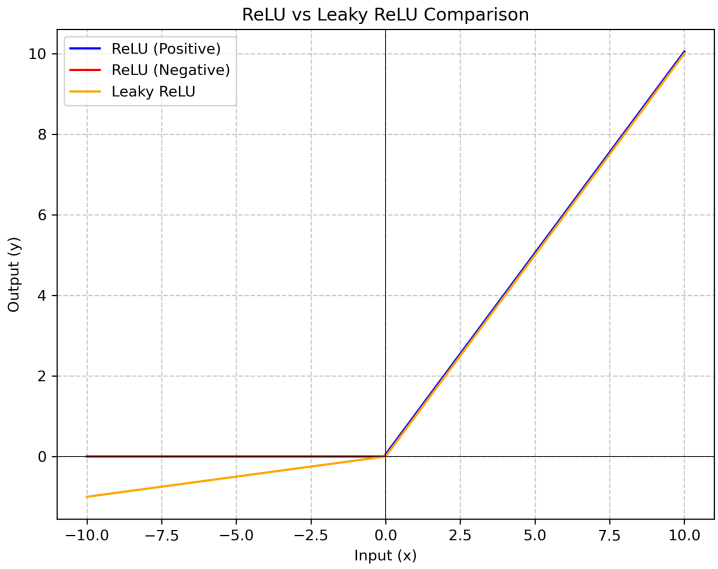


Figure 2.14: The figure illustrates the limitation of the popular ReLU activation function, which creates a "dead zone" where negative inputs are squashed to zero. In contrast, Leaky ReLU addresses this issue by allowing negative outputs as well, introducing a small, pre-defined slope for negative inputs. This slope, typically fine-tuned for specific tasks, prevents the dead zone and ensures gradients can flow even for negative inputs. We used the default value for this slope, which is set to 0.01 in the official PyTorch implementation (PyTorch Contributors 2024).

2.3.7 Squeeze-and-Excitation Block

The concept behind Squeeze and Excitation networks is straightforward: to enhance the performance of ResNets with minimal computational overhead, thus optimizing the performance tradeoff (J. Hu *et al.* 2017). The approach involves adding parameters to each channel of a convolutional block, allowing the network to adaptively adjust the weighting of each feature map. This enables the network to learn an ordering of **feature map importance**, effectively prioritizing certain features. This method is similar to feature selection techniques in machine learning (Yahya 2011), but with the key difference that it is learned dynamically by the network.

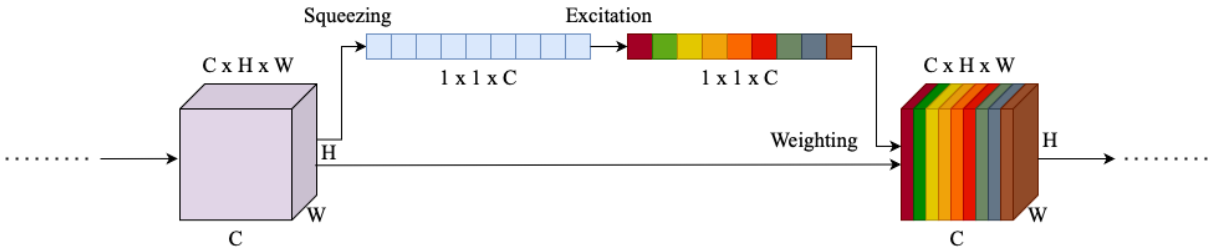


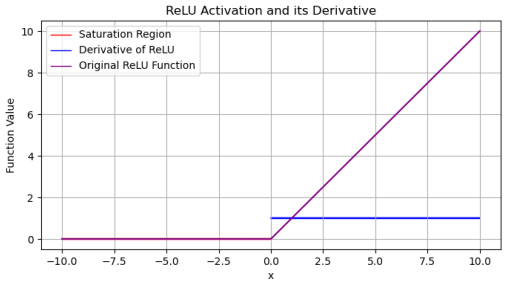
Figure 2.15: The figure illustrates how the channel-wise aggregation and self-gating mechanism generate per-feature channel weights for the original input. This process highlights the sequential transformation from spatial aggregation to the modulation of feature importance.

After the residual layers, the network branches into a **squeezing** path that compresses the input from $H \times W \times C$ to $1 \times 1 \times C$, effectively creating a channel descriptor that aggregates the feature maps across the spatial dimensions H and W (J. Hu *et al.* 2017). This aggregation is followed by an **excitation** operation, a self-gating mechanism that generates per-channel modulation weights from input embeddings. These weights are then applied to scale the channels of the original input $H \times W \times C$, which is then passed to subsequent layers for further processing.

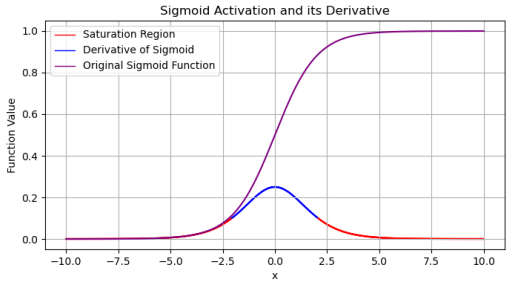
2.3.8 Batch Normalization

As networks grew deeper, not only was there a need for residual connections, but also for advanced regularization techniques as well (Tian & Zhang 2022). Batch normalization has come to light as a suitable solution in response to the challenges of deepening neural networks. Deep networks often employ **saturating nonlinearities** or activation functions that lead to a phenomenon known as internal **covariate shift** (Ioffe & Szegedy 2015). This shift refers to the change in the distribution of network activations due to the updating of network parameters during training. When inputs pass through an activation function, they may be pushed to the extreme values of the function's range, resulting in saturation. Saturation is problematic because it leads to derivatives near zero, especially noticeable in functions like the sigmoid, which squishes values between zero and one. Consequently, the derivatives at the extremes of the sigmoid function approach zero, causing very small updates during backpropagation (Rakitianskaia & Engelbrecht 2015). This issue becomes more pronounced as network depth increases, eventually reaching a point where the network cannot effectively update its weights based on the loss function.
Batch Normalization originates from the standard practice of normalizing input data to zero mean and unit variance, which optimizes gradient descent by ensuring updates are consistent across all feature dimensions. This uniformity in the loss landscape results in smoother updates. Batch normalization applies this principle to batches of data within the neural network, normalizing the activations from each layer to help the gradients converge more effectively (Ioffe & Szegedy 2015).

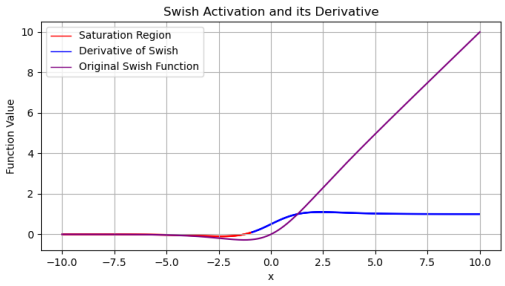
Batch normalization utilizes two learnable parameters, beta (β) and gamma (γ), and two non-learnable parameters, the moving averages of mean and variance. The process involves calculating the mean and standard deviation for the batch, normalizing inputs, and then using β and γ to scale and shift these values. These adjustments allow the activations to shift to means other than zero and variances other than unit variance. The moving averages of mean and variance are updated based on the current batch's calculations. This normalization is performed for each feature channel. By stabilizing the learning process, batch normalization reduces the need for



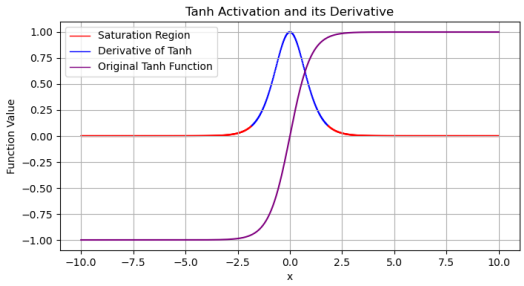
(a) ReLU nonlinearity.



(b) Sigmoid nonlinearity.



(c) Swish nonlinearity.



(d) Tanh nonlinearity.

Figure 2.16: These figures illustrate various popular activation functions alongside their derivative functions. We highlight regions at the extremes of each function where the derivatives are squished to very small values. In these regions, the network's updates during training are minimal.

dropout layers, enables **higher learning rates**, and decreases dependency on advanced initialization techniques (Ioffe & Szegedy 2015).

In our model, we use batch normalization at **each hidden layer**. We decided to apply the normalization **before the activation functions** since this is what the original paper's authors suggested. In the layers where batch normalization is enabled, biases are disabled to avoid unnecessary computations (Goodfellow *et al.* 2016).

2.3.9 Spectral Normalization

Spectral normalization, originally employed to stabilize the training of **discriminator networks** in Generative Adversarial Networks (GANs) (Miyato *et al.* 2018), has been sparsely adopted for use in actor-critic policy gradient methods in reinforcement learning. In GANs, spectral normalization addresses training instability and the issue of **mode collapse**, which occurs when the generated and real distributions become disjoint, leading to vanishing gradients (Yoshida & Miyato 2017). The structure of GANs, with distinct discriminator and critic networks, is similar to actor-critic methods in reinforcement learning, where actors and critics serve complementary roles. Additionally, Wasserstein GANs (WGANs) utilize the Wasserstein distance (Wikipedia 2024d) for their cost function, which provides smoother gradients. However, its effectiveness is based upon adhering to a **Lipschitz constraint** K. Lipschitz continuous functions restrict how quickly the function values can change, maintaining a slope between any two points that is less than or equal to a constant K, known as the Lipschitz constant (Wikipedia 2024e).

This same principle can be applied in actor-critic methods in reinforcement learning, where spectral normalization stabilizes the training of critics by rescaling the weight tensor using the spectral norm σ, calculated using power iteration methods (Ansel *et al.* 2024; Equation 2.20).

$$W_{\text{sn}} = \frac{W}{\sigma(W)}, \quad \sigma(W) = \max_{h:h\neq 0} \frac{\|Wh\|_2}{\|h\|_2} \tag{2.20}$$

When spectral normalization is applied in reinforcement learning, both the actor and critic are trained simultaneously and are interdependent. Applying spectral normalization to the critic helps reduce the risk of **policy collapse** (Dohare *et al.* 2023), where the actor might learn a suboptimal or **degenerate policy**. Furthermore, in policy gradient methods, if learning rates and initialization are not properly managed, one network may dominate the other, leading to an imbalance. Spectral normalization has been demonstrated to increase the generalization and stability of reinforcement learning algorithms, particularly in complex domains such as those involving Atari games, where it has been effectively integrated with Deep Q-Networks (Gogianu *et al.* 2021).

2.3.10 Orthogonal Weight Initialization

Weight initialization is essential in deep neural networks as it sets the **starting point** for training. Various methods have been employed for this purpose, with random initialization being one of the simplest. In random initialization, weights are assigned values from a normal distribution, which helps mitigate the vanishing gradient problem but can introduce significant variance within the network (W. Hu *et al.* 2020). Other popular methods include Xavier (Kumar 2017) and Kaiming initialization (K. He *et al.* 2015b), which are effective for shallow and moderately deep networks. For more complex architectures, especially in deeper or recurrent neural networks (RNNs), orthogonal initialization is often used. This technique involves initializing the weights with an **orthogonal matrix** (Wikipedia 2024f), a square matrix whose columns and rows are orthogonal unit vectors. This ensures that the weights are independent and maintain equal magnitude.

Orthogonal initialization offers several benefits, particularly for deep linear networks where it has been shown that the network width necessary for convergence to the global optimum does not depend on depth, unlike other initialization techniques that scale linearly with the number of layers (W. Hu *et al.* 2020). Additionally, the benefits of orthogonal initialization persist throughout training. This technique is also applied in practical scenarios, for instance, in the implementation of Proximal Policy Optimization (PPO), where it is used to initialize the hidden layers in Mujoco tasks, scaled by $\sqrt{2}$ with biases set to zero (S. Huang, Dossa, Raffin, *et al.* 2022).

The scaling factor of the initialization can have a substantial effect on the variance of both the activations and the gradients (Glorot & Bengio 2010). Proper weight initialization for the Rectified Linear Unit (ReLU) activation function with the scale of $\sqrt{2/n_t}$, where n_t is the number of input units to the layer, has been shown to ensure zero mean and unit variance of the outputs (K. He *et al.* 2015c), stabilizing the learning process. Omitting the n_t terms causes higher variance but has been widely used in practice, for which the original PPO implementation is a good example. Since we use Leaky ReLU activations, we used the slightly modified scaling of Equation 2.21, which was the scaling recommendation for that specific activation in Pytorch (PyTorch Contributors 2024).

$$\text{gain} = \sqrt{\frac{2}{1 + \text{negative\_slope}^2}} \tag{2.21}$$

Following the recommendations of S. Huang, Dossa, Raffin, *et al.* 2022, we used the activation function's recommended gain value as the scale for weight initialization in our hidden layers and a scale of 0.01 for the initialization of the actor heads. We initialized the critics with the same scale to get predictions close to zero, matching our scaled-down rewards (subsection 2.4.8). After learning how much downscaling the weights can help, in some benchmark tests boosting performance by 66% (Andrychowicz *et al.* 2020), we further scaled down the output layers of the network by a factor of 100 to make the action distribution more closely resemble a uniform distribution. We were still noticing fluctuations in performance based purely on the seed the networks were initialized with, so we performed the same scaling on the critic value output layers' weights and, eventually, the hidden layers.

2.3.11 Feature Extractor Model

We transitioned from a heavily shaped feature space to a **CNN-based feature extractor**, aiming to map the observations detailed in section (subsection 2.1.3) to an action tensor of size $[16, 48, 48]$. In this representation, 16 signifies the size of the discrete action space, while 48 denotes the map size. Given that both our observation tensor and action space possessed a channel size of 16, encompassing various map, unit, factory, and global features (Table 2.1), we required a mapping strategy to align the observation space with the action space structure (subsection 2.3.11). We implemented two distinct feature extractor architectures to achieve this goal.

The first network architecture adopted a U-Net style encoder (Ronneberger *et al.* 2015) with various variations of bottlenecks and a decoder with skip connections (D. Wu *et al.* 2020). U-Net architectures are well-known for their effectiveness in image segmentation tasks (Ehab & Y. Li 2023; Ronneberger *et al.* 2015), and we aimed to achieve a similar outcome but at the **entity level segmentation**. Our goal was for the network to recognize that entity-free spaces represented inactive segments of the input space, while units and factories remained segmented as distinct entities. Each entity's class corresponded to the action it performed in the environment. In simpler terms, we aimed to create a segmentation map that **established a class system** for each agent, categorizing them into different classes. For instance, if the network predicted a dig action for a unit at a given pixel, it meant that the network had segmented it into a **"digger"** class. Similarly, when the model predicted a recharge action, the corresponding agent was categorized into a **"recharger"** class within the grid environment.

The encoder of the network employed **Downsampling blocks**, transforming the input tensor from a shape of $[16, 48, 48]$ to $[256, 3, 3]$. Each Downsampling block comprised a residual block (subsection 2.3.5) and a downsampling convolutional layer with a kernel size of 3 and a stride of 2, effectively **halving** the height and width of its input. The residual blocks consisted of 2 convolutional layers, a squeeze-and-excitation layer (subsection 2.3.7), and a leaky ReLU (subsection 2.3.6) as the residual, which was concatenated to the input of the residual block. Spectral (subsection 2.3.9) and batch normalization (subsection 2.3.8) techniques were applied after each convolutional layer, followed by a leaky ReLU activation function.

Furthermore, each convolutional layer was initialized using orthogonal initialization techniques (subsection 2.3.10) based on weighting and gain, following recommendations from the literature (S. Huang, Dossa, Raffin, *et al.* 2022). To ensure reproducibility and **reduce variation in different trial runs**, each layer was hashed and seeded. The Squeeze-and-Excitation module employed a reduction factor of 16, facilitating the calculation of channel weighting and importance for every input channel of the observation feature.

We explored various bottleneck architectures, including a **simple convolutional bottleneck** comprising two residual blocks; a **dilated bottleneck** with two 3x3 convolutions with dilation rates of 2 and 4 respectively (Z. Li *et al.* 2018), followed by a residual block; and a **multi-scale bottleneck** incorporating 1x1, 3x3, 5x5, and 7x7 convolution operations to expand the receptive field of the model (Gao *et al.* 2021). The multi-scale U-net architecture, in particular, aims to mitigate degradation issues in target regions of segmentation masks by creating representations of the input on different receptive fields and concatenating them to form a comprehensive feature space representation. This approach has been demonstrated to be effective in maintaining the integrity of target regions in segmentation tasks (P. Li *et al.* 2023; W. Zhu 2024; Bhojanapalli *et al.* 2020).

In the decoder component of the network, we designed upsampling blocks that mirrored the structure of the downsampling blocks but in reverse. Each upsampling block begins with a **transpose convolution** (Shelhamer *et al.* 2016) to upsample the input feature map, followed by batch and spectral normalization and a leaky ReLU nonlinearity. The block concludes with a residual block, maintaining consistency with the previous structure. The inputs to the upsampling blocks consist of outputs from preceding layers and corresponding residuals from their paired downsampling blocks. We termed this feature extractor network the **BottleNet** (Figure 2.17).
For our second network architecture, we opted to maintain the input and output sizes **without downscaling**, thus eliminating bottleneck layers. With the goal of creating a one-to-one mapping, we designed a compact residual network that preserves the height and width of the feature maps throughout the network. This network comprises an input convolutional block with spectral normalization, batch normalization, and leaky ReLU nonlinearity. It is followed by four residual blocks, each incorporating SE layers for channel importance weighting, similar to the BottleNet architecture. The final block of this feature extractor network consists of another convolutional block with spectral normalization, batch normalization, and a leaky ReLU activation function. We named this feature extractor the **DashNet**.

2.3.12 Actor and Critic

To ensure compatibility with Stable Baselines, we introduced an additional layer of feature extractor, separated for both the actor and policy networks. To simplify and reduce computational complexity, we passed the output of our feature extractor network through an Identity layer for the actor. Then, for each pixel, its corresponding output value was sent through a linear projection layer, which mapped the extractor network outputs to action **loglogits**. For the value network, to fully represent the value of the entire state from the output tensor, we used a single convolutional block paired with layer normalization and leaky ReLU nonlinearity. To refine the representation of value estimates, this mapping compressed the values into a more suitable space, enabling the inclusion of small negative values for more accurate estimates. Next, we employed a **global average pooling method** Lin *et al.* 2014 to average the values of all pix-

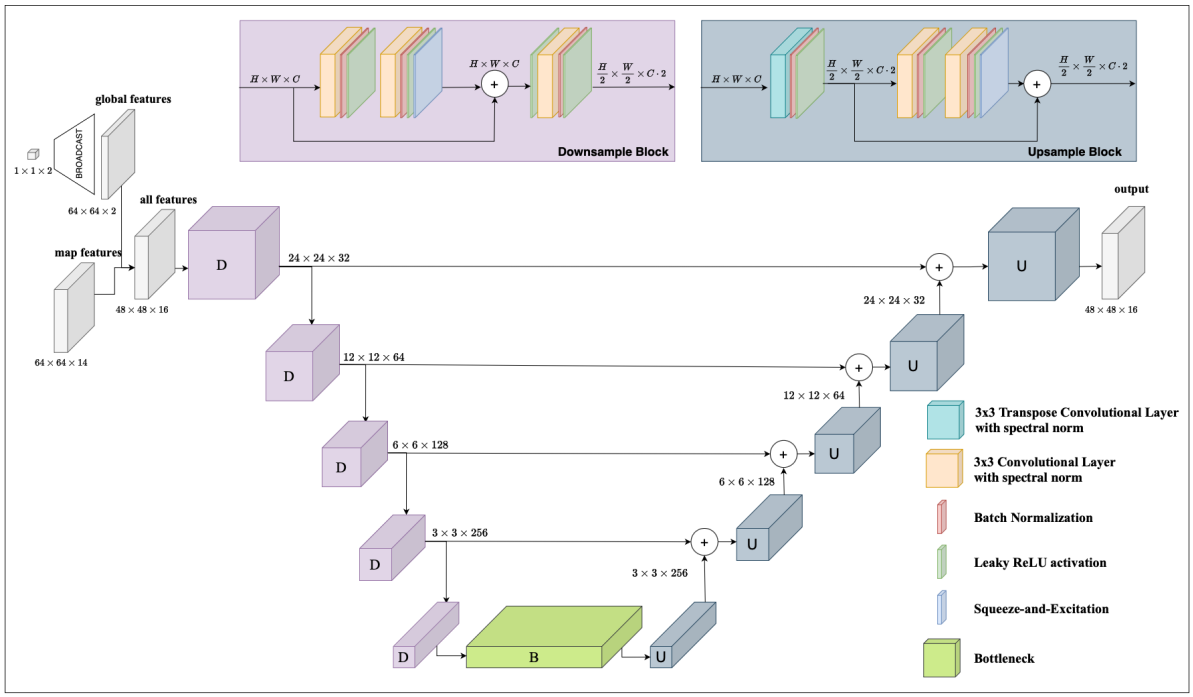


Figure 2.17: The figure depicts the entire feature extractor network, encompassing the process from input to output. Map features are combined with tiled-up global features, passing through a sequence of Downsample Blocks until reaching the bottleneck layer. The bottleneck layer's outputs are then upsampled via a series of Upsample Blocks, utilizing matching residuals from downsampling blocks to retain information. Finally, the output is directed towards both the critic and actor networks.

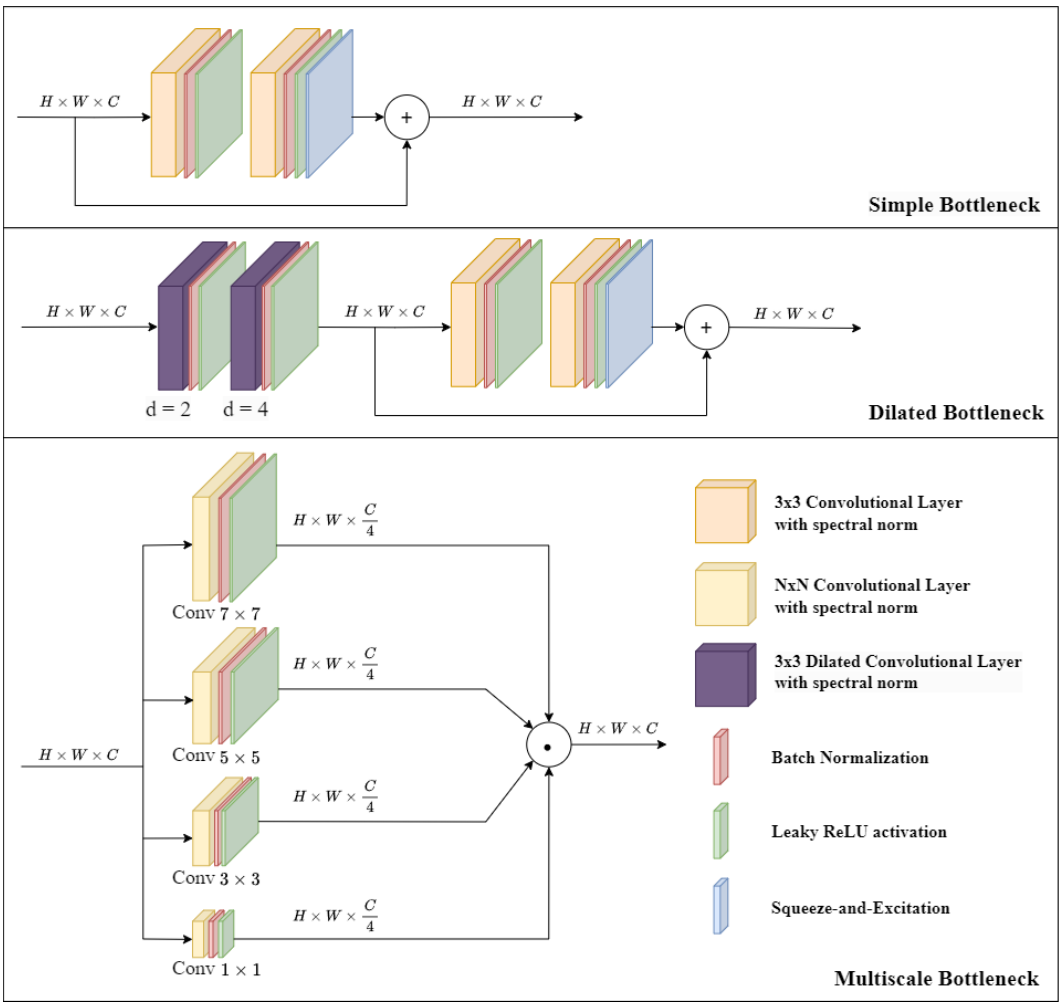


Figure 2.18: The three images illustrate various types of bottlenecks tested for our feature extractor. Starting from the top, the Simple Bottleneck consists of two residual blocks. The Dilated Bottleneck employs convolutional layers with dilation, where the filter expands by a certain factor, allowing it to capture a broader context. Lastly, the Multiscale Bottleneck features a four-branch network, emphasizing important features at different scales.

els channel-wise, resulting in a 48x48-sized tensor. This tensor was then flattened and passed through a linear projection layer to produce the final value estimate (Figure 2.19).

