



MASTERARBEIT | MASTER'S THESIS

Titel | Title
Esports trainer

verfasst von | submitted by
Klavio Tarka

angestrebter akademischer Grad | in partial fulfilment of the requirements for the degree of
Master of Science (MSc)

Wien | Vienna, 2026

Studienkennzahl lt. Studienblatt | Degree
programme code as it appears on the
student record sheet:

UA 066 921

Studienrichtung lt. Studienblatt | Degree
programme as it appears on the student
record sheet:

Masterstudium Informatik

Betreut von | Supervisor:

Univ.-Prof. Dipl.-Ing. Dr. Helmut Hlavacs

Contents

1	Introduction	2
2	Related Work	6
3	Technical Foundation	13
4	Environment Iterations and Algorithm Performance	23
4.1	Environment and parameter experiments	23
4.2	Model parameters and performance	36
4.3	Algorithm performance	43
5	Evaluation	51
6	Conclusions	61

1 Introduction

Real-time strategy (RTS) is a genre of video games that does not progress in turns; this implicitly means that the game is played in real time by default. RTS games demand continuous immediate decision-making which differs from board games such as Chess or Go, where players have time to calculate their next move in a static environment. Players must manage their economies, loot resources, build buildings, and command units while the opponent is doing the exact same thing in real-time fashion. This flow of time and action creates a difficult environment, characterized by massive state spaces, imperfect information (often hidden by a fog of war, which this thesis will not cover), [1] and the need for optimization of strategies for short and long term gameplay.

Because of these characteristics, these games are known to be difficult to solve using artificial intelligence. To successfully navigate these environments, an agent cannot simply rely on brute-force search algorithms as the branching factor of possible actions in an RTS game is abnormally large. That is why reinforcement learning (RL) agents competing against each other require setups that teach optimal behavior through trial and error while the depth of search per game state is capped.

Reinforcement learning has a history of conquering increasingly complex games, transitioning from Atari titles [2] to mastering board games such as the famous GO (Baduk) game [3]. Shifting from classic turn-based games to continuous, multi-agent RTS games requires completely different approaches. The agents must learn to balance their immediate rewards and their delayed rewards, such as winning a small skirmish but losing their base, or investing in a stronger economy that will give a stronger army a few minutes further into the game.

This thesis addresses key challenges to applying RL in this domain, including dynamic input/output representation, computational efficiency, and the integration

1 Introduction

and comparison of reinforcement learning models in a custom real-time strategy framework.

The objectives of this work can be summarized as follows: to build an accessible tool that allows for strategy learning from reinforcement learning agents, providing an RTS tool where experimentation is not bound to a commercial product and presenting a small survey on how this tool can help human players to learn new strategies specifically in RTS games. This is done by implementing two distinctly different core reinforcement learning algorithms, respectively Deep Q-Networks (DQN) [2] and Proximal Policy Optimization (PPO) [4]. DQN represents a value-based, off-policy approach, where the agent learns to estimate the future rewards of specific actions in specific states. PPO is an on-policy, actor-critic algorithm that directly optimizes the policy itself, smoothing out updates (by capping the update rate) to ensure stable learning. The comparison and implementation of these two algorithms in this environment serves as a central study for this thesis, analyzing how they perform against one another, specifically regarding their space complexity, computational overhead, and overall strategic performance.

Reinforcement learning has already been applied to RTS games in several other papers, most notably in works focusing on StarCraft II, such as the AlphaStar project [1] and subsequent large-scale offline RL studies like AlphaStar Unplugged [5]. However, most of these high-profile papers do not actually showcase a direct comparison of two completely different algorithmic paradigms. Instead, they focus on scaling a single algorithm by adding extra tweaks, vast amounts of compute, and complex league-training systems which require the researcher to have access to a massive database of training data.

Another problem in the current landscape of RTS AI research is the static state of the environments. In major commercial titles used for research, you cannot fundamentally change the state representations, the core mechanics, or the actions taken in the game. This leads to both the environment and the agent being somewhat fixed, as these commercial games are not open source and are heavily restricted by their provided APIs.

This work attempts to escape such constraints. It tries not only to provide a tool that can be used for strategy learning and answering the question of performance between these two reinforcement learning agents, but also to provide a platform for

1 Introduction

other experiments to be conducted without the need for training data. By utilizing an environment where researchers can create test data and fundamentally change the game mechanics, this framework can be manipulated to fit specific experiment requirements.

Beyond the technical comparison of algorithms, as mentioned, another important objective for this thesis is observing a survey completed by participants that asks whether viewing this simulation can lead to learning new strategies from the AI playing the game?