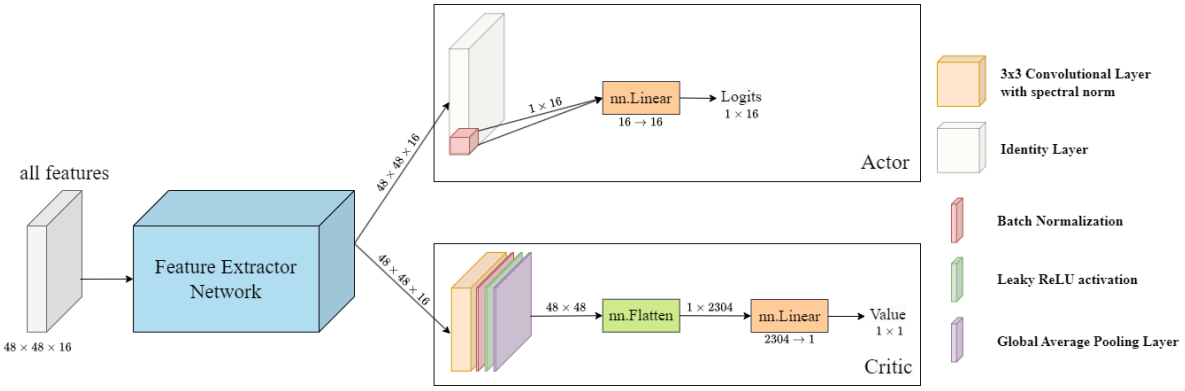


Figure 2.19: The diagram shows the actor and critic heads of the architecture. The actor head extracts action vectors for each pixel using an identity layer and a linear projection for logits. Meanwhile, the value head utilizes a final convolutional block with Global Average Pooling to calculate channel-wise averages, followed by a many-to-one linear projection.

2.3.13 Reward Function

We adjusted the rewards to incentivize global **ice collection, ice transfer, and water generation** in the environment. Ice collection was scaled down by a factor of four to normalize its value to that of water (as 4 ice can be turned into 1 water by the factory). Additionally, we further downscaled the ice collection by a tenth to incentivize not only the digging of ice but also its transfer to factories, which was rewarded 10 times more than simple collection. Water production was rewarded by a single unit, as it is the key metric for sustaining the factory and aligning our model with the task of maintaining factory viability. In order to incentivize units to fill the factories with ice as early as possible, the rewards are **multiplied by a factor**, made the following function:

$$f(\texttt{step}) = 1 + (1000 - \texttt{step})/1000 \times 0.1 \tag{2.22}$$

This factor **boosts the rewards** more the closer we are to the **start of the episode**. Our final reward function is the following:

$$r(\texttt{step}) = \frac{0.01}{f(\texttt{step})} \left[\frac{\Delta ice\_dug}{40} + \frac{\Delta ice\_transferred}{4} + \Delta water\_produced \right] \tag{2.23}$$

2.3.14 Training

We maintained the training environment and learning algorithm consistent with the single-agent testbench (section 2.1), employing Stable Baselines and its maskable PPO variant (subsection 2.1.10). While Stable Baselines is known for its compatibility with various single-agent environments (Raffin *et al.* 2021), it required significant adaptation for environments where a central decision maker governs actions for multiple entities. Consequently, we underwent a **complete rewrite of the PPO learning algorithm** from Stable Baselines to align with our requirements.

Our action space underwent a transformation from a single discrete 1x12 tensor to a **multidiscrete** 16x48x48 tensor. Within each HxW dimension, a 16-dimensional log logit tensor was generated. After exponentiation, an action mask was created for all logits in all HxW dimensions, effectively reducing the corresponding masked values to extremely small numbers, resulting in near-zero probabilities during normalization. Consequently, we had to permute and correctly reshape the action mask and policy head values to tensors of shape 48x48x16. Subsequently, for the calculation of log probabilities, advantages, entropy, and actions, a 48x48 tensor had to be supplied. While further refinements and adjustments were made, detailed discussions on these intricate aspects are omitted here, as they are available in our publicly accessible repository (`https://github.com/MagmaMultiAgent`).

It's crucial to note that implementing self-play in a multi-entity environment using stable baselines is not straightforward (Gleave *et al.* 2019). As a workaround, we employed a passive enemy agent that doesn't interact with the environment. Consequently, our M-PPO algorithm could only learn from the experiences collected by one player, effectively halving the available training data. To compensate for this reduction, we **doubled the size of the rollouts** from 4096 to 8192. Additionally, to maintain the desired number of updates during batched gradient descent at 8, we set the minibatch size to 1024. For a comprehensive overview of the hyperparameters and environment settings, please refer to section A.2.

We originally chose to train the agent over 100k steps, corresponding to 25 evaluation phases, with each phase covering rollouts of size 4096 steps. However, since self-play was not implemented in Stable Baselines (Gleave *et al.* 2019), we extended the training to 200k steps (8096×25) to compensate. This ensured that we **maintained the same number of model updates** (25), approximating a self-play environment as closely as possible. Subsequently, we applied the same heuristics and training approaches in our hybrid method as well.

2.3.15 Evaluation

We standardized our evaluation phases by conducting **evaluations after every training phase**, totaling 8192 steps, using 12 environments with an extended episode length of 1000, the upper limit in the Lux AI Competition (section 1.4). We employed different seeds for the evaluation environments to thoroughly assess the model's generalization performance [10]. Logging global metrics, including overall ice dug, transferred, and water generated in each evaluation environment, we conducted three trials as before for the single-unit test bench. For the final result, we averaged the global metrics collected for every parallel environment per trial (3×12 environments) and calculated their standard deviation. It's worth noting that not all environments ran for the same number of steps; in such cases, we computed averages and deviations based on the longest step environment and **filled any gaps with zeros** where the **environment ended sooner**. As for convergence, we measured the agent's ability to keep at least one factory alive until the end of the standard episode length of 1000 steps in the Lux AI Competition.

2.4 Hybrid Approach

In order to overcome the difficulties of both the monolithic architecture (section 2.3) and fully multi-agent methods, we implement a **pixel-to-pixel** architecture (Chen *et al.* 2023) for our

[10] The trials were trained and evaluated using seeds $42-0$, $43-1$ and $44-2$ respectively.

hybrid **centralized control** model. Just like in the monolithic architecture, the global observation is fed into a convolutional neural network, which outputs an **action for every pixel of the board**. The actions of factories and units are obtained by selecting the resulting action at their corresponding position. The main difference comes in the shape of the network. The pixel-to-pixel architecture preserves the board's shape at all steps, allowing the model to learn simple mappings instead of relying on a bottleneck. These mappings can be further simplified by the use of residual connections, explained more in depth in subsection 2.3.5. By keeping the shape of the layer outputs consistent, we can allow each action output to be based on features in the corresponding entity's neighborhood. Global information can also be accessed by using repeated convolution operations, as shown in Figure 2.20. Generating all actions for each pixel from a single global observation is very similar to processing batched local observations for multiple entities separately. An example can be seen on Figure 2.21. The pixel-to-pixel approach is **computationally faster** and requires less data to be stored in memory. Moreover, by using a shared convolutional network, the entities can access the internal representations of each other, establishing a possible channel for communication (Han *et al.* 2019).

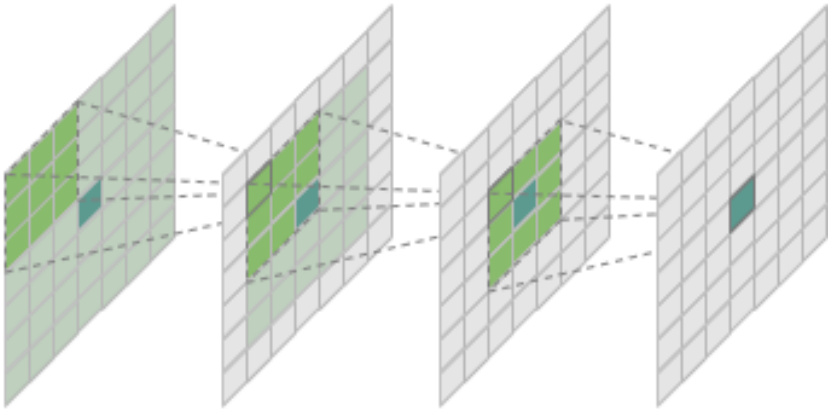


Figure 2.20: Figure demonstrating how the use of repeated convolution operations can process global information. The center of the 7x7 grid is marked with ocean color, and it represents an entity in the Lux environment. The increasing field of view is indicated by the shaded green area. The image shows how both local and global information can be utilized in the generation of the entity's action.

Because a central decision maker is used, but in a way that allows each agent to make their decisions based on their **localized observations**, we call this approach **hybrid**. Later, we augment this method with the ability to **assign rewards to individual** factories and units instead of purely global rewards. This change makes our approach more similar to a truly multi-agent control system while maintaining the benefits of operating with a **single global observation** of the game board. We will compare this modified reward-assignment method to the original global reward system. For a more detailed comparison of the mentioned methods, please refer to Table 2.2.

The training goal remained consistent for this phase as well: **learn how to keep factories alive** until the end of the episode. We will now outline the specifications of the training algorithm and model used in the test.

Property	Monolithic	Distributed with Local Observations	Distributed with Global Observations	Hybrid with Global Trajectories	Hybrid with Separate Trajectories
Passes through Model	once	per agent	per agent	once	once
Observations Stored	once	per agent	per agent	once	once
Observation Size	global	local	global	global	global
Field of View	global	local	global and local	global and local	global and local
Distributed Rewards	no	yes	yes	no	yes
Handles Changing Agent Numbers	yes	no	no	yes	yes
Changing Batch Size	no	no	no	no	yes

Table 2.2: Table showing the differences between agent control architectures. While storing global observations for every agent with a distributed approach is similar to the hybrid architecture, it requires much more memory and cannot handle changing agent numbers. A hybrid architecture with separating trajectories (subsection 2.4.5) has every benefit of distributed control.

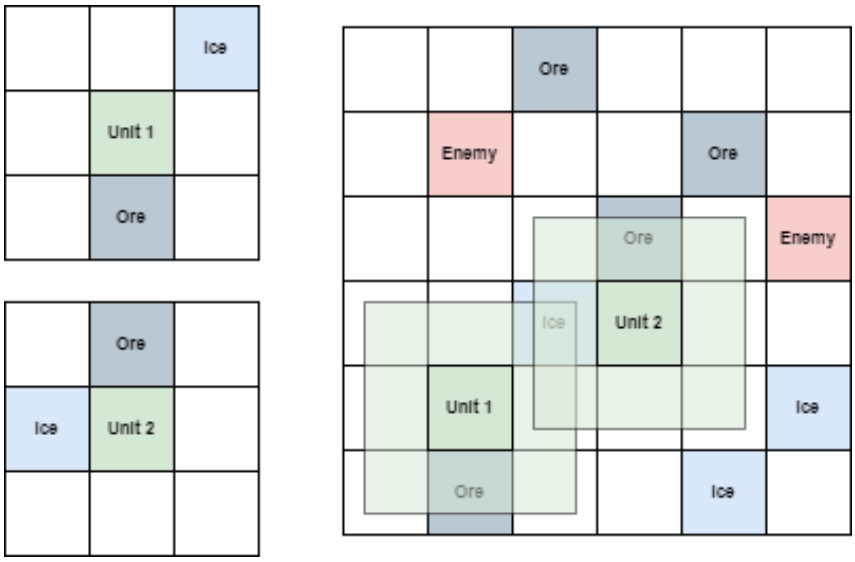


Figure 2.21: Image showing how performing shape-preserving convolution operations on the whole observed map (right) is similar to having localized observations for every unit (left). The range of a 3x3 convolution around the units can be seen on the right, marked by an opaque green filter.

2.4.1 Environment

The environment largely aligns with the description in subsection 2.3.1, with notable changes made to the action space. It now encompasses **all actions available** in the Lux AI environment (subsection 1.4.4). Additionally, a **stricter action masking** mechanism has been applied to limit unnecessary movements and collisions among units.

2.4.2 Heuristic Bidding and Factory Placement

We retained our initial Gaussian factory placement heuristic outlined in subsection 2.3.2, making minor adjustments to the weighting calculation and scale of the Gaussian filters to ensure factories are positioned **as close as possible to ice**.

2.4.3 Actions

Since our goal is to keep the factories alive until the end of the game, we decided to **get rid of the lichen mechanic** completely via action masking. This meant that the factories had three possible actions left: build heavy, build light, and do nothing. We forbade the self-destruct action for units since hostility was not needed to achieve the simplified objective.

Unit actions are comprised of an action type and action settings, depending on the type. Most action settings were implemented heuristically, such as the transfer amount and the repeat flag; however, the most significant parameters were left for the model to decide. Parameters include the move direction, pickup amount, transfer direction, and transfer resource.

We **forbade illegal actions**, such as factory actions that require more resources than the factory's current cargo, moving out of the map, and choosing unit actions that need more power than the agent's current battery. Actions that would slow down exploration were also disabled to **speed up the training**. Such actions are doing nothing when the factory could be producing

heavy units with its resources, units recharging, or picking up power from factories while at full power. Agent collisions between the team members were also disabled, with keeping track of each unit's movement possibilities and allowing only one at a time to step on a board tile. To stop factories from spawning new units on top of existing ones, agent creation is masked out if a unit is standing on the middle tile of a factory. Agents cannot enter the middle of factories once they have stepped away from them. Actions we considered too complex and unnecessary for the game were also removed with masking, such as picking up resources from factories and transferring resources or power to other agents. The complete list of possible actions and their requirements can be found in Table 2.3.

Entity	Action	Parameters	Requirements
Factory	Build Heavy unit	None	- 100 metal in factory - 500 power in factory - middle of factory is empty
Factory	Build Light unit	None	- 10 metal in factory - 50 power in factory - middle of factory is empty
Factory	Do nothing	None	- no resources to build heavy unit
Unit	Do nothing	None	- no power to update action queue
Unit	Recharge	Amount [0, 10]	- battery not full
Unit	Pickup	Amount [0, 10]	- battery not full - standing on factory
Unit	Move	Direction [0, 4]	- no friendly unit at target - enough power - no enemy factory at target
Unit	Transfer	Direction [0, 5] Resource [0, 1]	- has cargo - target is factory
Unit	Dig	None	- standing on ice, ore or rubble - enough power

Table 2.3: Table containing the list of possible actions, along with their requirements. If the requirements are not fulfilled, the action is masked out.

2.4.4 Observation

We maintained the observation space described in subsection 2.3.4, as it already covered the necessary information for the feature extractor model to align agents for ice resource collection and survivability. Although additional **location information** was provided to the network, it was only utilized for **auxiliary purposes** and filtering, not incorporated into the forward pass of the feature extractor.

2.4.5 Trajectory Separation

In a **multi-agent environment**, where the actions of units and factories are deeply intertwined, the question of **credit assignment** naturally arises. Suppose we use rewards corresponding to a global objective or to objectives that require the cooperation of multiple units or factories. In

such a scenario, tracking whom to reward and what amount is challenging since multiple agents work towards a common goal. A **centralized control** approach offers a potential solution, as all units and factories will belong under a single agent, which receives a **global reward** after performing the actions with all entities. Such reward assignment is still possible when using a fully multi-agent approach by giving every agent the sum or mean reward of the team. However, as we will later see, relying purely on global rewards makes it hard for the model to learn which entity's action was beneficial from the many numbers of entities. The more entities we have, the less of a chance there is that most of them behave optimally. A highly beneficial or determinal action can skew the total reward, which can lead to the **reinforcement of bad behavior** or the **punishment of good behavior** for the entities whose reward was less dominant in the final sum.

To tackle this complex problem, we use rewards that can be **immediately credited** to the performing or benefiting entity. The ice-dug reward is credited to the performing unit, while the ice-transfer reward is split between the performing unit and the receiving factory. Instead of summing up these rewards to a single global value, we **store them separately** to track which entities' actions resulted in how much reward. Since we are using an actor-critic method for training, the critic values should also be distributed. We utilize separate critic value predictions for each entity (subsection 2.4.7). These separate predictions are generated using information about the whole game state, similar to the method of Lowe *et al.* 2020.

In addition to **global** and **per-entity** reward assignments, we also experimented with putting entities into **groups** according to certain grouping rules, where a group's reward is the sum of its members' reward. An example of the reward separations between different groups can be seen in Figure 2.23.

Similar to the rewards and critic values, we also store the action log probabilities and entropies separately. This separation essentially splits every environment step into **multiple training examples**, creating multiple **distinct trajectories** from the same number of environment steps. The loss calculation does not change from the original PPO algorithm, as the modification simply translates into an additional tensor dimension, and all of the loss samples will get averaged into a single loss value as shown on Figure 2.22. After the modification, during backpropagation, the weight can be updated more accurately to reinforce good and diminish bad behavior since the fewer number of units and factories that belong to one group, the less of a chance a single entity has of skewing their group's reward, resulting in an inaccurate advantage calculation and a faulty weight update. We will refer to this method as **trajectory separation**.

PPO uses a **termination flag** stored for every step when calculating future rewards during advantage estimation. This flag signals the start of a new episode and ensures we do not include this episode's rewards in the cumulative return calculation of the previous steps. We need to modify this by adding an **extra dimension** to the stored termination flag tensor in order to keep track of the lifecycle of every group since we do not want to calculate steps into the cumulative return where the group was inactive. An example can be seen on Figure 2.24. A group is marked done when it contains no active units or factories. In the simplest case, when the groups consist of single units and factories, the flag is set to True if the entity: 1. has not been created yet, 2. is destroyed at that step, or 3. remains destroyed. A group consisting of multiple entities can be calculated by performing the AND operation on each of its member's termination flags. At the first step of an episode, every group is marked done.
If we use a grouping rule that allows groups to be occasionally inactive, for example, by assigning every unit or factory to its own group, the number of training examples belonging to one step

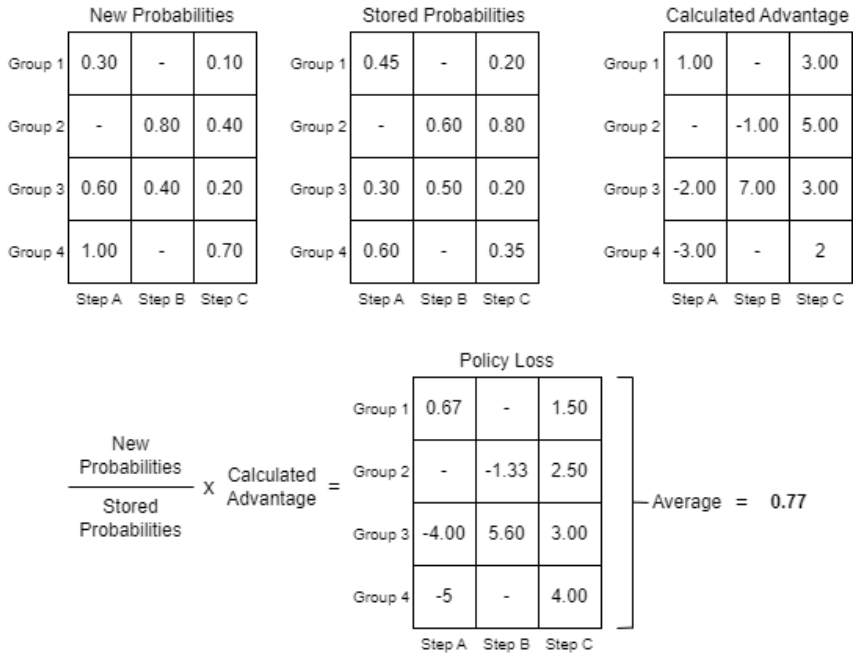


Figure 2.22: Demonstrating policy loss calculation with the extended dimension. The probabilities and advantages are calculated on a group level. The matrices on the image represent a mini-batch consisting of 3 environment steps. Inactive groups at all steps are marked with a dash, as these are excluded from the training data. The loss gets computed for all active groups, creating more training examples. In the above image, three environment steps resulted in 9 train steps. These loss values are then averaged to get a final scalar loss value. For simplicity, the clipping operation and advantage normalization are left out of this image.

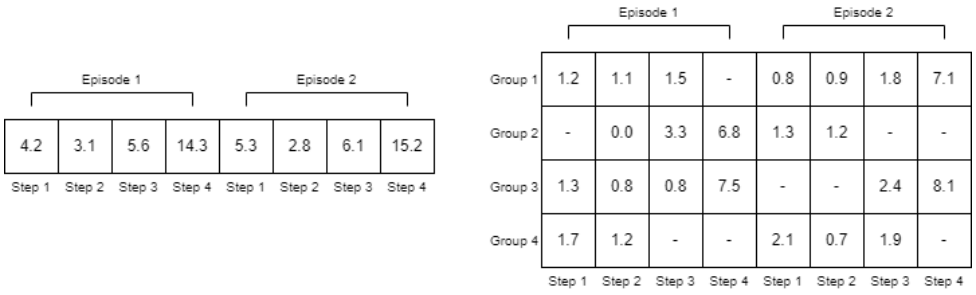


Figure 2.23: Image showing the difference between global rewards (left) and rewards distributed into groups (right). The examples are 4-length episodes with random data.

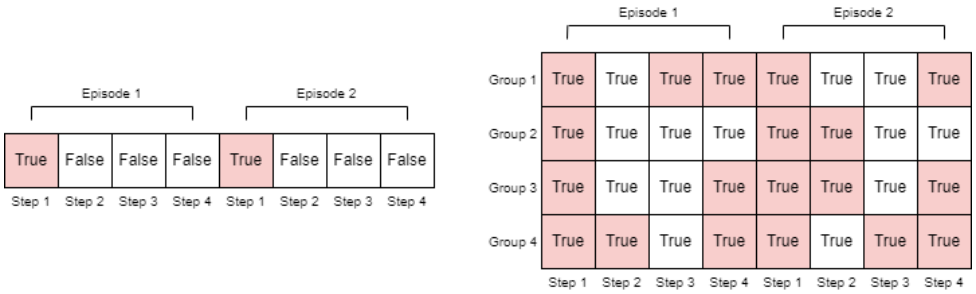


Figure 2.24: Image showing termination flags for a standard environment (left) and termination flags stored for every group (right). In addition to flagging the end of an episode, the death of a group is also indicated. The examples are 4-length episodes with random data.

will **not be constant**. Since we divide the **mini-batches** by environment step, this could mean that not all mini-batches consist of the same number of training examples. Because the loss values are averaged over a mini-batch to get the final loss, this could mean that training examples that belong to steps with fewer active groups have a **more dominant** effect during gradient updates since, on average, they create mini-batches with fewer training examples. Following the same logic, entities at more populated steps of the environment will have **less impact** on the final loss. We test using grouping rules that **keep group numbers consistent** to overcome this effect. We also experimented with keeping all entity trajectories separate but kept **only a set number of training examples** from each step to reduce the training example variance.

2.4.6 Feature Extractor Model

We use a **convolutional neural network** to map the observations into actions for each unit and factory on the board and to output critic values and log probabilities used by the training algorithm. Both the **Actor** and **Critic** networks have the same **Embedding Network** but do not share the weights to avoid the competing objectives of the policy and value functions (S. Huang, Dossa, Raffin, *et al.* 2022). The structure of the embedding can be seen on Figure 2.25. The main building blocks of the network are convolutional layers with **3x3 kernels** and padding to **keep the board's shape**. The layers alternate between a basic convolutional layer and a **residual connection** (subsection 2.3.5), where two convolutional layers are followed by a **Squeeze-and-Excitation** block (subsection 2.3.7). Each convolutional layer's output goes into a two-dimensional **batch normalization** block (subsection 2.3.8), then to a **Leaky ReLU** activation function. The weights of the convolutional layers are scaled by their **spectral norm** (subsection 2.3.9).

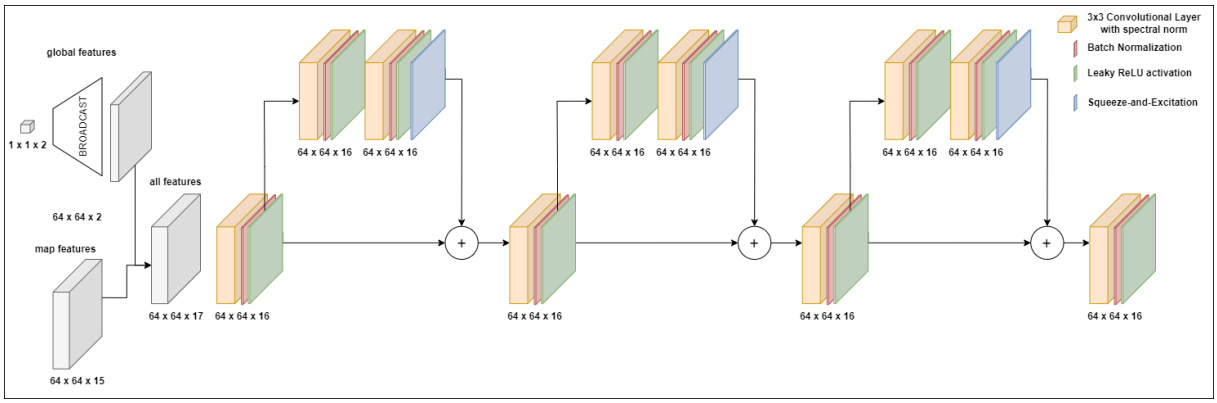


Figure 2.25: The structure of the Embedding network. The network uses convolutional layers with 3x3 kernels. Each convolutional block is followed by 2D batch normalization and a Leaky ReLU activation function (subsection 2.3.6). In order to regularize the network output and allow more layers, we use residual connections with a Squeeze-and-Excitation layer output. This embedding is the backbone of the critic and actor network.

2.4.7 Actor and Critic

After the embedding is created from observations, the critic values and the actions are constructed by their respective heads. Since the Lux environment is highly complex, and the entities need to solve multiple tasks in order to reach their goal, multiple reward sources (subsection 2.4.8) are needed to ensure the rewards are not too sparse for optimal training. In addition

to that, since the entities get rewarded based on outcomes, which in reality do not entirely depend on them, it is hard to predict an accurate critic value. Since multiple critic heads already proved to be successful in multi-task environments (Mysore *et al.* 2022), we decided to provide a separate critic value prediction for each entity and, in addition to that, provide two additional critic heads for both units and factories, respectively, that use global information for their predictions. The **Critic Network** (Figure 2.26) is made up of 4 components: a **unit** and a **factory** value network that outputs a value at every board position, and a unit and factory **global value** network that, in addition to outputting a value for every position, averages them into a single scalar value. The global unit and factory value scalars are then added to every position of the unit and factory value net output, respectively. This structure ensures that each unit and factory's value calculation is based on **both local and global** information. The final outputted values are created by selecting the predicted value at each unit's and factory's position on the board and then adding the value to the entity's group (subsection 2.4.5).

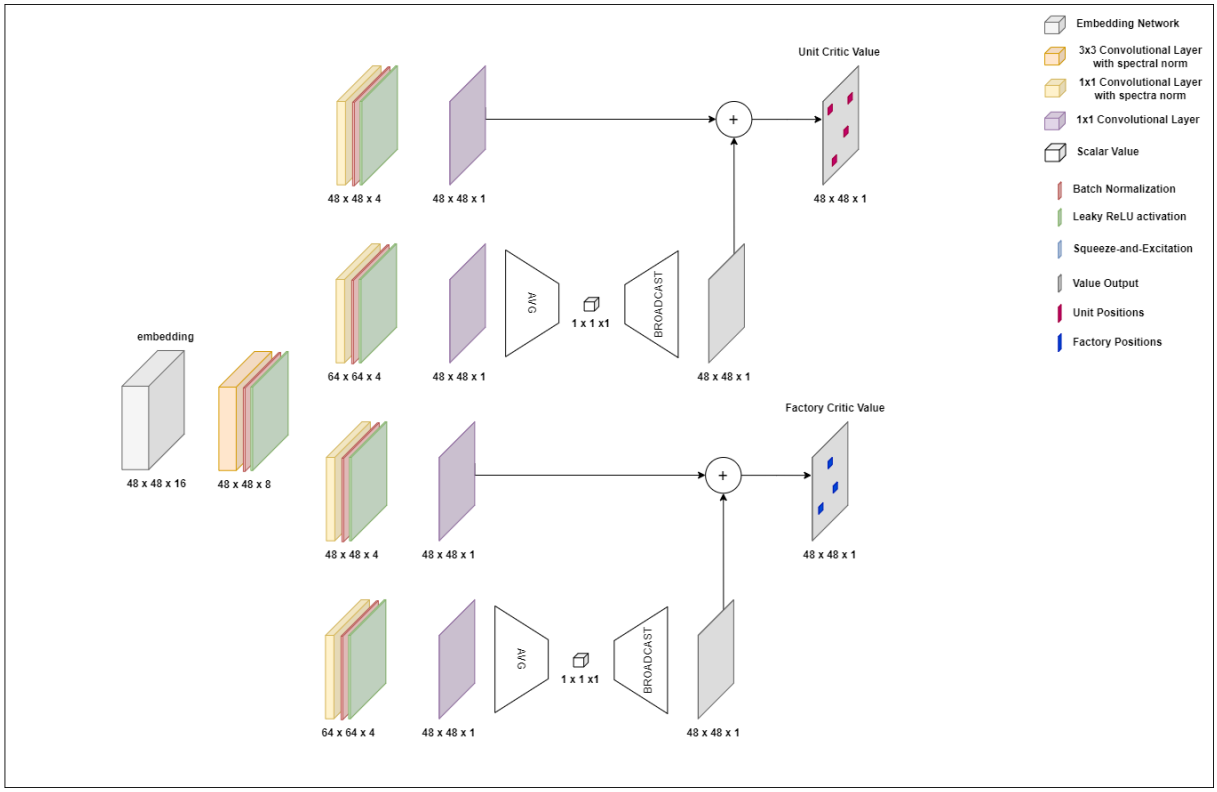


Figure 2.26: The Critic network takes the output of the Embedding and outputs critic values for each entity on the board. Each value is comprised of a local value and a global value, which is averaged over the whole board and then added to each position.

Actions are created by the **Actor Network** (Figure 2.27). Factories and units have different action spaces, so different actor heads output their actions. The action space of the factories is a simple discrete action space, while the units have a **multi-dimensional action space**, where each action has a different set of parameters. Due to some actions having a **heuristic parameterization** (subsection 2.4.3), only four parameter action heads are used in addition to the action type head. The heads output log probabilities, which are then used to create **categorical distributions** (Wikipedia 2023), discrete probability distributions over the possible actions. Actions are sampled from these distributions after setting the probability of masked-out actions to zero.

The sampled actions are outputted in the same board shape at the positions of units and factories. For each entity, the composite action's log probability and the distributions' entropies are summed together at an entity level. These values are added to the total log probability and entropy of the given entity's group (subsection 2.4.5).

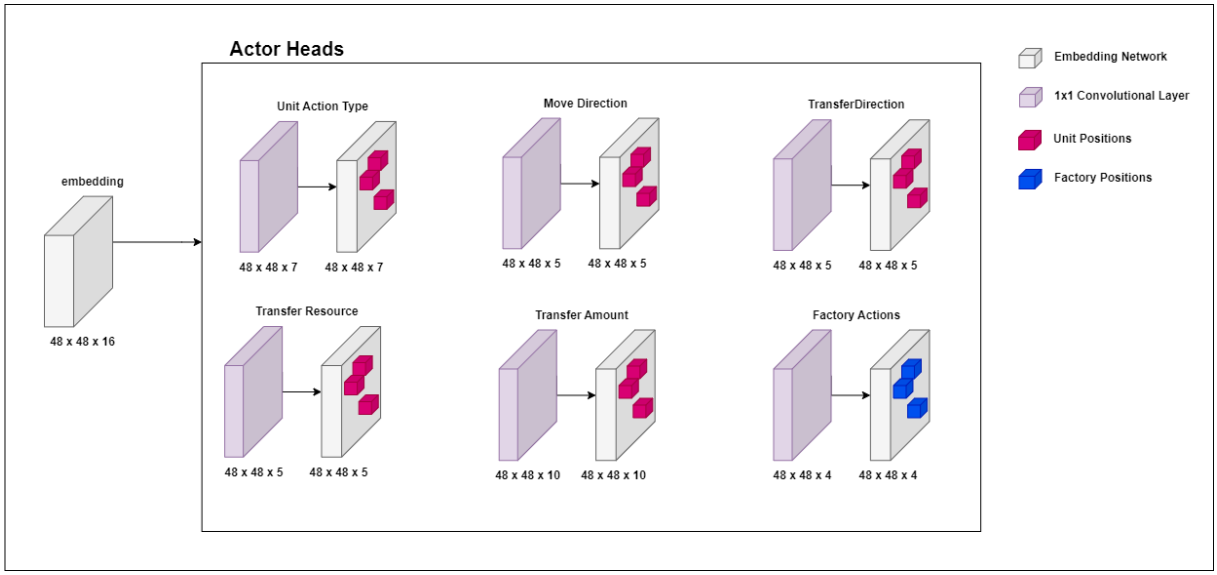


Figure 2.27: The Actor network is made up of separate actor heads, which share the same Embedding network. The output of these heads is the log probabilities of the action distributions. Factory actions have only one output, while unit actions require multiple outputs to parameterize them.

2.4.8 Reward Function

The rewards are shaped to **encourage the units to bring ice to factories**. In order to achieve this, 1 unit of reward is given after every one water worth of **ice is transferred to a factory**. After **mining** one water worth of **ice**, a 1/10 unit of reward is given to ensure that units find desirable behavior patterns as early as possible. All of these rewards are then divided by to **scale them closer to zero**. In order to incentivize units to fill the factories with ice as early as possible, we utilized the same formulation for the multiplication factor in Equation 2.22.

2.4.9 Training

To keep experimenting and minimizing training duration, we continued utilizing the Masked PPO algorithm (subsection 2.1.10). Our implementation, derived from Kai Yang 2023, provided by the contest organizers as a baseline, **differed from conventional approaches**. We opted to train the model twice after each rollout step, using data collected from each player separately. This approach enabled us to gather experience from both players without mixing their training examples during mini-batch creation. Leveraging **self-play** and access to both enemy and own player data, we halved our rollout size, which had been doubled in the monolithic approach (subsection 2.3.14), to 4096. Additionally, we reduced the batch size by half, from 1024 to 512. Further details on hyperparameters and subsequent modifications can be found in section A.3.

2.4.10 Evaluation

The evaluation settings remained consistent with those outlined in the monolithic approach (subsection 2.3.15), employing 12 evaluation environments with episode lengths set to the maximum value of 1024 steps. Transitioning to the hybrid approach allowed us to implement a more **comprehensive logging system**, enabling us to devise and calculate more specific training metrics. While episode length serves as a primary metric reflecting our training goal, measuring the **amount of ice transferred by units to factories** provides a more nuanced comparison, as it is practically unbounded, unlike episode length, which can be quickly reached in multiple trials. Additionally, we recorded other metrics, such as the **average factory count per step**, to assess the ability of entities to maintain multiple functioning factories instead of just one. Most other metrics were retained as baselines for comparison.

Chapter 3

Results

This section presents the quantitative results of our experiments with the methods introduced in section 2.1, section 2.3, and section 2.4. Here's an outline of our result presentation:

- In section 3.1 we present results from the single-agent testbench, utilizing metrics such as **episode length**, representing the survival duration of at least one factory of the learning agent, **ice dug**, indicating the amount of ice collected by units, and **training speeds** measured in **steps per second**. We also assess convergence, indicating the agent's ability to keep the factory alive for the Lux environment's specified episode length.
- In section 3.2 we present results from the monolithic network outlined earlier. We use similar metrics as in the single-unit testbench, but introduce an additional metric: **ice transferred**. This metric evaluates whether agents are transferring resources between each other, which was not relevant in the single-unit case.
- In section 3.3 we showcase the novelty of our research: the results on the hybrid agent control. This approach incorporates trajectory separation, distributed credit assignment, multiple actor and value heads, and a highly scalable feature extractor. We introduce a new metric called **average factory alive**, indicating not only the agent's capability to keep one factory alive but multiple factories. We conduct experiments with **different trajectory numbers** (subsubsection 3.3.1.2), including **top N unit trajectories**, **trajectory groupings** (subsection 3.3.2), and **interchanging separate trajectories with global rewards and advantages** (subsubsection 3.3.3.2). Additionally, we **compare our methods with other works** (subsection 3.3.4) and present a comprehensive **ablation study** (subsection 3.3.5).

3.1 Single Unit Testbench

For our initial comparison, we focused on environment-related metrics such as episode length, ice dug, and water produced to determine which algorithm performed best in exploration and resource collection tasks. It's important to note that the results presented in Table 3.1 are averages from 125 evaluation episodes during the best-performing evaluation phase. We chose not to include averages throughout the training process as we consider this information irrelevant for selecting the best algorithms among those tested. A high average could indicate that an algorithm is stuck in a potentially good local optimum rather than the global optimum. By presenting only the best evaluation phase, we aim to showcase the true capabilities of the algorithms.

Algorithm Type	Algorithm	Episode Length	Ice Dug	Water Produced
Policy Gradient Methods	TRPO	613	3200	744
	A2C	509	1201	249
	PPO	**1024**	14460	3610
	R-PPO	301	410	1
	M-PPO	**1024**	**14780**	**3667**
Value-Based Methods	DQN	560	2350	536
	QR-DQN	**1024**	13355	3108
Evolutionary Method	ARS	301	265	1

Table 3.1: The table shows environment-related results from the single-unit testbench, focusing on the top three performers for this task. We emphasize the crucial aspect of convergence in the table, where the average episode length during evaluation must be 1000 [1]. Three algorithms reached this threshold; however, M-PPO achieved the highest amount of ice dug and water produced during its best evaluation phase.

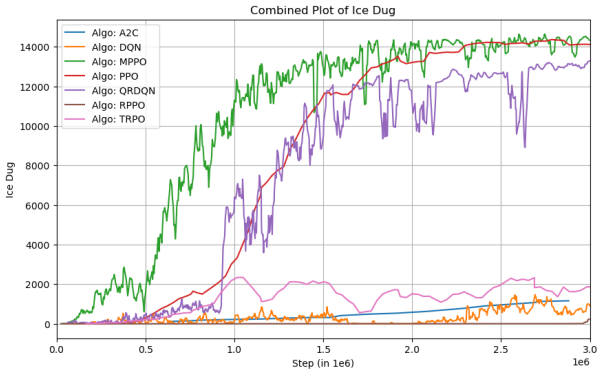


Figure 3.1: Average ice dug during evaluation phases throughout training.

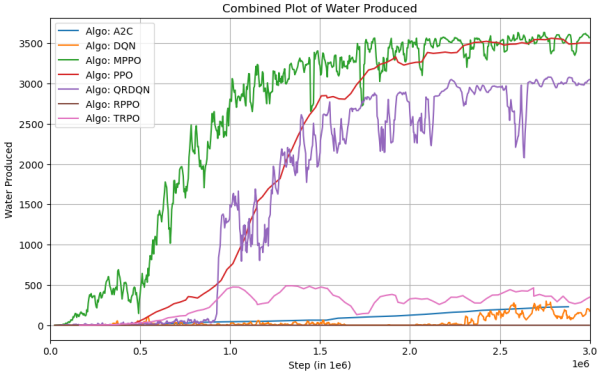


Figure 3.2: Average water collected during evaluation phases throughout training.

Figure 3.1 and Figure 3.2 demonstrate that PPO, M-PPO, and QR-DQN significantly outperformed all other algorithms. These plots also reveal that the performance of certain algorithms, such as A2C and DQN, has been improving, while TRPO remained stagnant in a local optimum throughout the evaluations. In stark contrast, ARS and R-PPO underperformed significantly. To further assess the models, we also considered the speed and convergence time of the algorithms to identify the most suitable one for our specific task.

[1] Note that in the table, the episode length is presented as 1024. This is because the evaluation length was set to a power of 2 for easier calculations.

Algorithm	SPS	Convergence
TRPO	1897	-
A2C	3032	-
PPO	**2742**	**1.6M steps**
R-PPO	656	-
M-PPO	**1258**	**1.2M steps**
DQN	2464	-
QR-DQN	**170**	**1.7M steps**
ARS	3246	-

Table 3.2: Steps per second were recorded throughout training, indicating how many environmental steps in the rollout buffer could the algorithms process per second in the evaluation phases. Convergence data is missing for five of the algorithms, as they did not achieve an episode length of 1000 in any evaluation phase. In the table, we calculated steps in millions, hence the notation of **M**.

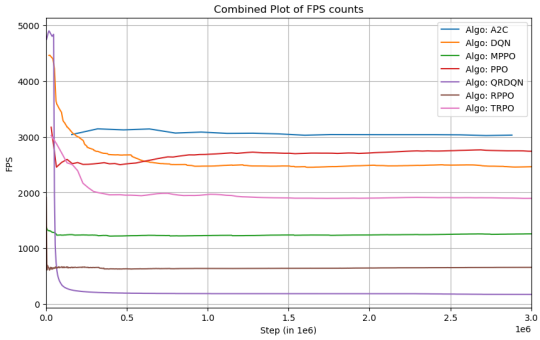


Figure 3.3: The plot represents a running average of steps per second during evaluation phases throughout training. It is evident from the plot that the numbers eventually stabilize around a specific value, which accurately reflects the true speed of the algorithm.

The data in Table 3.2 validates a key assumption: masking out invalid actions in the environment, thereby limiting the exploration space, proves advantageous in early training. This approach accelerates learning by preventing illegal moves. The ongoing cost of recalculating masks in later training stages yields diminishing returns, as the model naturally learns to avoid meaningless actions over time. Nonetheless, this trade-off is highly favorable, as the early acceleration is essential to prevent the agent from getting stuck in local optima, and the minor decrease in training speed during later phases is acceptable. It's worth mentioning that QR-DQN (subsection 2.1.12) exhibits significantly slower performance, primarily due to the utilization of 200 quantiles, which multiplies the action space by 200-fold. Consequently, the action space expands to 11 actions, resulting in an output shape for the heads of $11 \times 200 = 2200$, compared to just 11.

3.2 Monolithic Approach

3.3 Hybrid Approach

In this section, our primary focus will be on showcasing the effectiveness of our technique called **trajectory separation** (subsection 2.4.5) to demonstrate its potential for improving the hybrid architecture (section 2.4). In addition, we will explore potential enhancements and limitations of the method, including examining the components of trajectory separation to determine the specific factors that contribute to the improvement of performance. We will also conduct a comparative analysis between the trajectory-separated hybrid, **pixel-to-pixel** architecture (section 2.4) and existing solutions for the Lux AI competition. At the end of the section, an ablation study will be carried out, analyzing the methods in addition to trajectory separation.

In the subsequent configurations, factories, and units will be organized into distinct groups, each with aggregated outputs and trajectories. Groups can exhibit various characteristics, from

global groups encompassing all entities of a specific type to individual groups in which each entity functions as its own separate group. In this study, we illustrate the significance of distinct trajectories within a multi-agent setting and investigate various approaches for distributing rewards among the agents. All of the experiments shared a common training objective, which was to train the units to keep the factories alive until the end of the game, spanning 1,000 steps. In the following experiments, we will employ the metrics specified in subsection 2.4.10. These metrics include the length of an episode, the aggregate amount of ice transferred to factories within a given episode, and the count of operational factories at each environment step, normalized by the maximum episode duration.

The experiments consist of three separate runs, each utilizing different seeds and running for a total of 102,400 steps (25 training cycles with a 4,096-step batch size). We felt the difference between variants could be sufficiently highlighted in this step range. Contrary to the single-unit test bench (section 3.1) and the monolithic approach test (section 3.2), in this scenario, both players are active and performing actions simultaneously. The same model, as detailed in subsection 2.4.6, was used for both players and was trained on their collective trajectories. Following each training cycle, we ran an evaluation of the model by executing 12 distinct environments until the conclusion of the matches. These environments were seeded differently for each of the three runs. The evaluation process yielded 36 data points per training cycle for environment-specific metrics, specifically the metric of `Episode Length`. Additionally, by considering each player separately, there were 72 data points for player-specific metrics, including `Ice Transferred` and `Average Factory Alive`. The lines on the charts visually represent the metrics' means, while the shaded area shows the standard deviation. In addition to the charts, tables are included to make comparisons easier. The columns `Final Ice Transferred` and `Final Episode Length` refer to the metrics collected at the last evaluation of the model. The mean value is presented, along with the standard deviation in brackets. The `X% of Episodes Finished by` metrics measure how long it took for the agents to learn how to keep the factories alive until the end of the game in terms of environment steps. The percentage indicates the ratio of evaluation environments that resulted in a finished episode, which is an episode where at least one factory survived from both players until 1,000 steps.

3.3.1 Trajectory separation

3.3.1.1 Global Trajectory vs Separated Trajectories

An initial experiment was done to see what effect **separated trajectories** (subsection 2.4.5) had on how fast the agents learned. As the control group, we implemented a variant that closely resembles the original pixel-to-pixel method (Chen *et al.* 2023) by utilizing a single **global group** (`"single global trajectory"`), resulting in a centralized architecture. To evaluate our more decentralized approach, we experimented with various group configurations. These configurations included the formation of distinct **global groups** for each entity type (`"global factories, global units"`), the creation of a single global group for one entity type while maintaining individual groups for the other (`"global factories, separate units"`, `"separate factories, global units"`), and the establishment of **individual groups** for each entity (`"separate trajectories for all entities"`). As demonstrated in Figure 3.4 and Table 3.3, the variants with separated trajectories perform much better compared to the global trajectory variants. The most notable improvement is the rate of convergence, which resulted in learning to keep the factories alive until the end of the game by around the

50,000th step, which the fully global variant could not achieve at all. Upon arriving at this significant milestone, the agents continued to improve their strategies further, leading to an average transfer of more than 40,000 ice to factories throughout the 1,000 steps in an episode. The value is 13 times greater than the achievement of the global trajectory variant. The accomplishment was reached by acquiring the skill to transfer ice to all of the factories rather than solely to one, which would have been sufficient to reach the maximum episode length. We can see this by the `Average Factory Alive` metric, which continues to rise even after the `Episode Length` levels off at the maximum possible value. The new technique also reduced the frequent fluctuations of the model's performance, which hinted at unlearning the progress made by the previous steps. We suspect this was the cause of destructive behavior reinforcement, which our method avoids since agents are only rewarded for their actual work (explained in subsection 3.3.3). The findings of the experiment indicate that utilizing separate trajectories can offer a notable benefit to the agents' training process.