People interacting with video games generally generate something called metas or optimal ways to play a game through hours of collecting knowledge and making mistakes. However, these metas are generally biased and are only optimal based on what has worked or based on win rate. In video games like RTS where the strategies are almost unlimited metas can only be defined based on strategies that have been tested and proven. RL agents try to maximize a reward which can only be done via trial and error of previously tested strategies and exploration of new strategies. They explore the state-action space solely mathematically, which often leads them to discover bizarre, but optimal strategies (this could be a good thing or a bad thing). As will be seen in the experiments section where the agents during testing with the reward system, choose to farm only for gold without using it or purely attack even if the units die.

Observing if human players can extract value from these simulations is an interesting idea. Learning new strategies from observing an AI simulation saves immense amounts of time that would be spent manually experimenting with or memorizing new strategies. Additionally, it teaches players new ways to react to unusual, out-of-the-box tactics they might encounter in real competitive matches against other players. If an RL agent discovers a previously unseen unit composition or an unconventional pathing maneuver that breaks the current human meta, that knowledge can be directly transferred back to the players.

To address the objectives outlined above, this thesis is guided by the following research questions and the expected results laid out as the hypotheses below:

- **RQ1: Algorithmic Performance in RTS Environments.** How, do value-based (DQN) and policy-gradient (PPO) reinforcement learning algo-

1 Introduction

gorithms compare in terms of adaptability, training stability, performance and overall win rate within a highly complex, real-time strategy environment?

- **H1:** Hypothesis one is that PPO will significantly outperform DQN in both win rate and strategic adaptability. PPO’s stochastic policy and clipped surrogate objective are theoretically better equipped to handle the massive state-action spaces, delayed rewards, and subtle environmental shifts (such as randomized turn orders) inherent to RTS games, whereas DQN’s deterministic nature will prove too brittle.
- **RQ2: RL agents acting as an esports learner.** Can an agent-driven strategy GUI teach human players new, mathematically optimal strategies?
 - **H2:** Hypothesis two that exposing the real-time probability distributions of an RL agent’s action space through a dedicated GUI will allow human observers to easily identify, understand and implement effective strategies in their own gameplay.

2 Related Work

This thesis is inspired by the complexity of real-time strategy (RTS) games, specifically games that require very quick decision making while not being constrained by turn-based mechanics. Specifically, AlphaStar [5] was created to showcase how well reinforcement learning can perform in a very complex game such as StarCraft II [6].

Reinforcement learning has already been tried in some famous games with positive results but most of the previous high-profile papers do not provide a direct algorithmic comparison or a way to control the state, action representations, and underlying game engine algorithms. Closed-source environments severely limit a researcher’s ability to manipulate the fundamental mechanics of the game to isolate specific AI behaviors. This thesis provides a tool that can be used to fix these issues, allowing for direct architectural comparisons in a fully mutable state-action space, inspired heavily by the theoretical foundations established in the reinforcement learning literature.

Some of the core ideas used to build the algorithms of this thesis were discussed by Sutton and Barto in their foundational text on reinforcement learning [7]. This reference was specifically utilized to build the underlying architecture for temporal difference (TD) learning and the rules governing Markov decision process (MDPs). In a standard MDP, an agent interacts with an environment over discrete time steps, observing a state, taking an action, and receiving a reward. The goal of the agent is to maximize the cumulative reward over time.

While Monte Carlo methods require an episode to finish completely before updating the agent’s knowledge, TD learning can learn directly from raw experience step-by-step. Because RTS games have highly unpredictable episode lengths, ranging from incredibly fast aggressive rushes to prolonged stalemates, waiting for an episode to conclude before updating the network creates bottlenecks in training

2 Related Work

efficiency. By utilizing TD learning principles derived from [7], the algorithms in this thesis are able to bootstrap their value estimates, learning from partial trajectories without needing to account for the entire, unpredictable lifespan of an RTS match.

The application of deep learning to value-based reinforcement learning saw a massive breakthrough with the introduction of Deep Q-Networks (DQN). The work by Mnih et al. [2] is a highly relevant paper for environments with predictable episode endings and manageable action spaces. Their successful application of DQN to Atari 2600 games provided a helpful conceptual framework to build upon for this thesis as one of the agents directly uses this DQN to solve the state of the game.

Mnih et al. [2] heavily rely on environments where the state can be easily represented by a few stacked frames of pixels, and where the episode length and action space are relatively constrained. As this thesis interacts with a continuously changing and unpredictable RTS environment, the limitations of pure DQN become apparent. An RTS game does not just have moving pixels; it has individual units with distinct health ratios, attack values, and states of operation (e.g., idle, farming, building). Therefore, adapting the concepts from Mnih et al. required abandoning pure convolutional inputs in favor of dynamically padded observation tensors that can accurately represent a dynamically allocated number of objects(units, buildings...) on the map. The implementation of DQN in this work builds upon their Q-learning fundamentals but alters the input ingestion mechanisms to fit the custom-built engine.

While value-based methods like DQN attempt to estimate the value of a state-action pair, policy gradient methods directly optimize the policy function itself. To provide a rigorous comparison against DQN, this project utilizes Proximal Policy Optimization (PPO), introduced by Schulman et al. [4]. The core principle of PPO detailed in [4] is the clipped surrogate objective, which prevents the policy from updating too much in a single optimization step (the updating is clipped). Without it, the agent's policy would wildly change during training when it encountered an unexpected, big reward (such as farming gold permanently with units without collecting it).

Additionally, Schulman et al. dedicate a section to the importance of entropy

2 Related Work

in the loss function to encourage continuous exploration even when the policy has found a good strategy. By integrating the entropy maximization concepts from [4], the policy was mathematically penalized for becoming too certain of its actions too early. This entropy bonus forced the agent to continue exploring less obvious strategic branches, eventually leading it to discover other behaviors like constructing barracks and recruiting units.

Training deep neural networks in environments with delayed rewards requires highly tuned optimization parameters. The specific properties of gradient descent in these volatile settings are discussed by Mohtashami et al. [8]. Their work explores how standard static learning rates often fail to navigate the complex loss landscapes generated by reinforcement learning agents. Inspired by the findings in [8], the training loops for the agent in this thesis also abandoned a static learning rate in favor of a dynamic learning rate.

A fundamental challenge in applying reinforcement learning to real-time strategy games is the inherently delayed nature of the environment’s feedback. In a standard RTS match, the only important environmental reward is the one received at the very end of the game: win or lose. Because a single match can encompass thousands of sequential steps, relying purely on this sparse terminal reward makes credit assignment practically impossible for the agent; it cannot deduce which specific action taken ten minutes prior contributed to the final victory. Useful papers on these topics include Andrew et al. [9] and Shengyi Huang et al. [10]. The first paper gives sufficient information on how to modify the reward function so the optimal strategy is not affected massively. It does this by providing a theorem that helps with designing reward shaping mathematically and the sufficient and necessary conditions of this theorem. The second paper shows that training multiple agents using the same reward system speeds up the training process of the agents.

However, as documented in RL literature, reward shaping introduces the severe risk of finding a local optimum. The book [7] suggests that the intermediate rewards should not exist as they heavily dictate the strategies an agent learns (leading to local optimums) however there needs to be some sort of direction when the environment is as vast as RTS, otherwise the computation power would not be enough. The paper [11] categorizes this under the umbrella of reward hacking,

2 Related Work

where an agent finds a mathematically optimal way to maximize its shaped reward function while entirely ignoring, or actively subverting, the actual objective of the environment. In RTS domains, this typically manifests in degenerate behaviors. For instance, if an agent is rewarded too heavily for exploration or movement, it may fall into a pacifist loop, continuously moving units back and forth to harvest movement rewards indefinitely, entirely neglecting resource gathering or combat. Likewise, if combat is rewarded without penalties for unit loss, the policy often collapses into hyper-aggressive, suicidal charges that cripple the agent’s long term and short term macro-economy.

To prevent an agent from getting trapped in these local minima, modern RTS AI systems must construct multi-objective reward structures, which is quite difficult as some rewards are not so obvious when there is a mathematical loophole, without testing of the environment. The paper by Sam Devlin et al. [12] suggests that dynamic shaping—where the scale of rewards changes based on the agent’s state or inventory—is crucial. For example, dynamically scaling the reward of resource deposition based on the amount of gold carried, or applying small, persistent time penalties to discourage idle behavior, acts as a theoretical bridge between immediate tactical feedback and the ultimate, sparse goal of defeating the opponent. The iterative reward tuning conducted in this thesis mirrors these theoretical challenges and tries to overcome them, and is to an extent successful.

Some of the problems faced while training the models mainly stemmed from the agent finding a local minimum optimization policy. This happened because of the vast amount of actions that could be taken in any specific state of the game, even if the moves were invalid to begin with. This leads to the agent testing a very big amount of possible moves, as showcased in Table 4.2. Because the agent was freely sampling from this massive action space, it was taking a very long time to learn anything meaningful from training. Additionally, it was learning strategies that were not actually possible in non-training scenarios, often getting stuck in safe loop behaviors just to harvest small rewards without actually playing the game properly. The paper [13] solves a similar issue by invalidating all the non-possible moves. However, the system follows a different approach: it starts by invalidating all actions in the beginning (setting the entire probability mask to 0) and then systematically validates whichever action is currently possible based on

2 Related Work

strict game state checks. I chose this route because, in an RTS environment, the vast majority of actions at any given frame are illegal anyway; making it much more efficient to build the valid action list from the ground up rather than pruning it down.