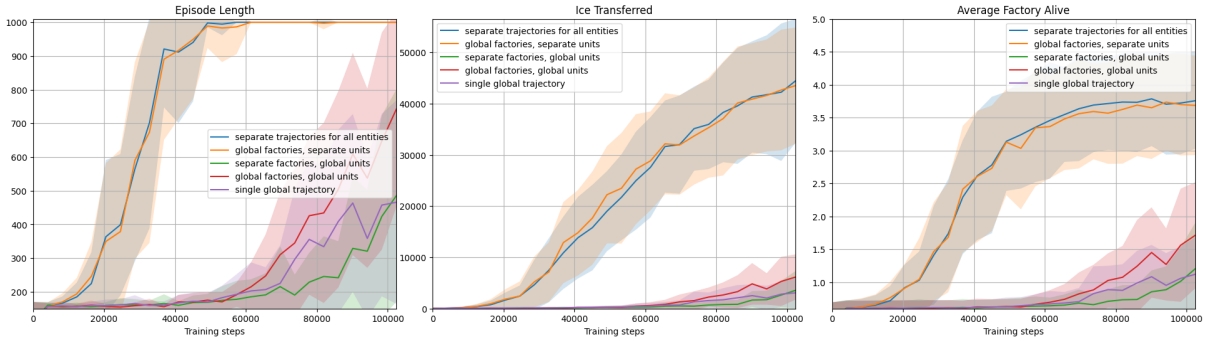


Figure 3.4: Plot comparing the usage of trajectory separation and global trajectories in terms of the length of the episodes, ice transferred by units, and number of active factories. The data presented in the figure indicates a clear improvement in all metrics thanks to the utilization of trajectory separation. In addition to faster convergence, the variants with separated trajectories exhibit reduced fluctuations in performance, thereby leading to more efficient training.

Group Name	Final Ice Transferred	Final Episode Length	10% of Episodes Finished by	95% of Episodes Finished by
separate trajectories for all entities	44,472 (12,065)	1,000 (0)	28,672 steps	49,152 steps
global factories, separate units	43,528 (11,327)	1,000 (0)	28,672 steps	49,152 steps
separate factories, global units	3,573 (3,753)	485 (315)	98,304 steps	-
global factories, global units	6,162 (4,413)	742 (290)	81,920 steps	-
single global trajectory	3,066 (2,996)	466 (302)	102,400 steps	-

Table 3.3: Table comparing the usage of trajectory separation and global trajectories. The metrics featured include the amount of ice transferred by units and the length of the episodes in the evaluation phase following the last training cycle. The table also contains the observed environment steps needed until the model reaches the maximum episode length in the specified percentage of evaluation environments. The variant with completely separate trajectories was able to provide the factories with 13 times more ice after the training and has managed to reach the maximum episode length in 10% of the evaluation environments more than 3 times faster.

3.3.1.2 Number of Optimal Trajectories

In order to demonstrate the significance of separating the entity trajectories, we conducted an additional experiment in which we categorized the units into N groups based on the modulus of their unique IDs. This categorization led to approximately random allocation across the groups, enabling us to examine the optimal number of trajectories to divide our training steps into. The units were the only entities involved in the grouping, as the factories remained entirely separate during this test. The results are displayed in Figure 3.5 and Table 3.4. It is evident that increasing the number of trajectories resulted in a faster and more efficient training process for the agents, suggesting that the optimal training method is utilizing a separate trajectory for all entities.

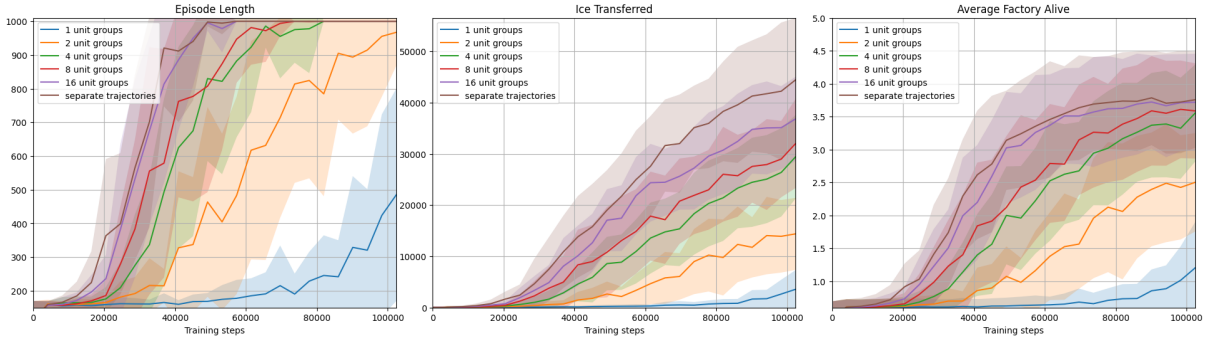


Figure 3.5: Plot comparing the performance of separating the global trajectory into N separate trajectories in terms of the length of the episodes, ice transferred by units, and number of active factories. In addition to the test variants, the global and completely separate trajectory variants are also present. The figure indicates a positive correlation between the number of separate trajectories and the performance achieved on all metrics.

Group Name	Final Ice Transferred	Final Episode Length	10% of Episodes Finished by	95% of Episodes Finished by
1 unit groups	3,573 (3,753)	485 (315)	98,304 steps	-
2 unit groups	14,388 (7,019)	967 (99)	57,344 steps	-
4 unit groups	29,459 (8,150)	1,000 (0)	36,864 steps	77,824 steps
8 unit groups	32,049 (8,701)	1,000 (0)	32,768 steps	61,440 steps
16 unit groups	36,847 (8,224)	1,000 (0)	28,672 steps	57,344 steps
separate trajectories	44,472 (12,065)	1,000 (0)	28,672 steps	49,152 steps

Table 3.4: Table comparing the performance of separating the global trajectory into N separate trajectories. The metrics featured include the amount of ice transferred by units and the length of the episodes in the evaluation phase following the last training cycle. The table also contains the observed environment steps needed until the model reaches the maximum episode length in the specified percentage of evaluation environments. In addition to the test variants, the global and completely separate trajectory variants are also present. The table shows a massive jump in the final average episode length even by separating the global trajectory into two random groups. While the episode length metric reaches its maximum value with 4 unit groups, an increased convergence rate can be observed by using even more separate trajectories.

3.3.2 Trajectory Number Reduction

The technique of **trajectory separation** (subsection 3.3.1) provides an optimized approach that enables efficient training of a centralized model for a multi-agent problem. However, it should be noted that the large number of trajectories does have certain limitations. Given that the separated **training examples** are tied to their respective environment steps, which are used to partition our **mini-batches** during training, and considering that the number of trajectories can vary at each step, it is possible for the mini-batches to have an imbalanced number of training examples. Given that the sizes of the matrices remain constant and padding values are used for inactive groups, this does not pose any issues regarding vectorization or memory utilization. Nevertheless, this phenomenon could potentially lead to an asymmetry in the impact that each training example has on the policy loss, as the final loss is obtained by averaging all training examples. Steps with a large number of currently active groups will carry more significance, whereas the impact of a group's example within a step with a low number of active groups will be more pronounced compared to a group's example within a step with a higher population. This is because, on average, the former will be included in a mini-batch with fewer examples. Even if the mini-batch sizes were balanced, it would remain uncertain whether the increased number of trajectories is advantageous for the training process. The generation of a large number of training instances from the available environment steps introduces a regularization effect to the model's training process, thereby reducing the influence of each individual instance. If the primary challenge associated with a single global trajectory is the complex action space and the incomprehensible singular reward value, it is possible that by eliminating these factors, a reduced quantity of training examples may be sufficient to obtain convergence. Reducing the size of the batches while maintaining the same performance would further optimize our technique. In order to assess the impact of the aforementioned issues, we will employ two methodologies: agent grouping, wherein we will evaluate the performance attainable through various grouping criteria, and trajectory sample reduction, wherein we will selectively utilize data from the top or bottom N groups based on a specified metric for each step in the environment.

3.3.2.1 Grouping

The results of the modulus test in subsubsection 3.3.1.2 show a strong link between performance and the number of allowed separate trajectories. The variant that was unconstrained by any kind of grouping outperformed all the other variants. However, it raises the question of whether comparable performance can be attained with a bounded variant by implementing a grouping rule that is beneficial for the training process. As shown in Figure 3.6 and Table 3.5, our experiment involved evaluating the effectiveness of three different grouping strategies: grouping by map segments, grouping by closest factories, and grouping by closest heavy units. All of these groupings led to a bounded number of trajectories, as the size of the map remains constant, and the number of factories can only vary between three and five. At first glance, the number of heavy units may appear to be limitless; however, the resources available to a factory only permit the production of a single heavy unit per factory. None of the groups that were tested were able to achieve performance comparable to that of the completely separated variant. Additionally, the number of groups remained a significant indicator of performance. We used the variants from Figure 3.5 to see if non-random grouping rules can do better than a random group assignment with the same number of separated groups. Unfortunately, all of the variants that were subjected to testing demonstrated lower performance or demonstrated comparable results to those assigned randomly. The sole exception was the `"closest heavy"`; however, it still failed to outperform random assignment in terms of converging to the maximum episode length. The

findings indicate that the approach of grouping the trajectories of entities together in order to reduce the number of training examples yields poor performance.

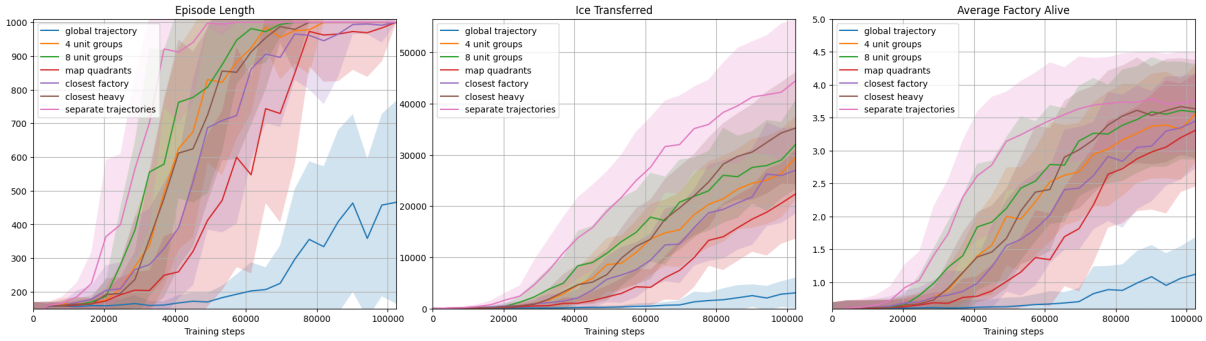


Figure 3.6: Plot comparing the usage of different grouping rules in terms of the length of the episodes, ice transferred by units, and number of active factories. In addition to the test variants, the global and completely separate trajectory variants are also present. None of the tried variants managed to beat the separate trajectory variant in performance.

Group Name	Final Ice Transferred	Final Episode Length	10% of Episodes Finished by	95% of Episodes Finished by
global trajectory	3,066 (2,996)	466 (302)	102,400 steps	-
4 unit groups	29,459 (8,150)	1,000 (0)	36,864 steps	77,824 steps
8 unit groups	32,049 (8,701)	1,000 (0)	32,768 steps	61,440 steps
map quadrants	22,385 (8,733)	1,000 (0)	53,248 steps	98,304 steps
closest factory	27,009 (8,420)	1,000 (0)	45,056 steps	94,208 steps
closest heavy	35,224 (10,946)	1,000 (0)	36,864 steps	73,728 steps
separate trajectories	44,472 (12,065)	1,000 (0)	28,672 steps	49,152 steps

Table 3.5: Table comparing the usage of different grouping rules. The metrics featured include the amount of ice transferred by units and the length of the episodes in the evaluation phase following the last training cycle. The table also contains the observed environment steps needed until the model reaches the maximum episode length in the specified percentage of evaluation environments. In addition to the test variants, the global and completely separate trajectory variants are also present. While each grouping configuration performed better than the global trajectory variant, none could compete with the convergence rate of the completely separate trajectory variant. Grouping units by specific rules did not perform better than random grouping.

3.3.2.2 Train Sample Reduction

Given that all grouping methods performed less effectively than completely separate trajectories for all entities, we attempted an alternative approach to ensure a more consistent number of training examples at each step. Our technique, referred to as **train sample reduction**, involves selecting the top N examples from each step and training the model exclusively on those chosen examples. This action accomplishes two objectives. Firstly, it establishes an **upper limit for the train examples**, thereby ensuring that the influence of each group's action remains

relatively consistent, regardless of the number of other active groups. Secondly, in theory, by selecting only the most relevant examples at each step, we should be capable of **accelerating the training process** of the model in terms of the number of environment steps and the amount of time required. In the following experiments, we utilize completely separated trajectories for all variants and perform the sample reduction solely on the train examples of units.

In order to first determine the optimal metric for selecting the desired subset of examples by, we conducted an experiment. We tried selecting the top N=1 samples by using various sampling techniques, including **random sampling**, **top advantage value**, **top absolute advantage value** and **top return value**. Among all of these variations, the subset that includes the examples with the greatest advantage value achieved the highest performance, as evidenced by the results presented in Figure 3.7 and Table 3.6. Interestingly, utilizing solely the example with the **most prominent advantage value** exhibited **almost the same performance** as employing all of the train examples. Consequently, we decided to investigate sampling by this metric further, with different N values.

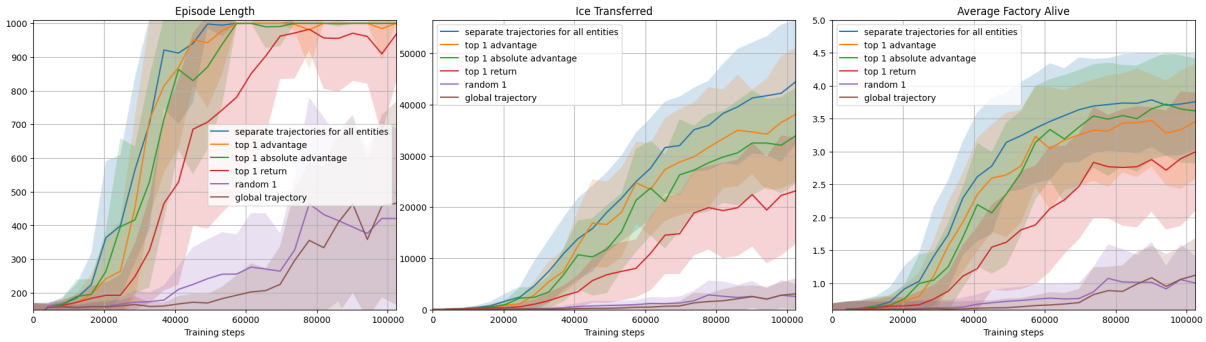


Figure 3.7: Plot comparing the different methods of trajectory sample reduction in terms of the length of the episodes, ice transferred by units, and number of active factories. In addition to the test variants, the global and completely separate trajectory variants are also present. Randomly selecting a single example at each step has the same effect as using a global trajectory. Out of all the sampling methods tried, selecting by advantage is the most prominent, leading to almost the same performance as using all data.

We tested the impact of selecting different numbers of examples with the highest advantage values from each environment step. Specifically, we tested the effects of taking 1, 2, 4, 8, and 16 such examples. The results are presented in Figure 3.8 and Table 3.7. Curiously, the inclusion of additional examples does not appear to significantly enhance performance. In fact, the performance is slightly hindered when the sample size is increased to 16. This implies that the significance of an entity's action may be diminished by the multitude of other actions currently occurring in the surrounding environment, making the action less dominant in the loss calculation. By reducing the number of separate training examples and subsequently removing unnecessary data beforehand, we can help the convergence of the model. The slight edge exhibited by the sampled variants could also potentially be attributed to avoiding the problem of changing batch sizes, although a definitive conclusion cannot be drawn.

We were also interested in investigating the performance that can be achieved through random sampling. We conducted an experiment examining the performance of different numbers of random samples, specifically 1, 2, 4, 8, and 16. The aforementioned variants were subsequently

Group Name	Final Ice Transferred	Final Episode Length	10% of Episodes Finished by	95% of Episodes Finished by
separate trajectories for all entities	44,472 (12,065)	1,000 (0)	28,672 steps	49,152 steps
top 1 advantage	38,072 (13,071)	1,000 (0)	32,768 steps	57,344 steps
top 1 absolute advantage	33,926 (9,324)	1,000 (0)	32,768 steps	57,344 steps
top 1 return	23,156 (10,369)	968 (138)	40,960 steps	-
random 1	2,550 (2,434)	420 (269)	77,824 steps	-
global trajectory	3,066 (2,996)	466 (302)	102,400 steps	-

Table 3.6: Table comparing the different methods of trajectory sample reduction. The metrics featured include the amount of ice transferred by units and the length of the episodes in the evaluation phase following the last training cycle. The table also contains the observed environment steps needed until the model reaches the maximum episode length in the specified percentage of evaluation environments. In addition to the test variants, the global and completely separate trajectory variants are also present. The table shows how selecting a single train example from each step by the right metric could approximate the training performance on all data. The advantage value proved to be the best metric for sampling.

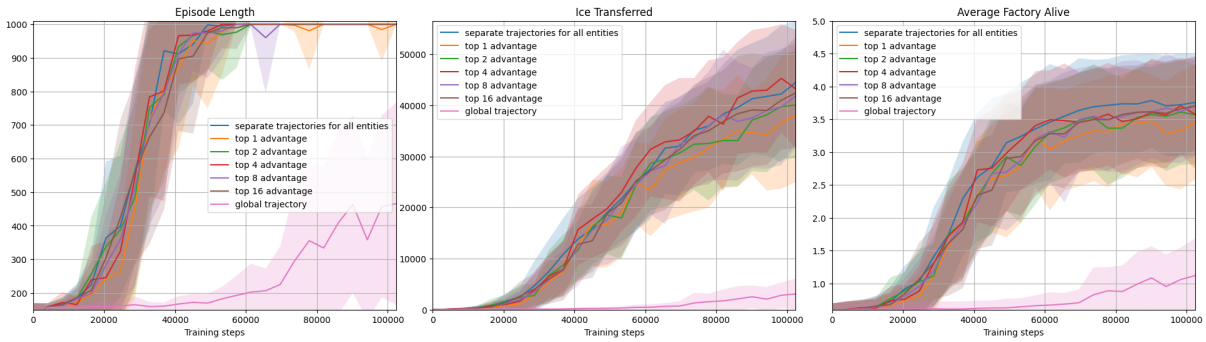


Figure 3.8: Plot comparing the performance of sampling train examples based on the advantage values in terms of the length of the episodes, ice transferred by units, and number of active factories. In addition to the test variants, the global and completely separate trajectory variants are also present. Increasing the number of sampled values appears to have minimal effect on performance, leading us to believe that the main benefit of utilizing a larger training set is to find outstandingly beneficial examples.

Group Name	Final Ice Transferred	Final Episode Length	10% of Episodes Finished by	95% of Episodes Finished by
separate trajectories for all entities	44,472 (12,065)	1,000 (0)	28,672 steps	49,152 steps
top 1 advantage	38,072 (13,071)	1,000 (0)	32,768 steps	57,344 steps
top 2 advantage	40,053 (10,402)	1,000 (0)	24,576 steps	61,440 steps
top 4 advantage	43,246 (11,518)	1,000 (0)	28,672 steps	49,152 steps
top 8 advantage	41,855 (10,159)	1,000 (0)	28,672 steps	57,344 steps
top 16 advantage	42,433 (10,792)	1,000 (0)	24,576 steps	53,248 steps
global trajectory	3,066 (2,996)	466 (302)	102,400 steps	-

Table 3.7: Table comparing the performance of sampling train examples based on the advantage values. The metrics featured include the amount of ice transferred by units and the length of the episodes in the evaluation phase following the last training cycle. The table also contains the observed environment steps needed until the model reaches the maximum episode length in the specified percentage of evaluation environments. In addition to the test variants, the global and completely separate trajectory variants are also present. Increasing the number of advantage-sampled values did not appear to have a significant effect on performance.

compared to the established, completely separate, and global variants. The results are available in Figure 3.9 and Table 3.8. Using a random sampling approach cannot provide comparable performance to utilizing the entire dataset. As expected, the performance improved as we increased the number of sampled examples, consistent with the random grouping observed in the trajectory separation experiment (subsubsection 3.3.1.2).

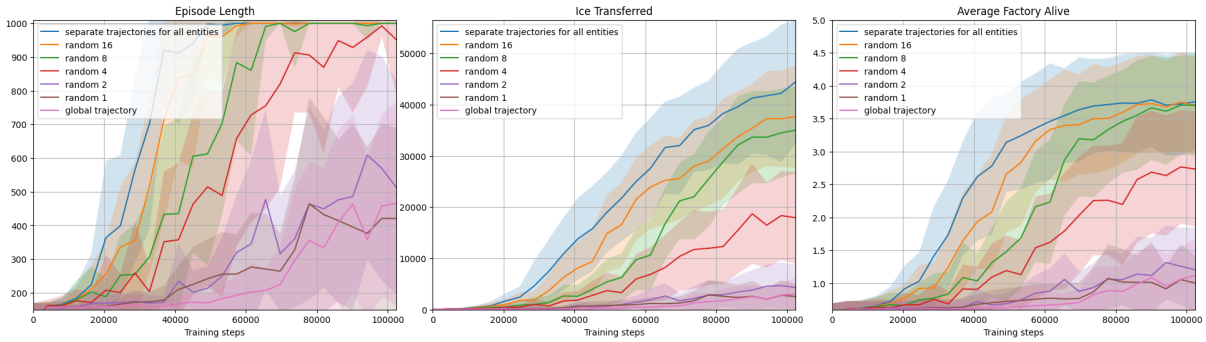


Figure 3.9: Plot showcasing the performance of random train example sampling in terms of the length of the episodes, ice transferred by units, and number of active factories. In addition to the test variants, the global and completely separate trajectory variants are also present. As expected, more randomly sampled trajectories caused better performance.

3.3.2.3 Emphasizing Data Quality over Quantity

In subsubsection 3.3.2.2, we observed that the act of sampling a single training example from an environment step with separate trajectories yields the same suboptimal outcome as utilizing a global trajectory. Moreover, employing a sample selection method based on a reward-like metric can yield comparable performance to training on the entire dataset. From our perspective,

Group Name	Final Ice Transferred	Final Episode Length	10% of Episodes Finished by	95% of Episodes Finished by
separate trajectories for all entities	44,472 (12,065)	1,000 (0)	28,672 steps	49,152 steps
random 16	37,665 (9,965)	1,000 (0)	32,768 steps	61,440 steps
random 8	35,022 (8,492)	1,000 (0)	45,056 steps	69,632 steps
random 4	17,927 (9,007)	951 (160)	45,056 steps	-
random 2	4,284 (4,475)	513 (317)	77,824 steps	-
random 1	2,550 (2,434)	420 (269)	77,824 steps	-
global trajectory	3,066 (2,996)	466 (302)	102,400 steps	-

Table 3.8: Table showcasing the performance of random train example sampling. The metrics featured include the amount of ice transferred by units and the length of the episodes in the evaluation phase following the last training cycle. The table also contains the observed environment steps needed until the model reaches the maximum episode length in the specified percentage of evaluation environments. In addition to the test variants, the global and completely separate trajectory variants are also present. As expected, more sampled trajectories resulted in better performance.

this suggests the significance of data quality over data quantity. In order to further examine this concept, we conducted a comparison between **train sample reduction** and **random grouping** (subsubsection 3.3.1.2). The results are presented in Figure 3.10 and Table 3.9. In this study, the two observed methods of training sample reduction were random sampling, since it is the most easily comparable to random grouping, and advantage-based selection, which achieved the best performance in the trajectory separation test (subsubsection 3.3.2.2). We can derive from these charts that increasing the number of training examples does not necessarily lead to improved performance, as advantage-based selection managed to beat random sampling even with a smaller sample size. This shows that the primary purpose of using a large amount of data in reinforcement learning is to identify actions that are clearly more favorable. In the experiment, both grouping and trajectory separation resulted in a maximum of 4 training examples per environment step. However, it is important to note that grouping incorporates data from all entities, although it poses challenges in terms of learnability, as discussed in subsection 3.3.3. This increased complexity of the training data leads to a significant decrease in performance, to the extent that even random sampling, which only includes a small subset of all entity data, can achieve similar performance. Furthermore, when we switch to advantage-based sampling, which learns only from the pre-selected most relevant examples, its training massively outperforms the grouped variant's. This further confirms the significance of data quality.

3.3.3 Method Components

Although the results obtained in subsection 3.3.1 seem impressive, it is challenging to identify the specific factor that has led to such a substantial improvement in performance. Based on our current understanding, the act of separating the trajectories leads to four notable enhancements in the training process. The process generates additional **training examples** from the same environmental steps, thereby enhancing the model's ability to acquire broader policies at an accelerated rate through increased exploration. Another, perhaps the most noteworthy ad-

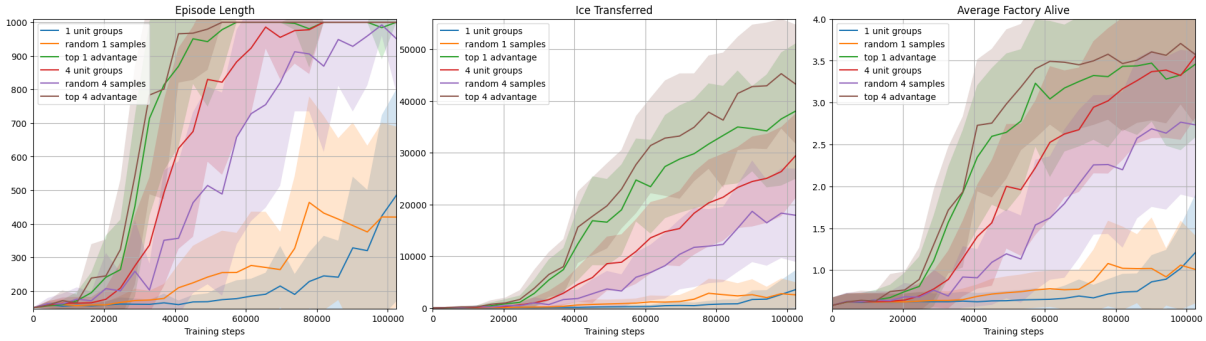


Figure 3.10: Plot comparing random grouping, random sampling, and advantage-based sampling in terms of the length of the episodes, ice transferred by units, and number of active factories. Sampling by advantage outperforms both random grouping and random sampling, even if the advantage-based sampling contains fewer trajectories, suggesting greater importance of data quality than quantity.

Group Name	Final Ice Transferred	Final Episode Length	10% of Episodes Finished by	95% of Episodes Finished by
1 unit groups	3,573 (3,753)	485 (315)	98,304 steps	-
random 1 samples	2,550 (2,434)	420 (269)	77,824 steps	-
top 1 advantage	38,072 (13,071)	1,000 (0)	32,768 steps	57,344 steps
4 unit groups	29,459 (8,150)	1,000 (0)	36,864 steps	77,824 steps
random 4 samples	17,927 (9,007)	951 (160)	45,056 steps	-
top 4 advantage	43,246 (11,518)	1,000 (0)	28,672 steps	49,152 steps

Table 3.9: Table comparing random grouping, random sampling, and advantage-based sampling. The metrics featured include the amount of ice transferred by units and the length of the episodes in the evaluation phase following the last training cycle. The table also contains the observed environment steps needed until the model reaches the maximum episode length in the specified percentage of evaluation environments. The advantage-based sampling managed to outperform the other variants in all metrics, even with fewer training examples, while the random sampling underperformed random grouping.

vantage, is the eradication of the **reinforcement of undesirable behavior**. Such reinforcement could arise if one entity performs an action that results in a high advantage value while another entity behaves in a manner that is suboptimal for the team's success but doesn't get punished by enough negative advantage. In this case, the incorrect action would be reinforced alongside the positive action. By employing separate trajectories with **localized rewards**, the occurrence of such events can be prevented. Our method also avoids calculating advantage from steps where the original performing entity is no longer active. If only a single trajectory were utilized, there would be no way to keep track of the status and reward of individual entities. Within highly dynamic environments, such as Lux, this phenomenon leads to a somewhat misleading calculation due to the significant probability that the units and factories currently in existence will not persist until the conclusion of the episode. Consequently, it is impossible to determine whether their actions in the present have truly contributed to any real advantage. By keeping the rewards separate, we ensure that the entity receives credit for its immediate and observable efforts, both in the present and future. Lastly, separate trajectories result in separate **advantage** calculations and separate **critic value** outputs for all groups. Estimating the entire game state in order to provide a final global value is a significantly more complex task for the model compared to predicting multiple values based on local observations. Within this subsection, we examine the different components of trajectory separation to determine which part has the greatest impact on enhancing performance.

3.3.3.1 Inividual Components

We initiated the study by performing a limited ablation analysis on the trajectory separation technique. Our initial hypothesis was that the most crucial components that needed to be separated were the **reward**, **terminal state** indication, **critic value**, **log probabilities**, and **entropy** values. Therefore, we conducted a series of experiments where we systematically removed the separation of these components one by one. The purpose of these experiments was to observe the impact of each component's loss on the model's overall performance.

Both the **reward** and **critic** values are essential for the computation of the **advantage**, an integral component of the Proximal Policy Optimization (subsection 2.1.8) algorithm. Therefore, we decided to start with removing separation between these values. The results can be seen in Figure 3.11 and Table 3.10. The metrics for the `"separate trajectories, global reward"` variant indicate that the utilization of separate trajectories with global rewards leads to a notable decrease in convergence speed, in comparison to separate trajectories with **localized rewards**. This negative effect was expected since the absence of separated rewards hinders the model's ability to identify the specific actions undertaken by the units and factories that led to the rewards received. Consequently, this issue aligns with the problem outlined in subsection 2.4.5. In order to achieve efficient global advantage calculation, the critic head of the model must learn how to predict the performance of the entire team rather than solely relying on predictions based on the local observations of individual entities. The decline in performance due to losing the ability to calculate **critic value** predictions based on local information is further indicated by the `"separate trajectories, global critic"` variant. The agents were able to sustain the operation of the factories for up to 1,000 steps in most environments, albeit at a slower pace, demonstrating how harder it is for the model to work solely from global information. Only when we remove both the **critic values** and **rewards** in test `"separate trajectories, global advantage"` do we observe a reduction in performance to the level of a single global trajectory. A fully global **advantage** calculation results in the same advantage value output for all entities, meaning there is no way to differentiate between positive and

negative actions under the same step, massively slowing down the training. Following this modification, the only difference in the policy loss between the entities is the **probability ratios**, which fail to provide us any information about the desired direction and magnitude of policy updates. Due to the abovementioned factors, we believe that separate rewards hold significant importance. The impact of their effect is further examined in subsubsection 3.3.3.2.

The outcomes following the elimination of the remaining separation components, namely the termination flags and the action probabilities, are observable in Figure 3.12 and Table 3.11. The fact that the performance remained consistent even after the removal of the separated `terminal state flags` was unexpected and surprising to us. This group can be seen under the name `"separate trajectories, global dones"`. Given that the trajectories are separated, and only currently active entities receive rewards, indicating episode terminations prematurely at the destruction of an entity is somewhat redundant since there will be no future rewards assigned to them that could skew the return calculation anyway. The act of indicating the inactivity of a group of agents is more logical in situations where the group has the potential to become active once again, particularly in cases where multiple entities are present within the same group. If all entities within the group are destroyed, but subsequently, a new entity is assigned, it is possible for the group to regain its activity. Whether stopping the future reward calculation between these two group states is beneficial is yet to be explored. The separation of distribution entropy values does not appear to have any significance, as these values are ultimately combined into a single value to calculate the entropy loss regularization term. What massively degraded performance, however, was the removal of the separated log probabilities. Through the aggregation of log probabilities, the gradients can propagate across the action calculations of all entities. Similarly to the combined advantage value, this global gradient propagation causes weight adjustments based on the whole team's performance, making it impossible to separate individual contributions. This observation shows the necessity of utilizing separate action probabilities in order to achieve optimal training.

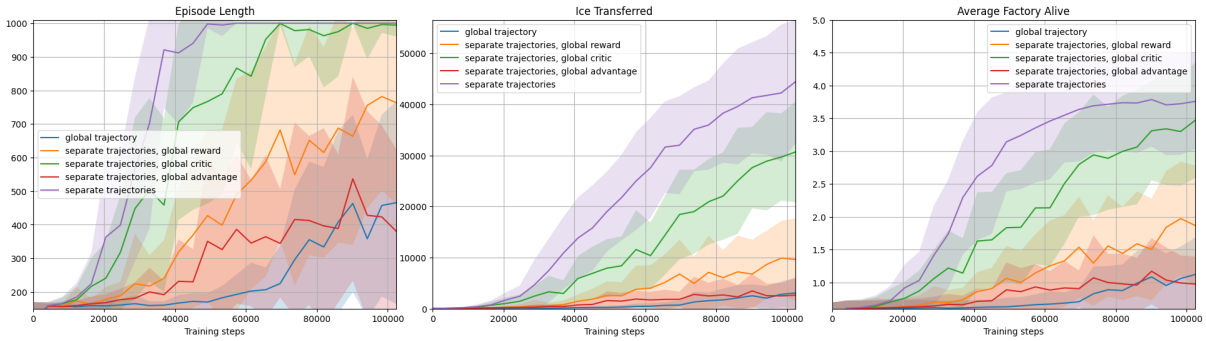


Figure 3.11: Plot comparing the removal of separated components relevant to the advantage calculation in terms of the length of the episodes, ice transferred by units, and number of active factories. In addition to the test variants, the global and completely separate trajectory variants are also present. The separated reward plays a significant role but doesn't account for the whole performance boost. Removing separated critic value predictions and separated rewards together results in a similar performance to a single global trajectory.

3.3.3.2 Reward Assignment

Based on the findings reported in subsubsection 3.3.3.1, which indicated a significant impact of reward separation on performance, our objective was to investigate the importance of **localized**

Group Name	Final Ice Transferred	Final Episode Length	10% of Episodes Finished by	95% of Episodes Finished by
global trajectory	3,066 (2,996)	466 (302)	102,400 steps	-
separate trajectories, global reward	9,614 (8,094)	763 (300)	53,248 steps	-
separate trajectories, global critic	30,687 (9,844)	994 (34)	40,960 steps	69,632 steps
separate trajectories, global advantage	2,643 (3,353)	380 (244)	73,728 steps	-
separate trajectories	44,472 (12,065)	1,000 (0)	28,672 steps	49,152 steps

Table 3.10: Table comparing the removal of separated components relevant to the advantage calculation. The metrics featured include the amount of ice transferred by units and the length of the episodes in the evaluation phase following the last training cycle. The table also contains the observed environment steps needed until the model reaches the maximum episode length in the specified percentage of evaluation environments. In addition to the test variants, the global and completely separate trajectory variants are also present. Removing the separation of rewards and critic values caused a significant performance decrease. Reward appears to be the most important component.

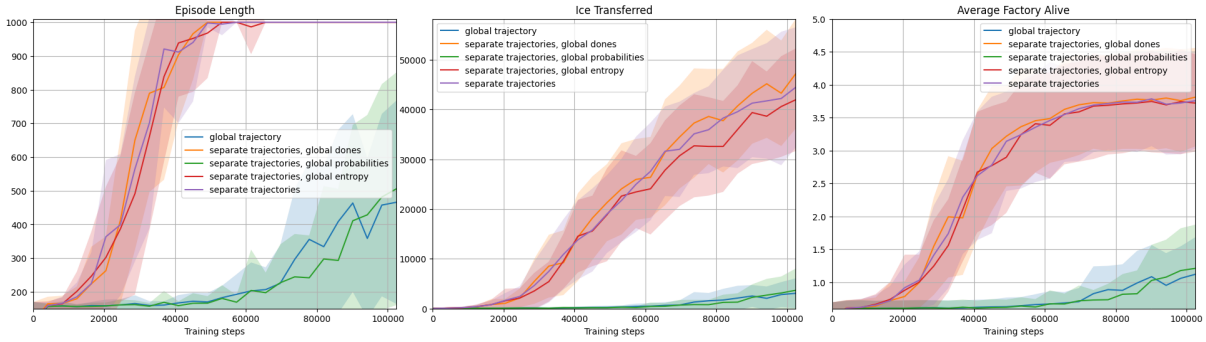


Figure 3.12: Plot comparing the removal of other separated components in terms of the length of the episodes, ice transferred by units, and number of active factories. In addition to the test variants, the global and completely separate trajectory variants are also present. Aggregating the action probabilities globally degrades the model's performance to the level of a single global trajectory.

Group Name	Final Ice Transferred	Final Episode Length	10% of Episodes Finished by	95% of Episodes Finished by
global trajectory	3,066 (2,996)	466 (302)	102,400 steps	-
separate trajectories, global dones	47,201 (11,018)	1,000 (0)	28,672 steps	49,152 steps
separate trajectories, global probabilities	3,673 (4,372)	506 (345)	98,304 steps	-
separate trajectories, global entropy	41,970 (10,305)	1,000 (0)	28,672 steps	53,248 steps
separate trajectories	44,472 (12,065)	1,000 (0)	28,672 steps	49,152 steps

Table 3.11: Table comparing the removal of other separated components. The metrics featured include the amount of ice transferred by units and the length of the episodes in the evaluation phase following the last training cycle. The table also contains the observed environment steps needed until the model reaches the maximum episode length in the specified percentage of evaluation environments. In addition to the test variants, the global and completely separate trajectory variants are also present. The table demonstrates the importance of separate action probabilities in order to limit the gradient flow to only relevant parts of the network during training.

rewards. To achieve this, we conducted an experiment combining the local, separated reward with the aggregated **global reward**. Agents were allocated different percentages of their own rewards, specifically 0%, 50%, 75%, and 100% of their own rewards, with the remaining percentages being made up by the global reward. As evident from the data presented in Figure 3.13 and Table 3.12, the variant with completely separate reward values demonstrates superior performance compared to the other variants. Utilizing 25% of the global rewards appears to result in convergence to an episode length of 1,000 at approximately the same step. Nevertheless, it gets outperformed in the `Ice Transferred` and `Average Factory Alive` metrics by the completely separate variant. The variant that receives 50% global rewards achieves the maximum episode length on average, whereas the variant that solely relies on global rewards fails to converge even after 100,000 steps. This experiment provides additional evidence to support the significance of localized rewards.

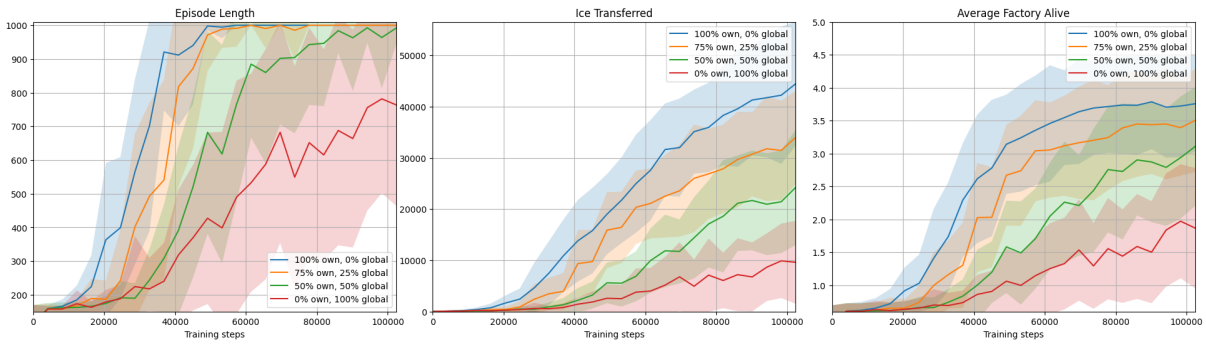


Figure 3.13: Plot comparing the performance of different percentages of global and local rewards in terms of the length of the episodes, ice transferred by units, and number of active factories. In addition to the test variants, the global and completely separate trajectory variants are also present. Mixing in global rewards seems to hinder performance.

Group Name	Final Ice Transferred	Final Episode Length	10% of Episodes Finished by	95% of Episodes Finished by
100% own, 0% global	44,472 (12,065)	1,000 (0)	28,672 steps	49,152 steps
75% own, 25% global	33,985 (9,059)	1,000 (0)	28,672 steps	57,344 steps
50% own, 50% global	24,244 (11,238)	992 (47)	45,056 steps	94,208 steps
0% own, 100% global	9,614 (8,094)	763 (300)	53,248 steps	-

Table 3.12: Table comparing the performance of different global and local rewards percentages. The metrics featured include the amount of ice transferred by units and the length of the episodes in the evaluation phase following the last training cycle. The table also contains the observed environment steps needed until the model reaches the maximum episode length in the specified percentage of evaluation environments. In addition to the test variants, the global and completely separate trajectory variants are also present.

3.3.4 Comparison with other works

As previously stated in this section, the training objectives in the experiments demonstrating trajectory separation were focused on achieving the maximum episode length, which can be accomplished by keeping at least one factory alive from both players. Factories have a lichen watering action (subsection 1.4.7), which is used to determine the winner if both players successfully maintain their factories until the maximum duration of the episode. In the experiments, we masked out (subsection 2.4.3) this watering action in order to achieve faster convergence. This decision was made since watering can reduce the potential lifespan of a factory. Consequently, in order to conduct a comparative analysis between our approach and state-of-the-art solutions, it was necessary to enable lichen watering. As a result, we proceeded to train a slightly different version of our model. We used trajectory separation (subsection 3.3.1) without any kind of grouping rule (subsection 2.4.5) or trajectory sample reduction (subsubsection 3.3.2.2). Given that the other solutions gave rewards for growing lichen, we made adjustments to the reward functions so that the factories received rewards proportional to the number of lichen tiles present on the board at the end of the episode. Lichen can only be grown on rubbleless tiles; thus, we also rewarded units after clearing away rubble next to factories and lichen tiles. We measured how long it took for the modified model to learn how to keep the factories alive up to the maximum episode length. The performance of the modified model compared to the original can be observed in Figure 3.14 and Table 3.13. Even though enabling and rewarding the watering action caused a slight decrease in performance, the model still managed to learn how to transport sufficient ice to factories. It also learned how to successfully grow lichen, as can be seen in Figure 3.15.

We wanted to compare our solution to the top-performing submissions in the Lux AI competition. The basis of our comparison is how long it took for the different models to reach a state where they could keep the factories alive until the maximum episode length in most environments. In our case, this goal was 95% of episodes. Unfortunately, the leaderboards were predominantly controlled by rule-based methods, resulting in very few RL-based submissions that could be used for a comparative analysis. Many refrained from sharing their work, and in lots of cases where they did share it, their trained models or training metrics were not made publicly available. The only somewhat reliable source we could find was FLG's submission (Limburg 2023), which was somewhat similar to the pixel-to-pixel architecture we started with.

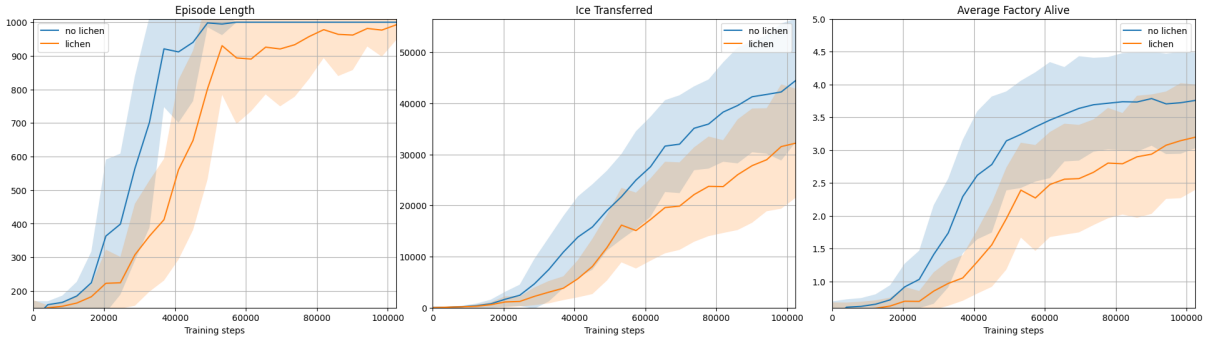


Figure 3.14: Plot comparing the performance of the new lichen-enabled model to the original in terms of the length of the episodes, ice transferred by units, and number of active factories. While lichen watering degrades performance, the goal of reaching the maximum episode length is still achieved at the end of the observed step range.

Group Name	Final Ice Transferred	Final Episode Length	10% of Episodes Finished by	95% of Episodes Finished by
no lichen	44,472 (12,065)	1,000 (0)	28,672 steps	49,152 steps
lichen	32,199 (10,670)	992 (45)	40,960 steps	102,400 steps

Table 3.13: Table comparing the performance of the new lichen-enabled model to the original. The metrics featured include the amount of ice transferred by units and the length of the episodes in the evaluation phase following the last training cycle. The table also contains the observed environment steps needed until the model reaches the maximum episode length in the specified percentage of evaluation environments. In addition to the test variants, the global and completely separate trajectory variants are also present. The lichen-enabled variant managed to reach the desired 95% of completed episodes by the end of the observed step range.

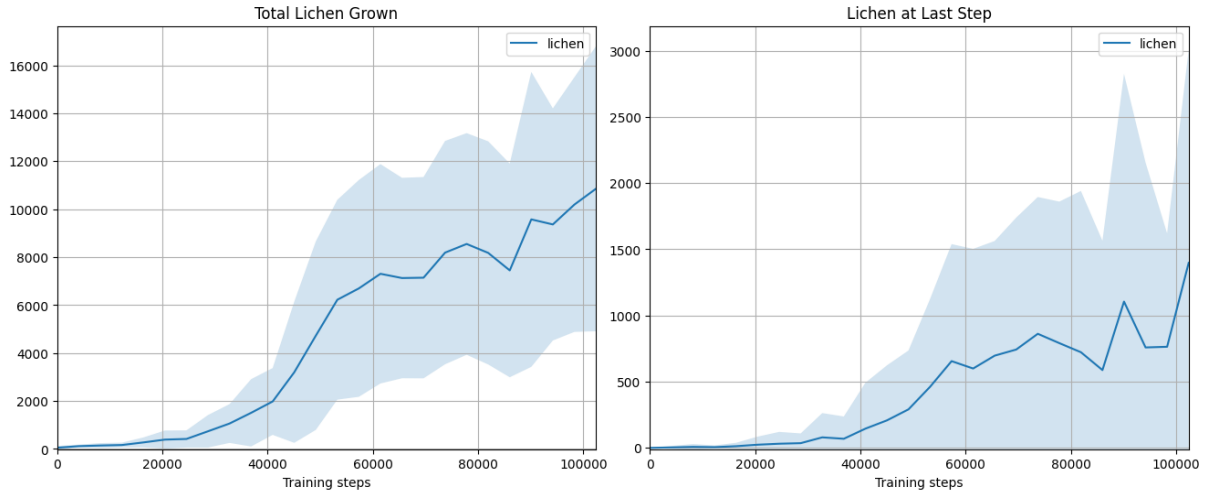


Figure 3.15: Plot showcasing the amount of total lichen grown and the amount of lichen at the end of the episodes. In addition to learning how to reach the maximum episode length (Figure 3.14), the agent demonstrated the ability to solve a secondary task, as shown by the ascending lines.

Their training objective during the initial "set-up phase" was comparable to ours, which aimed to teach the agent fundamental aspects of the game, such as gathering resources. The duration of their training in terms of time and steps was outlined; however, we are not sure what their stopping criteria were. Consequently, for the purpose of our comparison, we will assume that the stopping criterion used in the referenced submission was identical to our own. In addition, we included a baseline solution provided by the competition organizers (Kai Yang 2023). The comparison can be seen in Table 3.14. Our approach massively outperformed both the baseline and the FLG's submission in every single one of our metrics. We managed to reach the maximum episode length in 95% of the environments by step 102,400. In comparison, the baseline model required 1.4 million steps to reach the same milestone, while the submission of FLG was trained for 65 million steps. Given that we utilized a larger value for the hyperparameter "number of epochs" compared to the baseline, resulting in faster training, we believed it was necessary to account for this difference in our comparison. Having said that, our approach is still 5 times more efficient in terms of weight updates needed. Our approach also yielded significantly reduced model sizes, with a mere 200 thousand trainable parameters, as opposed to 451 thousand and 6.08 million, respectively. We also compared the training times, although it should be noted that the resources used for training the other two solutions remain undisclosed, potentially introducing a bias to the presented values.

Solution	**Parameters**	**Training time**	**Environment Steps**	**Training Epochs**
Trajectory Separated Hybrid Approach	**200K**	**2 hours**	**102K**	**250**
Baseline Solution (Kai Yang 2023)	451K	2 days	1.4M	1364
Best RL Submission (Limburg 2023)	6.08M	2 days*	65M*	?**

* exact stopping criterion is not known, only the number of steps the model was trained on and the time of training.
** "number of epochs" hyperparameter not provided.

Table 3.14: Table containing the comparison of our work with other implementations. We included the best-performing reinforcement learning submission of the Lux AI competition and a baseline repository provided by the organizers. Our method outperforms both of them in terms of both training time and observed environment steps needed to reach the designated goal. The table also shows a significant difference in model sizes.

3.3.5 Model Ablation Study

Through the utilization of trajectory separation (subsection 3.3.1), we achieved a significant acceleration in the training process of our model. However, as previously discussed in section 2.3 and section 2.4, many additional components played an important role in achieving the level of performance described, with certain components being absolutely indispensable. In this subsection, we performed an ablation study on the key methods employed to demonstrate the various factors that can impact reinforcement learning and the potential pitfalls that can arise.

3.3.5.1 Weight Initialization

The proper initialization of weights plays a crucial role, particularly in the application of algorithms such as Proximal Policy Optimization (subsection 2.1.8). We have provided an overview

of the weight initialization structure of our model in subsection 2.3.10 and will delve into this issue further in section 4.4. In this study, we aim to illustrate the impact of incorrect initialization on the convergence of our trajectory-separated hybrid (section 2.4) model. The study involved the examination of two components: the weight initialization scaling in the output layers and an additional scaling applied to the weights after initialization, referred to as "extra scaling". We tested removing the extra scaling from the hidden layers (`"no extra hidden layer scaling"`) as well as from all layers (`"no extra scaling"`). We also conducted experiments to evaluate the effects of removing any kind of scaling from the actor output layers (`"no actor scaling"`), the critic value output layers (`"no value scaling"`), and all output layers (`"no scaling"`). As presented in Figure 3.16 and Table 3.15, the removal of any weight initialization component leads to deterioration of performance. Eliminating all weight-downscaling techniques renders the model incapable of any kind of learning. The main problem with large weights is their effect on the initial policies. A network that exhibits a high degree of bias may not assign equal probabilities to all possible actions, resulting in a decrease in the exploration of different actions and a slower rate of learning. This biased network can sometimes lead to policy collapse, as observed with variant `"no actor scaling"`. Interestingly, removing the critic value output's scaling causes even larger performance degradation because the early parts of the training must be spent on training the value network to overcome its inherent bias. As a result, the predicted advantage values, which are essential for the Proximal Policy Optimization (PPO) algorithm, will be inaccurate.

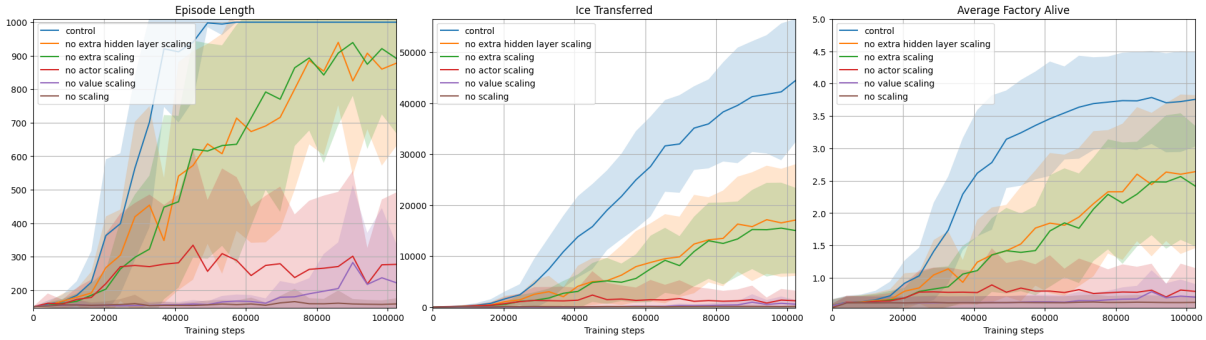


Figure 3.16: Plot showcasing the difference in performance weight magnitudes can cause in terms of the length of the episodes, ice transferred by units, and number of active factories. Removing any kind of weight downscaling caused a significant performance decrease. Not scaling down the weights of the output layer at all makes training impossible for the model.

3.3.5.2 Removing Heuristics

Our implementation of action masking (subsection 2.3.3) and factory placement (subsection 2.3.2) introduces a great level of heuristics into the system. Action masking decreases the action space, resulting in faster convergence. Additionally, the strategic placement of factories in close proximity to ice tiles simplifies the environment. Both of these factors play a crucial role, as can be observed in Figure 3.17 and Table 3.16. Without the implementation of action masking, the model is presented with an excessive number of potential combinations in its action space, leading to an inability to learn an optimal policy. Consequently, there was no significant increase observed in the `"Episode Length"` metric throughout the entire training period. By setting the factory placement to random, the agents can still learn how to transport

Group Name	Final Ice Transferred	Final Episode Length	10% of Episodes Finished by	95% of Episodes Finished by
control	44,472 (12,065)	1,000 (0)	28,672 steps	49,152 steps
no extra hidden layer scaling	17,093 (10,944)	878 (247)	32,768 steps	-
no extra scaling	15,019 (8,371)	892 (223)	36,864 steps	-
no actor scaling	1,255 (1,986)	277 (215)	45,056 steps	-
no value scaling	591 (811)	222 (116)	-	-
no scaling	145 (257)	159 (17)	-	-

Table 3.15: Table showcasing the difference in performance weight magnitudes can cause. The metrics featured include the amount of ice transferred by units and the length of the episodes in the evaluation phase following the last training cycle. The table also contains the observed environment steps needed until the model reaches the maximum episode length in the specified percentage of evaluation environments. The table clearly shows the decline in performance caused by taking away the weight scalings one by one. None of the studied variants managed to learn how to keep the factories alive until the maximum episode length in most environments.

ice to factories. However, due to the varying distances of ice tiles from factories in each episode, the units are unable to learn how to keep the factories alive within the observed step range.

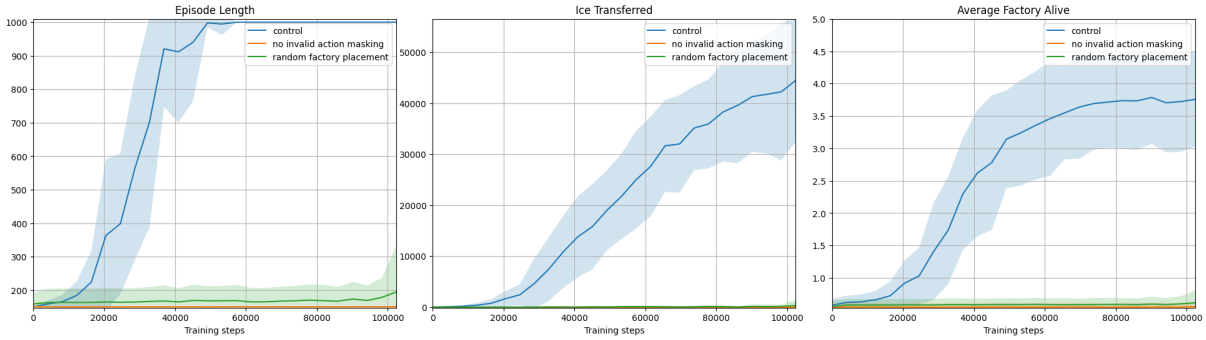


Figure 3.17: Plot showcasing the difference in performance that removing heuristics causes in terms of the length of the episodes, ice transferred by units, and number of active factories. The elimination of action masking renders training infeasible within the specified step range that we have presented. In addition, the placement of factories at random locations introduces complexity to the environment, resulting in a significant decline in performance.

3.3.5.3 Network Regularization

In our network, we incorporated two crucial regularization components: batch normalization (subsection 2.3.8) and spectral norm (subsection 2.3.9). In the following experiment, we tested the necessity of employing both regularization methods. As can be seen in Figure 3.18 and Table 3.17, it is apparent that the inclusion of some form of regularization is essential. Removing batch normalization resulted in our model's inability to learn how to reach the maximum episode length within the observed step range. Interestingly, eliminating the spectral norm by itself did not significantly affect performance. Nevertheless, in the absence of batch normalization,

Group Name	Final Ice Transferred	Final Episode Length	10% of Episodes Finished by	95% of Episodes Finished by
control	44,472 (12,065)	1,000 (0)	28,672 steps	49,152 steps
no invalid action masking	0 (0)	151 (0)	-	-
random factory placement	316 (1,055)	195 (137)	-	-

Table 3.16: Table showcasing the difference in performance that removing heuristics causes. The metrics featured include the amount of ice transferred by units and the length of the episodes in the evaluation phase following the last training cycle. The table also contains the observed environment steps needed until the model reaches the maximum episode length in the specified percentage of evaluation environments. Without action masking, the units were unable to transfer any ice to factories. Random factory placement still allowed the units to learn that mining and transferring ice are advantageous, but they could not perform said actions optimally.

the performance deteriorated even further when the spectral norm was removed. This observation suggests that the simultaneous use of both regularization methods may not be necessary. However, including at least one of them is crucial, preferably batch normalization.

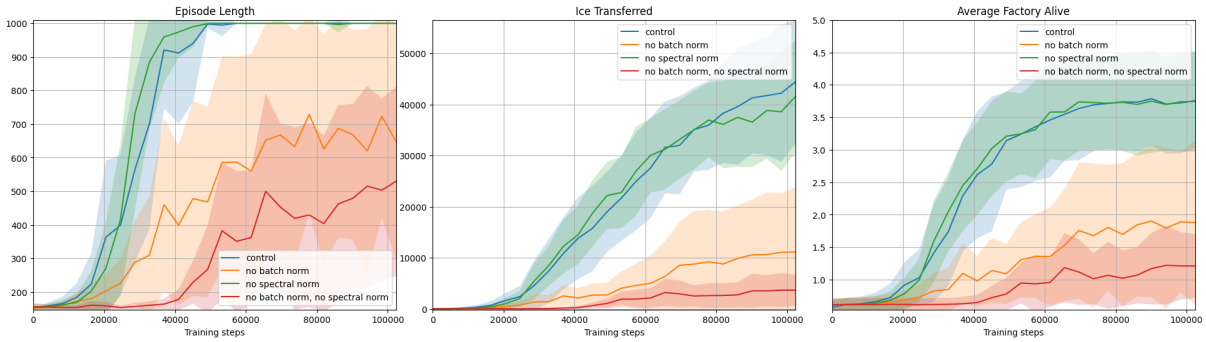


Figure 3.18: Plot showcasing the difference in performance that removing regularization from our network causes in terms of the length of the episodes, ice transferred by units, and number of active factories. It is clearly visible that batch normalization is the most important regularization component. However, if there is no batch normalization utilized in the network, the presence of spectral norm can help mitigate the need for regularization. Removing batch normalization appears to also cause high variance in the observed metrics.

Group Name	Final Ice Transferred	Final Episode Length	10% of Episodes Finished by	95% of Episodes Finished by
control	44,472 (12,065)	1,000 (0)	28,672 steps	49,152 steps
no batch norm	11,184 (12,688)	646 (351)	53,248 steps	-
no spectral norm	41,611 (10,938)	1,000 (0)	28,672 steps	45,056 steps
no batch norm, no spectral norm	3,696 (3,039)	530 (282)	65,536 steps	-

Table 3.17: Table showcasing the difference in performance that removing regularization from our network causes. The metrics featured include the amount of ice transferred by units and the length of the episodes in the evaluation phase following the last training cycle. The table also contains the observed environment steps needed until the model reaches the maximum episode length in the specified percentage of evaluation environments. Removing batch normalization decreases the model's performance significantly, making it unable to learn how to reach the maximum episode length in all environments. Removing spectral norms further degrades performance.

Chapter 4

Discussion

In the results sections, we presented extensive charts, data, and convergence metrics, offering a **data-driven analysis** of the limitations in multi-agent reinforcement learning and potential solutions. However, our discussion chapter aims to provide a **broader perspective** on our research journey, encompassing pivotal transitions and stages, along with novel insights and challenges encountered. We wrap up each section with concise **recommendations for constructing robust systems** and cautionary notes on **practices to avoid**.

- section 4.1 examines the **adaptability of RL algorithms** across tasks and their effectiveness in multi-agent settings.
- section 4.2 analyzes the **inability of dictatorial approaches to converge** in complex, dynamic environments where global views may not suffice.
- section 4.3 investigates the **transition to a multi-scale environment** overview and its implications on individuality and agent performance.
- section 4.4 discusses the critical **role of initialization and regularization** in deep RL, highlighting their impact on convergence and avoidance of local optima.
- section 4.5 explores the **trade-offs of model complexity and training time**, particularly in the context of Lux AI competitions where submissions often prioritize deep neural nets without sufficient consideration of problem-solving approaches.
- section 4.6 analyzes the **transferability of learned behaviors** in MARL systems, focusing on the dynamics of alignment and reward shaping and their implications on agent performance.

4.1 Diverse Domains, Diverse Algorithms

We argue that a **one-size-fits-all algorithm does not exist** in the context of reinforcement learning, particularly in multi-agent environments (C. Liu & G. Liu 2024). Typically, when a new technique or algorithm emerges in reinforcement learning, it is tested within a specific task or a set of similar environments. These are designed for single-agent settings, simplifying the problem space by avoiding dynamic entity cardinalities, credit assignment burdens, and trajectory separation issues. Our benchmark study of novel RL algorithms highlights potential issues in their applicability to fuzzy tasks and complex environments, such as Lux AI (section 1.4).

Augmented Random Search (subsection 2.1.13) tends to underperform as the number of learnable parameters in a neural network increases. In environments like Lux, which require extensive state space representations and larger networks to learn input-output mappings, the **usability of derivation-free methods degrades**. Neuroevolution is still an evolving subfield of AI, making architectural searches inefficient for our purposes; employing domain knowledge and prior insights proves more effective.

In our benchmark study, TRPO (subsection 2.1.6) and A2C (subsection 2.1.7) delivered similar results, illustrating the necessity for PPO's development. Given the sparsity of positive actions in such environments, **TRPO and A2C are not regulative enough** with the magnitude of policy updates. PPO addresses this with an advanced clipping mechanism that trims extensive updates at a hyperparameter-defined threshold, preventing detrimental changes and maintaining the direction of beneficial updates.

The comparison between M-PPO and standard PPO clearly shows that **invalid action masking significantly enhances the convergence speed of the algorithms**. This improvement is consistent across most environments tested in the literature, as it introduces domain-specific knowledge into the learning model (S. Huang & Ontañón 2022). To illustrate, consider teaching a child to bicycle down an alleyway with the promise of a reward at the end but failing to mention the need to avoid obstacles that could cause a fall and potentially prolong the journey. With invalid action masking, it's like explicitly instructing that colliding with obstacles is prohibited because it never leads to positive rewards.

Recurrent PPO (subsection 2.1.9) is particularly effective in scenarios where interdependencies between environmental steps are crucial for decision-making, such as in Partially Observable Markov Decision Processes (POMDPs), where agents sense only a local area of the environment. This perceptional handicap generates a trajectory that significantly enhances an agent's decision-making capabilities. However, in our study, the impact of interdependencies between steps is minimized due to the fully observable nature of the environment. Given that resources, units, and factories are always visible, a single state of the environment provides sufficient information for making decisions. In our case, the recurrent nature of LSTMs and their reliance on sequence-based input can lead to noise from past observations, which results in a worse performance than approaches that do not incorporate historical data. **R-PPO**, however, **could be utilized for generating action queues**, which we did not use in our study. A recurrent policy could plan a sequence of action autoregressively, reducing the frequency of action queue updates and lowering the power costs of the units. This specialized application was beyond the scope of our current benchmark study but remains a promising area for future research.

Regarding DQNs (subsection 2.1.11; subsection 2.1.12), it is evident that learning the full distribution over returns provides significantly better results, especially in environments where future rewards are sparse and estimating a mean could distort the true value of an action in a given state. Utilizing a quantization method on a fixed uniform distribution can better capture the actual value and future impact of choosing an action a in state s. However, this approach comes with substantial computational costs, which limit our ability to conduct further tests. QR-DQN remains a promising candidate for our proposed hybrid method in the future.

In conclusion, PPO provides an ideal parallelizable framework due to its fixed rollout sizes and effective trajectory separation. Our hybrid approach aims to extend these capabilities to fully adapt it for multi-agent domains, leading us to designate it as **MA-PPO**.

4.2 The Failure of Dictatorship in Multi-Agent Societies

In a dictatorship, one person or a small group holds absolute power without constitutional limitations (Britannica, T. Editors of Encyclopaedia n.d.). In the realm of multi-agent systems, this concept translates to a social dynamic where one entity dictates rules and decisions without input from others. As discussed in section 2.3, we referred to this dynamic as **direct input**, where individual entity actions impact the model's behavior at either an individual or group level through their trajectories. However, in our monolithic solution, trajectory separation is absent, and entities are perceived as a singular entity rather than individual contributors. This phenomenon, we termed **Multi-Agent Dictatorship**, characterizes this approach.

Looking at the results in section 3.2, a significant degradation in model performance is evident, with the model failing to grasp the necessary behaviors in the environment. Even in instances where it managed to accumulate a larger sum of rewards due to exploratory ice collection or transfer to a factory, it struggled to reinforce this behavior. One might question why this is the case. When the model observes a global increase in ice at one of the factories and receives a reward, it **should theoretically converge** towards this behavior. However, **in the monolithic approach, this is not the case**. Our environment is sparse, with the exploration space scaling exponentially as the agent progresses through an episode. Despite running 32 environments with an episode length of 256 steps each, totaling 8k step samples (subsection 2.3.14), and an average of around 20 units on the map per step, **beneficial actions are exceedingly rare**. Even when an agent does discover a highly beneficial move, it typically involves just one unit on the map, while the rest are engaged in non-beneficial actions. Paradoxically, this can lead to a **reinforcement of suboptimal behavior**, as the single unit's beneficial action contributes to the global reward of the system, reinforcing the actions of all units, including those that did not contribute positively to the outcome.

Another bottleneck arises from the single trajectory approach (section 2.2), where the environment's trajectory is one long sequence. In this setup, the death of a unit does not impact the trajectory length, as termination only occurs when all factories of one player are destroyed. However, the death of a unit is highly detrimental because any subsequent beneficial actions taken by other units can lead to **reinforcement of the deceased unit's past actions**. This creates another paradoxical situation where the monolithic approach cannot adequately explain the reinforcement dynamics, especially in cases involving the creation or destruction of units. This phenomenon is exacerbated by how the total accumulated reward throughout a rollout is calculated. As a result, we observe peculiar interactions and reinforcement of actions, even when units die or are created. In some cases, a newly created unit can **inherit reinforcement from a past action** executed by a different unit, despite the newly created unit not yet contributing positively to the environment.

Additional bottlenecks arise from the single value head (subsection 2.3.12), which contributes to the global perspective on the environment by calculating a single value over all states. This approach becomes **highly sparse**, considering that only a small percentage of the total map is occupied by entities, typically ranging from 1 to 2 percent (20 units on average + 4 factory locations). Similarly, using a single critic is suboptimal because we aim to have multiple agents performing various tasks in the environment. Moreover, relying solely on a single actor head is **insufficient to capture the complexity** of multi-agent interactions. Furthermore, the feature extractor of the network (subsection 2.3.11), particularly a U-net architecture, **requires significantly more data** than what is available in our 200k-step training scenario to learn the class

segmentation of units on the map (e.g., distinguishing between diggers and rechargers). The limited data, coupled with incorrect reinforcements and weight updates by the PPO algorithm, can degrade the quality of updates and lead to **divergence**.

While a dictatorial PPO approach may seem easier to implement, it ultimately proves **ineffective** in multi-agent systems due to its inability to account for individuality and the nuanced interactions among agents.

4.3 Beyond One Mind

What is an agent? This one question haunted us throughout our work. When looking at existing solutions for the Lux AI environment, virtually all approaches relied on a single idea. Create a single-brain model that outputs an action for every entity on the board (Chen *et al.* 2023, Limburg 2023). This did not look like a truly multi-agent solution to us since there was only one real agent with a very complex action space. We wondered why everyone gravitated towards this idea instead of a more traditional multi-agent control approach. We think this architecture became popular for a couple of reasons. First, it solves, or at least mitigates the credit assignment problem since there is effectively a single agent. Second, the 2D grid map can be easily interpreted by a convolutional network, which also can output actions in the same shape. Third, since the number of entities constantly changes, letting one agent manage the whole team simultaneously is much simpler.

Upon closer examination, we realized, there is more to this architecture than a monolithic, single-brain model. If we keep the board's shape at each layer, we can interpret this network as predicting the entities' actions from their local environment's observation through convolutional filters. This is important since if we used some bottleneck to represent the environment's global state, the network would have difficulty capturing and using every relevant information. However, if we use local information through convolutions and at no point change the size of the board, the model can focus on the specific entity's localized observations. Localized observations result in needing to understand much less about the whole game to output an action for a single entity. Using residual connections helps even more since the features do not need to be entirely remapped at every layer, allowing the network to capture important information around the entity. This is why we started using the term "hybrid architecture" to differentiate it from simple monolithic architecture. We think this change matters a lot and should be studied more.

Even though the hybrid architecture can, in theory, allow a smaller model to solve the task by using localized observations, we still do not consider this a multi-agent solution. There is a single agent, the team itself, who can take an action with every single active entity. For this composite action, it receives a single reward value. The problem is that there is no way to tell which particular action resulted in that reward, so all of it would get reinforced or punished simultaneously. We solved this with our technique called **trajectory separation** (subsection 2.4.5). With it, we can treat every entity as a separate trajectory and massively improve the performance of our model. With this implementation, we can call entities agents since each entity can work separately from the other, using local observations and getting a unique reward value for their actions while keeping the optimization benefits of having a centralized model.

4.4 Initialization is All You Need

Reproducibility has always been a problem in scientific research. In a 2016 survey, more than 1,500 scientists were questioned about the reproducibility of their research and the research of others. 70% of them could not replicate other people's results, and 50% could not even replicate their own findings (Karbasi *et al.* 2023, M. Baker 2016). Unfortunately, this **reproducibility crisis** is relevant in machine learning too, so much so that conferences have established so-called **reproducibility challenges** (with Code 2022).

When it comes to reinforcement learning, reproducibility is especially hard. Henderson *et al.* 2019 showed that environment stochasticity and randomness in the learning process, such as weight initialization, matters so much that even averaging together multiple runs with different random seeds can lead to misleading results. We also experienced a significant performance fluctuation during our initial experiments purely because of different random seeding. Having a differently initialized neural network meant that the initial preferred policies of the agents were different, which in turn led to different observed states in the environment, resulting in training trajectories that were very different from each other.

At first, we had trouble reproducing our results, even with the same seeds. We learned that many things depend on the state of the random generator. In our environments, we had to seed the random generator for Pytorch since that is the machine learning framework we used for training and Numpy since the Lux AI environment depended on it. Python's random generator also had to be seeded since simple things like the hash function are stochastic to prevent certain attacks (Pieters 2014). We also learned that the convolutional layer's output in Pytorch with CUDA is not entirely deterministic since multiple convolution algorithms are benchmarked to find the fastest (PyTorch Contributors n.d.). This feature can be turned off, and doing so solved our issues. However, while turning off benchmarking and proper seeding solves our reproducibility issues, it is worth noting that convolutional layers used on GPUs with a multi-threaded setting will not result in the same outputs because addition on floating point numbers is not associative (David Goldberg 1991), resulting in very slight differences.

During our architecture search, we noticed a strange behavior. Modifying the order of layer definitions changed how quickly the model's training converged. In one such case, the initialization was so optimal that the entities learned to keep factories alive until the end of the game very early, while changing any one line in the model's structure caused the model not to learn anything. Changing the order of layer definitions even by one line meant that the random generator was at entirely different states when initializing the same parameters. It turns out that it is pretty standard to initialize the networks with orthogonal initialization, with the policy layers' weights scaled down in order to represent better a uniform distribution (S. Huang, Dossa, Raffin, *et al.* 2022), allowing each action to be explored with the same frequency. Some research suggests scaling down the output layers even more by a factor of 100. In Andrychowicz *et al.* 2020, they found that doing so increases performance by 66% in one benchmark environment, which corresponds to what we experienced. A badly-initialized model means that the agent must first unlearn its inherent bad behavior. That is, if it can even experience beneficial actions. After scaling down the policy layers, the massive fluctuations in performance disappeared. However, there was still a noticeable difference between seeds. We looked at the PPO losses and saw that, in some cases, the value loss exploded. At that time, we used a shared embedding network for the actor and critic heads, and an exploding value loss meant that the value loss's gradient completely dominated the embedding network's weight updates, causing policy collapse. Scaling

down the value network by the same factor solved this exploding loss issue. Having said that, the difference between seeds only got under control after we scaled down every single weight in the hidden layers. We are not sure why these tiny weights were necessary, mainly because our network already contains many regularization elements with batch norm (subsection 2.3.8), spectral norm (subsection 2.3.9) and residual connections (subsection 2.3.5).

4.5 Sometimes Less is More

Looking at the submissions for the best performing reinforcement learning submissions in the Lux AI competition, we noticed how long it took to train them. The best submission was trained on 65 million envrionment steps (Limburg 2023), just to learn rudemantary skills, for example collecting resources. The baseline solution that was provided by the organizers learned how to keep factories alive until the end of the game only after 1.4 million envrionment steps, and 2 days of constant training. Since we wanted to search through a lot of architectures, we knew that these training times would not work for us, because of both time and resource contraints. Therefore, we wanted to speed up the convergence of the models to as fast as possible.

We experimented with small, fully connected neural networks at first, with a lot of hand-engineered features. For example, the vector pointing towards the closest ice or factory. We could not achieve much success with this method, so we decided to stop using such features and, like many others, switch to a pixel-to-pixel (Chen *et al.* 2023) convolutional arhictecture. We built the model from the bottom up, trying to be as small as possible, while still being able to learn our designated goals. We also tried changing the number of rollout steps, minibatch-size and the number of parallel environments. We found out that by reducing the maximum episode length by a factor of 4, the model could still learn to keep the factories alive until the original episode length, while being able to train much faster due to now having more environments running in parallel.

The largest boost in performance came from the introduction of our technique called **trajectory separation** (subsection 2.4.5). After separating the rewards and the network outputs to entity level, we saw a huge boost in performance (subsection 3.3.1). We did experiments comparing this approach to global trajectories and found that without this feature, the model can still converge to solving the problem, but much slower. This naturally makes us wonder what performance could be achieved with our approach, if we trained for the same time and step size as the leading submissions. In our opinion, working within constraints can be beneficial, in order to find new solutions.

4.6 Journey from Singular to Spectrum

Transferring domain knowledge from specific domains has been shown to reduce sample efficiency in Reinforcement Learning algorithms (J. Huang & N. He 2023). In our case, starting with the simple tasks of ice gathering and water generation for factories covers only a fraction of the potential activities in the Lux environment, which also includes ore collection, rubble destruction, enemy unit destruction, and resource sabotage. This approach to learning multiple dissimilar tasks in the same environment is known as **Multi-Task RL (MT-RL)** (Vithayathil Varghese & Mahmoud 2020) while teaching an agent to apply learned knowledge from one task to another is termed **Transfer RL** (Lazaric 2012; Taylor & Stone 2009; Z. Zhu *et al.* 2023).

Both MT-RL and Transfer RL come with challenges. MT-RL often requires a weighting framework because not all tasks are equally important, and Transfer RL can struggle when the learned task is substantially different from the new target task (D'Eramo *et al.* 2024; Brunskill & L. Li 2013). However, even tasks as distinct as enemy unit sabotage and ice collection share some similarities, such as basic movement patterns in the environment.

Implementing these methods involves detailed processes like proper checkpointing, resetting annealing hyperparameters, and updating reward signals. One approach could involve learning a source and a target network concurrently, with the source network acting as a regularizer for the target network (J. Huang & N. He 2023). This process allows the target network to converge on a new task while maintaining relevance to the source domain. Due to its complexity and time demands, a Multi-Task learning framework remains a prospective area for future exploration in our research.

To expand from focusing on a single task to addressing a broader spectrum of activities, we retained the same configuration—using the same state representation, network architecture, and credit assignment system—but we also **introduced rewards for new behaviors** (subsection 3.3.4). For instance, we began rewarding units for clearing rubble next to factories and for factories watering lichen around themselves. This approach effectively aligns the system to new goals within the environment but requires learning from scratch, which is a significant drawback.

This strategy could be further extended to include additional tasks, each requiring a new training phase from scratch. Alternatively, one could adjust the alignment of agents through invalid action masking; however, **we recommend employing established retraining or continual training methods** from the literature, applied appropriately.

Chapter 5

Conclusion and Future Work

Multi-agent research plays a crucial role in advancing the pursuit of **Artificial General Intelligence** by exploring emergent intelligence in environments that abstract real-world scenarios. These settings, characterized by dynamic agent interactions, diverse interests, and fuzzy goals, **mirror the complexity of real-life situations**. Multi-agent competitions, like Lux, serve as invaluable platforms for developers, offering open-source frameworks to simulate diverse environments. These environments enable researchers to explore, test, and refine their ideas, fostering a **deeper understanding of communication, cooperation, interaction, and common goal achievement among agents**.

In this work, we introduced a **trajectory separation method** for improving on the performance of a popular policy gradient method, PPO. We also **provided insights** into creating suitable non-linear approximator architectures (NNs) for policy and value approximation. Additionally, we proposed **guidelines for selecting novel RL algorithms** suitable for MARL settings, focusing on the Lux AI environment.

Regarding the future, we aim to explore the applicability of **R-PPO in action queue generation** and **QR-DQN as another baseline** for MARL settings. Furthermore, we plan to experiment with **continual or multi-phase training** strategies to leverage additional possible strategies within the Lux AI environment. We aim to expand and test our hybrid model on **various environments** to assess its generalizability.

We also intend to explore additional more advanced techniques beyond the scope of this research. One such approach involves implementing a class system using trajectory splitting and per-class reward shaping. This method will allow us to categorize agents into different types, such as diggers or saboteurs, enhancing the versatility of our models. Furthermore, we plan to integrate an internal communication protocol for agent cooperation, leveraging attention mechanisms (Vaswani *et al.* 2023). Additionally, we aim to improve our models by implementing a transformer decoder-based action-queue generator, further extending their capabilities.

We are enthusiastic about the future of this project and expect significant potential in extending PPO to multi-agent settings, **officially coining the term MA-PPO** (Multi-Agent Proximal Policy Optimization).

Acknowledgements

This work was supported by the **Institute for Business Cooperation Scholarship** under grants **#E111SIVE**. We extend our gratitude to our supervisor, **László Gulyás**, for his extensive efforts, patience, and invaluable guidance, which were crucial throughout this project. We also thank **Eötvös Loránd University and the Department of Artificial Intelligence** for providing the resources and environment essential for this project's development.

Appendix A

Hyperparameters

A.1 Single Unit Testbench

Hyperparameter	ARS	A2C	DQN	PPO	QR-DQN	R-PPO	TRPO	MPPO
policy	MLP	MLP	MLP	MLP	MLP	MLP	MLP	MLP
learning_rate	2e-2	7e-4	3e-4	3e-4	7e-4	3e-4	1e-4	3e-4
n_steps	-	50	-	50	-	50	50	50
batch_size	-	800	800	800	800	800	800	800
n_epochs	-	-	-	10	-	10	-	10
gamma	-	0.99	0.99	0.99	0.99	0.99	0.99	0.99
gae_lambda	-	0.95	-	0.95	-	0.95	0.95	0.95
tau	-	-	0.95	-	0.95	-	-	-
ent_coef	-	0.0	-	0.0	-	0.0	-	0.0
vf_coef	-	0.5	-	0.5	-	0.5	-	0.5
exp_fraction	-	-	0.1	-	0.01	-	-	-
exp_initial_eps	-	-	1.0	-	1.0	-	-	-
exp_final_eps	-	-	0.05	-	0.1	-	-	-
clip_range	-	-	-	0.2	-	0.2	-	0.2
target_kl	-	-	-	0.05	-	0.05	0.05	0.05
net_params	128 128	128 128	128 128	128 128	128 128	128 128	128 128	128 128

Table A.1: The table displays key hyperparameters utilized in the single unit benchmark study. It's important to note that for the R-PPO algorithm, the value 128 denotes the hidden size of the LSTM layer, whereas for the other algorithms, 128 refers to the size of the linear projection layer.

A.2 Monolithic Approach

Hyperparameter	Monolithic Training	Comparison With Other Methods
policy	MLP	MLP
number of trainable parameters	X	X
shared actor and value embeddings	false	false
algorithm	M-PPO	M-PPO
optimizer	Adam	Adam
weight decay	false	false
learning rate	2e-4	2e-4
max episode length	256	1000
parallel environments	32	8
rollout steps	8,192	8,192
mini-batch size	1024	1024
epochs per train cycle	10	10
train until N steps	204,800	204,800
gamma	0.99	0.99
GAE lambda	0.95	0.95
entropy coefficient	1e-3	1e-3
value coefficient	0.5	0.5
gradient normalization	true	true
maximum gradient norm	0.5	0.5
clip range	0.2	0.2
clip value loss	true	true
target KL	none	none
runs per test	3	3
evaluation environments	12	12
evaluation after every N train cycles	1	1

Table A.2: Table containing all key hyperparameters utilized in the monolithic approach tests. The **Monolithic Training** hyperparameters were used in section 2.3, section 3.2, while the "Comparison With Other Methods" hyperparameters were only used in subsection 3.3.4.

A.3 Hybrid Approach

Hyperparameter	Trajectory Separation Test	Comparison With Other Methods
policy	MLP	MLP
number of trainable parameters	208,457	209,033
shared actor and value embeddings	false	false
algorithm	M-PPO	M-PPO
optimizer	Adam	Adam
weight decay	false	false
learning rate	2e-4	2e-4
max episode length	256	1000
parallel environments	16	4
rollout steps	4,096	4,096
mini-batch size	512	512
epochs per train cycle	10	10
train until N steps	102,400	102,400
gamma	0.99	0.99
GAE lambda	0.95	0.95
entropy coefficient	1e-3	1e-3
value coefficient	0.5	0.5
gradient normalization	true	true
maximum gradient norm	0.5	0.5
clip range	0.2	0.2
clip value loss	true	true
target KL	none	none
runs per test	3	3
evaluation environments	12	12
evaluation after every N train cycles	1	1

Table A.3: Table containing all key hyperparameters utilized in the hybrid approach tests. The **Trajectory Separation Test** hyperparameters were used in subsection 3.3.1, subsection 3.3.2, subsection 3.3.3 and subsection 3.3.5, while the "Comparison With Other Methods" hyperparameters were only used in subsection 3.3.4.

TAMAS:
- Monolithic Results

GERI:
- Credit assignment in background

Appendix B

Tools Used and Other Implementation Details

Our initial approach started with integrating the Lux Environment (Tao, Pan, *et al.* 2023a) directly into an easy-to-manage development environment through containerization. For this process, we went with the most popular option, **Docker** (Merkel 2014). For our initial setup, we selected two distinct Ubuntu configurations: a CPU-only version for debugging (**ubuntu:latest**) and another with NVIDIA CUDA support (**nvidia/cuda:12.2.0-base-ubuntu20.04**) for training purposes. However, the computational demands of our experiments increased to the point where it became completely unfeasible to conduct runs locally. Thus, we transitioned to cloud-based training, utilizing a **Google Colab's Pro** subscription (Bisong 2019) to train on GPUs with increased memory sizes. The rationale for containerization was to increase transparency and ensure compatibility across devices. Moreover, it aimed to facilitate future development and future integration with cloud services for training. This approach significantly reduced our research overhead associated with running experiments by utilizing **VSCode** (Microsoft 2015) and its DevContainer tool.

All code development was carried out on local machines, with code security managed through **Git** Source Control and hosted on **GitHub**. The Lux Environment was accessible in two forms: as a package installable via **pip** (Tao, Pan, *et al.* 2023b) or through its raw source code (Tao, Pan, *et al.* 2023c). To maintain consistency and minimize disruptions in our research, we opted to pin the version of the Lux Environment to the iteration current as of the last competition in March 2023 (Tao, Pan, *et al.* 2023a). Therefore, instead of using the pip package, we downloaded the source code to make it always available offline.

The engine was mandatory for both the development and the evaluation of actions to determine their correctness. It managed the stepping of the environment, making sure each step produced a stable state and provided consistent updates. Moreover, the engine was responsible for sending the fully visible environmental state to our client solution in JSON format through the default output stream (stdout). Because information exchange exhausted the two most used data streams on our system, debugging only became possible with the utilization of logger packages. As the engine was developed exclusively in **Python** (Python Software Foundation 2019), we naturally also chose Python for our implementation to maintain compatibility and evade serialization issues.

To ensure transparency and reproducibility in our research, we have implemented automated testing using **PyTest** and static code analysis. Our Docker images are uploaded to **Docker Hub**

and **GitHub Docker Registry**, and all documentation is accessible through the source code. The Docker images can be accessed through: `https://hub.docker.com/r/ranuon98/luxai_gpu`.

We have also used AI tools like **GitHub CoPilot** (GitHub 2021) to speed up development, **Grammarly** (Grammarly 2009) for grammar checking, and **ChatGPT** (OpenAI 2023) to help with phrasing.

As the starting point for the development of our **hybrid model** (section 2.4), we have branched out of a Github repository called **Luxai-s2-Baseline** by **RoboEden** (Kai Yang 2023), which we found recommended in the Github repository of the **Lux AI Challenge season 2** competition (Tao, Q. Li, *et al.* 2023).

The code, training visualizations, evaluation results, issues, and a project wiki are publicly available at: `https://github.com/MagmaMultiAgent`

Bibliography

1. Gilbert, T. K. *et al. Reward Reports for Reinforcement Learning* 2023. arXiv: 2204.10817 [cs.LG].
2. Priyadarshi, A. *et al. Reinforcement Learning Market Size, Share, Competitive Landscape and Trend Analysis Report* https://www.alliedmarketresearch.com/reinforcement-learning-market-A229407. [Online; accessed 2024-03-26]. 2023.
3. Kirk, H. R. *et al. Personalisation within bounds: A risk taxonomy and policy framework for the alignment of large language models with personalised feedback* 2023. arXiv: 2303.05453 [cs.CL].
4. Lowe, R. *et al. Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments* 2020. arXiv: 1706.02275 [cs.LG].
5. Foerster, J. N. *et al. Learning with Opponent-Learning Awareness* 2018. arXiv: 1709.04326 [cs.AI].
6. OpenAI *et al. Dota 2 with Large Scale Deep Reinforcement Learning* 2019. arXiv: 1912.06680 [cs.LG].
7. Baker, B. *et al. Emergent Tool Use From Multi-Agent Autocurricula* 2020. arXiv: 1909.07528 [cs.LG].
8. Nash, J. F. Equilibrium points in n-person games. *Proceedings of the National Academy of Sciences* **36,** 48–49. doi:10.1073/pnas.36.1.48. eprint: https://www.pnas.org/doi/pdf/10.1073/pnas.36.1.48. https://www.pnas.org/doi/abs/10.1073/pnas.36.1.48 (1950).
9. Nash, J. Non-cooperative Games. *Annals of Mathematics* **54,** 286–295 (1951).
10. Neumann, J. v. Zur Theorie der Gesellschaftsspiele. *Mathematische Annalen* **100,** 295–320. http://eudml.org/doc/159291 (1928).
11. Myerson, R. B. *Game theory - Analysis of Conflict.* I–XIII, 1–568. ISBN: 978-0-674-34116-6 (Harvard University Press, 1997).
12. Schulman, L. J. & Vazirani, U. V. The duality gap for two-team zero-sum games. *Games and Economic Behavior* **115,** 336–345. ISSN: 0899-8256. doi:https://doi.org/10.1016/j.geb.2019.03.011. https://www.sciencedirect.com/science/article/pii/S0899825619300466 (2019).
13. Li, C. *et al.* Verification and Design of Zero-Sum Potential Games. *IFAC-PapersOnLine* **53.** 21st IFAC World Congress, 16932–16937. ISSN: 2405-8963. doi:https://doi.org/10.1016/j.ifacol.2020.12.1236. https://www.sciencedirect.com/science/article/pii/S2405896320316360 (2020).
14. Suarez, J. *et al. Neural MMO: A Massively Multiagent Game Environment for Training and Evaluating Intelligent Agents* 2019. arXiv: 1903.00784 [cs.MA].

15. Agapiou, J. P. *et al. Melting Pot 2.0* 2023. arXiv: 2211.13746 [cs.MA].

16. Tao, S., Pan, I., *et al. Lux AI Season 2* 2023. https://kaggle.com/competitions/lux-ai-season-2.

17. Wong, A. *et al. Deep Multiagent Reinforcement Learning: Challenges and Directions* 2022. arXiv: 2106.15691 [cs.LG].

18. Chen, H. *et al. Emergent collective intelligence from massive-agent cooperation and competition* 2023. arXiv: 2301.01609 [cs.AI].

19. Ye, J. *et al. Towards Global Optimality in Cooperative MARL with the Transformation And Distillation Framework* 2023. arXiv: 2207.11143 [cs.MA].

20. Leroy, P. *et al. QVMix and QVMix-Max: Extending the Deep Quality-Value Family of Algorithms to Cooperative Multi-Agent Reinforcement Learning* 2020. arXiv: 2012.12062 [cs.LG].

21. Piccoli, B. *Control of multi-agent systems: results, open problems, and applications* 2023. arXiv: 2302.12308 [math.OC].

22. Kai Yang, S. T. *LuxAI-s2-Baseline* https://github.com/RoboEden/Luxai-s2-Baseline/tree/main. [Online; accessed 2023-11-15]. 2023.

23. Du, H. *et al. A Survey on Context-Aware Multi-Agent Systems: Techniques, Challenges and Future Directions* 2024. arXiv: 2402.01968 [cs.MA].

24. Aguayo-Canela, F. J. *et al.* Enriched multi-agent middleware for building rule-based distributed security solutions for IoT environments. *The Journal of Supercomputing* **77,** 13046–13068. ISSN: 1573-0484. doi:10.1007/s11227-021-03797-2. http://dx.doi.org/10.1007/s11227-021-03797-2 (Apr. 2021).

25. Anderson, R. *Logic-based Lux Strategy* https://www.kaggle.com/competitions/lux-ai-season-2/discussion/407982. [Online; accessed 2024-03-26]. 2023.

26. Tigga, T. *Yet Another Logic-Bot* https://www.kaggle.com/competitions/lux-ai-season-2/discussion/405476. [Online; accessed 2024-03-26]. 2023.

27. Kostuch, P. *Pure Logic Approach* https://www.kaggle.com/competitions/lux-ai-season-2/discussion/405245. [Online; accessed 2024-03-26]. 2023.

28. Politowski, C. *et al.* A large scale empirical study of the impact of Spaghetti Code and Blob anti-patterns on program comprehension. *Information and Software Technology* **122,** 106278. ISSN: 0950-5849. doi:10.1016/j.infsof.2020.106278. http://dx.doi.org/10.1016/j.infsof.2020.106278 (June 2020).

29. Hussein, A. *et al.* Imitation Learning: A Survey of Learning Methods. *ACM Comput. Surv.* **50.** ISSN: 0360-0300. doi:10.1145/3054912. https://doi.org/10.1145/3054912 (Apr. 2017).

30. Goecks, V. G. *et al. Combining Learning from Human Feedback and Knowledge Engineering to Solve Hierarchical Tasks in Minecraft* 2022. arXiv: 2112.03482 [cs.LG].

31. Garg, D. *et al. IQ-Learn: Inverse soft-Q Learning for Imitation* 2022. arXiv: 2106.12142 [cs.LG].

32. Nagradov, E. *IL-based Solution* https://www.kaggle.com/competitions/lux-ai-season-2/discussion/461697. [Online; accessed 2024-03-26]. 2023.

33. Limburg, F. *FLG's Approach* https://www.kaggle.com/competitions/lux-ai-season-2/discussion/406702. [Online; accessed 2024-03-26]. 2023.

34. Sevilla, J. *et al. Compute Trends Across Three Eras of Machine Learning* in *2022 International Joint Conference on Neural Networks (IJCNN)* (IEEE, July 2022). doi:10.1109/ijcnn55064.2022.9891914. http://dx.doi.org/10.1109/IJCNN55064.2022.9891914.

35. Epoch. *Parameter, Compute and Data Trends in Machine Learning* Accessed: 2024-03-31. 2024. https://epochai.org/data/epochdb/visualization.

36. Eberhardinger, M. *et al. Learning of Generalizable and Interpretable Knowledge in Grid-Based Reinforcement Learning Environments* 2023. arXiv: 2309.03651 [cs.AI].

37. Khorasgani, H. *et al. K-nearest Multi-agent Deep Reinforcement Learning for Collaborative Tasks with a Variable Number of Agents* 2022. arXiv: 2201.07092 [cs.LG].

38. Min, D. J. *et al. Dynamic Reward Adjustment in Multi-Reward Reinforcement Learning for Counselor Reflection Generation* 2024. arXiv: 2403.13578 [cs.CL].

39. Tan, J. *et al.* Multi-Objective Optimization Using Adaptive Distributed Reinforcement Learning. *IEEE Transactions on Intelligent Transportation Systems,* 1–13. ISSN: 1558-0016. doi:10.1109/tits.2024.3378007. http://dx.doi.org/10.1109/TITS.2024.3378007 (2024).

40. Zheng, X. & Yu, C. *Multi-Agent Reinforcement Learning with a Hierarchy of Reward Machines* 2024. arXiv: 2403.07005 [cs.AI].

41. Lee, D. *et al.* Optimization for Reinforcement Learning: From a single agent to cooperative agents. *IEEE Signal Processing Magazine* **37,** 123–135. ISSN: 1558-0792. doi:10.1109/msp.2020.2976000. http://dx.doi.org/10.1109/MSP.2020.2976000 (May 2020).

42. Sutton, R. S. & Barto, A. G. *Reinforcement Learning: An Introduction* Second. http://incompleteideas.net/book/the-book-2nd.html (The MIT Press, 2018).

43. Mnih, V., Kavukcuoglu, K., *et al. Playing Atari with Deep Reinforcement Learning* 2013. arXiv: 1312.5602 [cs.LG].

44. Silver, D. *et al.* Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature* **529,** 484–489. ISSN: 0028-0836. doi:10.1038/nature16961 (Jan. 2016).

45. Achiam, J. Spinning Up in Deep Reinforcement Learning (2018).

46. Tang, Y. & Agrawal, S. *Discretizing Continuous Action Space for On-Policy Optimization* 2020. arXiv: 1901.10500 [cs.LG].

47. AlMahamid, F. & Grolinger, K. *Reinforcement Learning Algorithms: An Overview and Classification* in *2021 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)* (IEEE, Sept. 2021). doi:10.1109/ccece53047.2021.9569056. http://dx.doi.org/10.1109/CCECE53047.2021.9569056.

48. Zitzler, E. in *Handbook of Natural Computing* (eds Rozenberg, G. *et al.*) 871–904 (Springer Berlin Heidelberg, Berlin, Heidelberg, 2012). ISBN: 978-3-540-92910-9. doi:10.1007/978-3-540-92910-9_28. https://doi.org/10.1007/978-3-540-92910-9_28.

49. Foerster, J. *et al. Counterfactual Multi-Agent Policy Gradients* 2017. arXiv: 1705.08926 [cs.AI].

50. Lehmann, M. *The Definitive Guide to Policy Gradients in Deep Reinforcement Learning: Theory, Algorithms and Implementations* 2024. arXiv: 2401.13662 [cs.LG].

51. Athreya, K. B. & Lahiri, S. N. in *Measure Theory and Probability Theory* 237–285 (Springer New York, New York, NY, 2006). ISBN: 978-0-387-35434-7. doi:10.1007/978-0-387-35434-7_9. https://doi.org/10.1007/978-0-387-35434-7_9.

52. Della Vecchia, E. *et al. Illustrated review of convergence conditions of the value iteration algorithm and the rolling horizon procedure for average-cost MDPs* Research Report RR-7710 (LIRMM ; INRIA, Aug. 2011). https://inria.hal.science/inria-00617271.

53. Miller, T. *Introduction To Reinforcement Learning* https://gibberblot.github.io/rl-notes/index.html. [Online; accessed 2024-02-22]. 2023.

54. Szepesvári, C. *Algorithms for Reinforcement Learning* ISBN: 978-3-031-01551-9 (Morgan and Claypool Publishers, 2010).

55. Yanes Luis, S. *et al. A Sample-Efficiency Comparison Between Evolutionary Algorithms and Deep Reinforcement Learning for Path Planning in an Environmental Patrolling Mission* in (Apr. 2021).

56. Holland & H., J. *Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence* (U Michigan Press, 1975).

57. Galvan, E. & Mooney, P. Neuroevolution in Deep Neural Networks: Current Trends and Future Challenges. *IEEE Transactions on Artificial Intelligence* **2,** 476–493. ISSN: 2691-4581. doi:10.1109/tai.2021.3067574. http://dx.doi.org/10.1109/TAI.2021.3067574 (Dec. 2021).

58. Tao, S., Li, Q., *et al. Lux AI Challenge Season 2, NeurIPS Edition* in *Thirty-seventh Conference on Neural Information Processing Systems: Competition Track* (2023). https://github.com/Lux-AI-Challenge/Lux-Design-S2.

59. Brockman, G. *et al. OpenAI Gym* 2016. eprint: arXiv:1606.01540.

60. Terry, J. *et al.* Pettingzoo: Gym for multi-agent reinforcement learning. *Advances in Neural Information Processing Systems* **34,** 15032–15043 (2021).

61. Raffin, A. *et al.* Stable-Baselines3: Reliable Reinforcement Learning Implementations. *Journal of Machine Learning Research* **22,** 1–8. http://jmlr.org/papers/v22/20-1364.html (2021).

62. Moritz, P. *et al. Ray: A Distributed Framework for Emerging AI Applications* 2018. arXiv: 1712.05889 [cs.DC].

63. Abadi, M. *et al. TensorFlow: A system for large-scale machine learning* 2016. arXiv: 1605.08695 [cs.DC].

64. Andrychowicz, M. *et al. What Matters In On-Policy Reinforcement Learning? A Large-Scale Empirical Study* 2020. arXiv: 2006.05990 [cs.LG].

65. Van Hasselt, H. *et al. Deep Reinforcement Learning with Double Q-learning* 2015. arXiv: 1509.06461 [cs.LG].

66. Huang, S., Dossa, R. F. J., Raffin, A., *et al. The 37 Implementation Details of Proximal Policy Optimization* in *ICLR Blog Track* https://iclr-blog-track.github.io/2022/03/25/ppo-implementation-details/ (2022). https://iclr-blog-track.github.io/2022/03/25/ppo-implementation-details/.

67. Huang, S., Dossa, R. F. J., Ye, C., *et al.* CleanRL: High-quality Single-file Implementations of Deep Reinforcement Learning Algorithms. *Journal of Machine Learning Research* **23,** 1–18. http://jmlr.org/papers/v23/21-1342.html (2022).

68. Dohare, S. *et al. Overcoming Policy Collapse in Deep Reinforcement Learning* in *Sixteenth European Workshop on Reinforcement Learning* (2023). https://openreview.net/forum?id=m9Jfdz4ymO.

69. Schulman, J., Levine, S., *et al. Trust Region Policy Optimization* 2015. arXiv: 1502.05477 [cs.LG].

70. Knowles, D. *Lagrangian Duality* https://www-cs.stanford.edu/people/davidknowles/lagrangian_duality.pdf. [Online; accessed 2024-02-14]. 2010.

71. Wikipedia. *Conjugate gradient method — Wikipedia, The Free Encyclopedia* http://en.wikipedia.org/w/index.php?title=Conjugate%20gradient%20method&oldid=1218500371. [Online; accessed 13-April-2024]. 2024.

72. Mnih, V., Badia, A. P., *et al. Asynchronous Methods for Deep Reinforcement Learning* 2016. arXiv: 1602.01783 [cs.LG].

73. Wu, Y. *et al. Scalable trust-region method for deep reinforcement learning using Kronecker-factored approximation* 2017. arXiv: 1708.05144 [cs.LG].

74. Williams, R. & Peng, J. Function Optimization Using Connectionist Reinforcement Learning Algorithms. *Connection Science* **3,** 241–. doi:10.1080/09540099108946587 (Sept. 1991).

75. Huang, S., Kanervisto, A., *et al. A2C is a special case of PPO* 2022. arXiv: 2205.09123 [cs.LG].

76. Schulman, J., Wolski, F., *et al. Proximal Policy Optimization Algorithms* 2017. arXiv: 1707.06347 [cs.LG].

77. Ernestus, M. *et al. Issue 213: PPO implementation does not take care of too large KL Divergence* https://github.com/hill-a/stable-baselines/issues/213. [Online; accessed 2023-11-03]. 2019.

78. Engstrom, L. *et al. Implementation Matters in Deep RL: A Case Study on PPO and TRPO* in *International Conference on Learning Representations* (2020). https://openreview.net/forum?id=r1etN1rtPB.

79. Pleines, M. *et al. Generalization, Mayhems and Limits in Recurrent Proximal Policy Optimization* 2022. arXiv: 2205.11104 [cs.LG].

80. Andrychowicz, M. *et al. What Matters for On-Policy Deep Actor-Critic Methods? A Large-Scale Study* in *International Conference on Learning Representations* (2021). https://openreview.net/forum?id=nIAxjsniDzg.

81. Tallec, C. & Ollivier, Y. *Unbiasing Truncated Backpropagation Through Time* 2017. arXiv: 1705.08209 [cs.NE].

82. Kapturowski, S. *et al. Recurrent Experience Replay in Distributed Reinforcement Learning* in *International Conference on Learning Representations* (2019). https://openreview.net/forum?id=r1lyTjAqYX.

83. Gupta, K. *What matters in recurrent ppo for long episodic and continuing partially observable tasks* tech. rep. [Online; accessed 2024-04-11] (Mila Quebec AI Institute, 2021). https://kshitijkg.github.io/data/RecurrentPPO_Report.pdf.

84. Huang, S. & Ontañón, S. A Closer Look at Invalid Action Masking in Policy Gradient Algorithms. *The International FLAIRS Conference Proceedings* **35.** ISSN: 2334-0762. doi:10.32473/flairs.v35i.130584. http://dx.doi.org/10.32473/flairs.v35i.130584 (May 2022).

85. Vinyals, O. *et al. StarCraft II: A New Challenge for Reinforcement Learning* 2017. arXiv: 1708.04782 [cs.LG].

86. Johnson, M. *et al. The Malmo platform for artificial intelligence experimentation* in *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence* (AAAI Press, New York, New York, USA, 2016), 4246–4247. ISBN: 9781577357704.

87. Hou, Y. *et al.* Exploring the Use of Invalid Action Masking in Reinforcement Learning: A Comparative Study of On-Policy and Off-Policy Algorithms in Real-Time Strategy Games. *Applied Sciences* **13.** ISSN: 2076-3417. doi:10.3390/app13148283. https://www.mdpi.com/2076-3417/13/14/8283 (2023).

88. Dietterich, T. G. *Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition* 1999. arXiv: cs/9905014 [cs.LG].

89. Fedus, W. *et al. Revisiting Fundamentals of Experience Replay* 2020. arXiv: 2007.06700 [cs.LG].

90. Dhariwal, P. *et al. OpenAI Baselines* https://github.com/openai/baselines. 2017.

91. Wikipedia. *Huber loss — Wikipedia, The Free Encyclopedia* https://en.wikipedia.org/wiki/Huber_loss. [Online; accessed 15-April-2024]. 2024.

92. Dabney, W. *et al. Distributional Reinforcement Learning with Quantile Regression* 2017. arXiv: 1710.10044 [cs.AI].

93. Bellemare, M. G. *et al. A Distributional Perspective on Reinforcement Learning* 2017. arXiv: 1707.06887 [cs.LG].

94. Yang, D. *et al. Fully Parameterized Quantile Function for Distributional Reinforcement Learning* 2020. arXiv: 1911.02140 [cs.LG].

95. Mania, H. *et al. Simple random search provides a competitive approach to reinforcement learning* 2018. arXiv: 1803.07055 [cs.LG].

96. Henderson, P. *et al. Deep Reinforcement Learning that Matters* 2019. arXiv: 1709.06560 [cs.LG].

97. Terry, J. K. *et al.* SuperSuit: Simple Microwrappers for Reinforcement Learning Environments. *arXiv preprint arXiv:2008.08932* (2020).

98. Eysenbach, B. *et al. A Connection between One-Step Regularization and Critic Regularization in Reinforcement Learning* 2023. arXiv: 2307.12968 [cs.LG].

99. Gai, S. & Wang, D. *Single-Task Continual Offline Reinforcement Learning* 2024. arXiv: 2404.12639 [cs.LG].

100. Mysore, S. *et al. Multi-Critic Actor Learning: Teaching RL Policies to Act with Style* in *International Conference on Learning Representations* (2022). https://openreview.net/forum?id=rJvY_5OzoI.

101. Wikipedia. *Gaussian filter — Wikipedia, The Free Encyclopedia* [Online; accessed 29-April-2024]. 2024. https://en.wikipedia.org/w/index.php?title=Gaussian_filter&oldid=1221267487.

102. Krizhevsky, A. *et al. ImageNet classification with deep convolutional neural networks* in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1* (Curran Associates Inc., Lake Tahoe, Nevada, 2012), 1097–1105.

103. Simonyan, K. & Zisserman, A. *Very Deep Convolutional Networks for Large-Scale Image Recognition* 2015. arXiv: 1409.1556 [cs.CV].

104. Szegedy, C. *et al. Going Deeper with Convolutions* 2014. arXiv: 1409.4842 [cs.CV].

105. He, K. *et al. Deep Residual Learning for Image Recognition* 2015. arXiv: 1512.03385 [cs.CV].

106. Xu, B. *et al. Empirical Evaluation of Rectified Activations in Convolutional Network* 2015. arXiv: 1505.00853 [cs.LG].

107. Agarap, A. F. *Deep Learning using Rectified Linear Units (ReLU)* 2019. arXiv: 1803.08375 [cs.NE].

108. Lu, L. Dying ReLU and Initialization: Theory and Numerical Examples. *Communications in Computational Physics* **28,** 1671–1706. ISSN: 1991-7120. doi:10.4208/cicp.oa-2020-0165. http://dx.doi.org/10.4208/cicp.OA-2020-0165 (June 2020).

109. Datta, L. *A Survey on Activation Functions and their relation with Xavier and He Normal Initialization* 2020. arXiv: 2004.06632 [cs.NE].

110. PyTorch Contributors. *PyTorch Documentation* https://pytorch.org/docs/stable/_modules/torch/nn/init.html#orthogonal_. 2024.

111. Hu, J. *et al. Squeeze-and-Excitation Networks* 2017. arXiv: 1709.01507 [cs.CV].

112. Yahya, A. Feature Selection for High Dimensional Data: An Evolutionary Filter Approach. *Journal of Computer Science* **7,** 800–820. doi:10.3844/jcssp.2011.800.820 (May 2011).

113. Tian, Y. & Zhang, Y. A comprehensive survey on regularization strategies in machine learning. *Information Fusion* **80,** 146–166. ISSN: 1566-2535. doi:https://doi.org/10.1016/j.inffus.2021.11.005. https://www.sciencedirect.com/science/article/pii/S156625352100230X (2022).

114. Ioffe, S. & Szegedy, C. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift* 2015. arXiv: 1502.03167 [cs.LG].

115. Rakitianskaia, A. & Engelbrecht, A. *Measuring Saturation in Neural Networks* in *2015 IEEE Symposium Series on Computational Intelligence* (2015), 1423–1430. doi:10.1109/SSCI.2015.202.

116. Goodfellow, I. *et al.* in. http://www.deeplearningbook.org. Chap. 8 (MIT Press, 2016).

117. Miyato, T. *et al. Spectral Normalization for Generative Adversarial Networks* 2018. arXiv: 1802.05957 [cs.LG].

118. Yoshida, Y. & Miyato, T. *Spectral Norm Regularization for Improving the Generalizability of Deep Learning* 2017. arXiv: 1705.10941 [stat.ML].

119. Wikipedia. *Wasserstein metric — Wikipedia, The Free Encyclopedia* http://en.wikipedia.org/w/index.php?title=Wasserstein%20metric&oldid=1218030467. [Online; accessed 25-April-2024]. 2024.

120. Wikipedia. *Lipschitz continuity — Wikipedia, The Free Encyclopedia* http://en.wikipedia.org/w/index.php?title=Lipschitz%20continuity&oldid=1220390774. [Online; accessed 25-April-2024]. 2024.

121. Ansel, J. *et al. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation* in (ACM, Apr. 2024). doi:10.1145/3620665.3640366. https://pytorch.org/assets/pytorch2-2.pdf.

122. Gogianu, F. *et al. Spectral Normalisation for Deep Reinforcement Learning: an Optimisation Perspective* 2021. arXiv: 2105.05246 [cs.LG].

123. Hu, W. *et al. Provable Benefit of Orthogonal Initialization in Optimizing Deep Linear Networks* 2020. arXiv: 2001.05992 [cs.LG].

124. Kumar, S. K. *On weight initialization in deep neural networks* 2017. arXiv: 1704.08863 [cs.LG].

125. He, K. *et al. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification* 2015. arXiv: 1502.01852 [cs.CV].

126. Wikipedia. *Orthogonal matrix — Wikipedia, The Free Encyclopedia* http://en.wikipedia.org/w/index.php?title=Orthogonal%20matrix&oldid=1202509034. [Online; accessed 25-April-2024]. 2024.

127. Glorot, X. & Bengio, Y. *Understanding the difficulty of training deep feedforward neural networks* in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics* (eds Teh, Y. W. & Titterington, M.) **9** (PMLR, Chia Laguna Resort, Sardinia, Italy, May 2010), 249–256. https://proceedings.mlr.press/v9/glorot10a.html.

128. He, K. *et al.* Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. *CoRR* **abs/1502.01852.** arXiv: 1502.01852. http://arxiv.org/abs/1502.01852 (2015).

129. Ronneberger, O. *et al. U-Net: Convolutional Networks for Biomedical Image Segmentation* 2015. arXiv: 1505.04597 [cs.CV].

130. Wu, D. *et al. Skip Connections Matter: On the Transferability of Adversarial Examples Generated with ResNets* 2020. arXiv: 2002.05990 [cs.LG].

131. Ehab, W. & Li, Y. *Performance Analysis of UNet and Variants for Medical Image Segmentation* 2023. arXiv: 2309.13013 [eess.IV].

132. Li, Z. *et al. DetNet: A Backbone network for Object Detection* 2018. arXiv: 1804.06215 [cs.CV].

133. Gao, S.-H. *et al.* Res2Net: A New Multi-Scale Backbone Architecture. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **43,** 652–662. ISSN: 1939-3539. doi:10.1109/tpami.2019.2938758. http://dx.doi.org/10.1109/TPAMI.2019.2938758 (Feb. 2021).

134. Li, P. *et al.* Multi-scale Bottleneck Residual Network for Retinal Vessel Segmentation. *Journal of Medical Systems* **47.** doi:10.1007/s10916-023-01992-7 (Sept. 2023).

135. Zhu, W. *Efficient Multiscale Multimodal Bottleneck Transformer for Audio-Video Classification* 2024. arXiv: 2401.04023 [cs.CV].

136. Bhojanapalli, S. *et al. Low-Rank Bottleneck in Multi-head Attention Models* 2020. arXiv: 2002.07028 [cs.LG].

137. Shelhamer, E. *et al. Fully Convolutional Networks for Semantic Segmentation* 2016. arXiv: 1605.06211 [cs.CV].

138. Lin, M. *et al. Network In Network* 2014. arXiv: 1312.4400 [cs.NE].

139. Gleave, A. *et al. Issue 181: How best to implement self-play/multiple agents in the same environment?* https://github.com/hill-a/stable-baselines/issues/181. [Online; accessed 2023-11-06]. 2019.

140. Han, L. *et al. Grid-Wise Control for Multi-Agent Reinforcement Learning in Video Game AI* in *Proceedings of the 36th International Conference on Machine Learning* (eds Chaudhuri, K. & Salakhutdinov, R.) **97** (PMLR, June 2019), 2576–2585. https://proceedings.mlr.press/v97/han19a.html.

141. Wikipedia. *Categorical distribution — Wikipedia, The Free Encyclopedia* [Online; accessed 29-April-2024]. 2023. https://en.wikipedia.org/w/index.php?title=Categorical_distribution&oldid=1168534479.

142. Liu, C. & Liu, G. *JointPPO: Diving Deeper into the Effectiveness of PPO in Multi-Agent Reinforcement Learning* 2024. arXiv: 2404.11831 [cs.MA].

143. Britannica, T. Editors of Encyclopaedia. *Dictatorship* Encyclopedia Britannica. Online; accessed 2024-05-08. https://www.britannica.com/topic/dictatorship.

144. Karbasi, A. *et al. Replicability in Reinforcement Learning* 2023. arXiv: 2305.19562 [cs.LG].

145. Baker, M. 1,500 scientists lift the lid on reproducibility. *Nature* **533,** 452–454. doi:10.1038/533452a. https://doi.org/10.1038/533452a (2016).

146. With Code, P. *Resource Catalog 2022* https://paperswithcode.com/rc2022. 2022.

147. Pieters, M. *How do I iterate over a range of numbers defined by variables in Bash?* https://stackoverflow.com/a/27522708. Online; accessed 2024-05-2. 2014.

148. PyTorch Contributors. *PyTorch Documentation: Notes on Randomness* https://pytorch.org/docs/stable/notes/randomness.html. Online; accessed 2024-05-2.

149. David Goldberg. *What Every Computer Scientist Should Know About Floating-Point Arithmetic* https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html. Online; accessed 2024-05-2. 1991.

150. Huang, J. & He, N. *Robust Knowledge Transfer in Tiered Reinforcement Learning* 2023. arXiv: 2302.05534 [cs.LG].

151. Vithayathil Varghese, N. & Mahmoud, Q. H. A Survey of Multi-Task Deep Reinforcement Learning. *Electronics* **9.** ISSN: 2079-9292. doi:10.3390/electronics9091363. https://www.mdpi.com/2079-9292/9/9/1363 (2020).

152. Lazaric, A. in *Reinforcement Learning: State-of-the-Art* (eds Wiering, M. & van Otterlo, M.) 143–173 (Springer Berlin Heidelberg, Berlin, Heidelberg, 2012). ISBN: 978-3-642-27645-3. doi:10.1007/978-3-642-27645-3_5. https://doi.org/10.1007/978-3-642-27645-3_5.

153. Taylor, M. E. & Stone, P. Transfer Learning for Reinforcement Learning Domains: A Survey. **10,** 1633–1685. ISSN: 1532-4435 (Dec. 2009).

154. Zhu, Z. *et al.* Transfer Learning in Deep Reinforcement Learning: A Survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **45,** 13344–13362. doi:10.1109/TPAMI.2023.3292075 (2023).

155. D'Eramo, C. *et al. Sharing Knowledge in Multi-Task Deep Reinforcement Learning* 2024. arXiv: 2401.09561 [cs.LG].

156. Brunskill, E. & Li, L. *Sample Complexity of Multi-task Reinforcement Learning* 2013. arXiv: 1309.6821 [cs.LG].

157. Vaswani, A. *et al. Attention Is All You Need* 2023. arXiv: 1706.03762 [cs.CL].

158. Merkel, D. Docker: lightweight linux containers for consistent development and deployment. *Linux journal* **2014,** 2 (2014).

159. Bisong, E. in *Building Machine Learning and Deep Learning Models on Google Cloud Platform: A Comprehensive Guide for Beginners* 59–64 (Apress, Berkeley, CA, 2019). ISBN: 978-1-4842-4470-8. doi:10.1007/978-1-4842-4470-8_7. https://doi.org/10.1007/978-1-4842-4470-8_7.

160. Microsoft. *Visual Studio Code* https://code.visualstudio.com/. [Online; accessed 2024-09-26]. 2015.

161. Tao, S., Pan, I., *et al. LuxAI S2, pypi Package Version 2.2.0* https://pypi.org/project/luxai-s2/2.2.0/. [Online; accessed 2023-08-15]. 2023.

162. Tao, S., Pan, I., *et al. LuxAI S2, Release Version 2.2.0* https://github.com/Lux-AI-Challenge/Lux-Design-S2/releases/tag/v2.2.0-fix2. [Online; accessed 2023-08-15]. 2023.

163. Python Software Foundation. *Python 3.8.0* https://www.python.org/downloads/release/python-380/. [Online; accessed 2023-07-05]. 2019.

164. GitHub. *GitHub Copilot* https://github.com/features/copilot. [Online; accessed 2023-07-28]. 2021.

165. Grammarly, I. *Grammarly* https://app.grammarly.com. [Online; accessed 2023-09-12]. 2009.

166. OpenAI. *ChatGPT* https://openai.com/blog/chatgpt. [Online; accessed 2023-10-11]. 2023.

List of Figures

List of Tables

List of Algorithms

Appendix C

Work Distribution

The distribution of the work presented in this research can be concisely summarized by the structure outlined here. The contributions of **Takács Tamás** to this work include the following:

- Created the single-unit testbench code and implemented the training and evaluation code for all algorithms (section 2.1).
- Implemented the code and evaluated the results for the monolithic approach (section 2.3).
- Conducted ablation studies on the finalized hybrid model (subsection 3.3.5). Outlined the feature extractor structure.
- Wrote the Introduction chapter (chapter 1) and the Single-Unit Testbench (section 2.1) and Monolithic Approach (section 2.3) sections in the Methods chapter of this work. Additionally, he authored the following discussions: (section 4.1; section 4.2; section 4.6).

The contributions of **Magyar Gergely** to this work include the following:

- Greatly optimized the implementation in terms of memory usage and speed, enabling it to fit into a smaller GPU than originally required by the baseline (subsection 3.3.4).
- Theorized and developed the method of trajectory separation (subsection 2.4.5).
- Conducted the experiments to evaluate the effectiveness of trajectory separation (subsection 3.3.1; subsection 3.3.3; subsubsection 3.3.3.2).
- Wrote the Multi-Agent Environment (section 2.2) and Hybrid sections (section 2.4) within the Methods chapter of this work, authored the entire Results section (chapter 3), and contributed to the following discussions: (section 4.3; section 4.4; section 4.5).

All work mentioned above is **subject to collaborative influence**, as most tasks, decision-making, testing, and writing were carried out collaboratively rather than individually. The contribution list outlined above is a close generalization.