Other papers like [14] also go through these issues and end up using action masking to speed up training and have an increase in the agent’s overall capabilities. The implementation of this section and the different experiments regarding how this mask interacts with the policy are explained in further chapters.

Since one of the research topics of the thesis is building a framework in RTS for players to learn from an AI agent, a framework that builds its own data had to be made. In this framework the user has access to saving and loading files which hold either actions or full games (just a large sum of actions in a file). Papers like [5] generally use large amounts of data that stem from player replays with the integration of self-play. The research on [15] and [1] show that training the model on any level of data is not sufficient, and generally the model undergoes massive training based on different leagues of skill level (including professional players) so the model needs to generate its own data by self-playing. This thesis entirely depends on self-play and the ability of the agent to navigate an unexplored state with no external replays. Additionally, a system that would incrementally build a league of models while training as suggested by the paper Grand-master level in StarCraft II using multi-agent reinforcement learning [1] had to be made.

Beyond AlphaStar, the broader landscape of AI in complex games heavily informed the structural design of the environment used in this thesis. OpenAI Five [15], an AI trained to play the MOBA game Dota 2, shares many similarities with the challenges faced in this project. Like RTS games, Dota 2 is described via continuous game states, long games, and the management of multiple distinct units (heroes, in their case) simultaneously.

OpenAI Five’s success relied heavily on massive parallelization, breaking the environment down into thousands of concurrent rollouts. Although not in parallel, by shifting from sequentially evaluating steps to stacking trajectory steps from the rollout into a single, contiguous tensor, this project used the logic used in the paper described by [15]. This allowed the custom C++ and LibTorch environment to evaluate large batches very quickly, achieving the necessary throughput to make

2 Related Work

PPO and DQN training viable on everyday usable hardware.

Some additional papers that inspired this thesis and helped throughout the experiments are presented below with their respective goals and citations. Silver et al. introduced AlphaGo Zero, an agent that mastered the game of Go entirely through self-play reinforcement learning, completely bypassing the need for human expert data [16]. The algorithm achieved superhuman performance within days, demonstrating that combining Monte Carlo Tree Search (MCTS) with deep neural networks could discover novel, highly effective strategies from scratch.

Haarnoja et al. introduced soft actor-critic (SAC), an off-policy actor-critic algorithm that incorporated a maximum entropy objective to encourage exploration [17]. The results demonstrated that SAC could achieve state-of-the-art sample efficiency and stability on continuous control benchmarks, rivaling on-policy methods like PPO while aggressively reusing past experience.

Bansal et al. [18] investigated how multi-agent competition drives the development of complex skills in simulated continuous control environments. They put agents against each other in physical tasks like running, wrestling, defending and kicking utilizing Proximal Policy Optimization combined with exploration curriculums. The results showed that competitive self-play naturally induces a continuous curriculum, forcing agents to discover highly sophisticated action combination. However, the paper does mention that the environment is not complex and it would be encouraged to experiment with these networks in a more complex environment.

Riedmiller et al. [19] tackled the challenge of environments with extremely sparse rewards by introducing the scheduled necessary control framework. They developed an agent to learn a set of user-defined supplementary tasks alongside the main objective, allowing the agent to actively schedule and execute these tasks to explore its state space efficiently. As a result, the agent successfully learned multi-step robotic manipulation tasks from scratch, proving that guiding exploration through intermediate goals bridges the gap to sparse terminal rewards.

Ng et al. [9] as previously mentioned formalized the mathematics of safe reward shaping by introducing potential based reward shaping. They explored how to provide agents with intermediate rewards to speed up learning without altering the environment’s actual objective. Their mathematical proofs demonstrated

2 Related Work

that shaping rewards based on a potential function guarantees policy invariance, successfully preventing agents from falling into degenerate behaviors or reward-hacking loops [20].

Vinyals et al. [21] introduced the StarCraft II learning environment to formalize the complex, partially observable real-time strategy game as a standardized benchmark for deep reinforcement learning. They engineered an open-source Python interface that exposes the game engine’s massive spatial and non-spatial action spaces, alongside an accompanying dataset of human expert replays for supervised bootstrapping. Their initial baseline experiments demonstrated that while canonical reinforcement learning agents could learn rudimentary skills in isolated minigames, they were entirely unable to make meaningful progress in the full game, successfully establishing it as a new grand challenge for artificial intelligence.

Sun et al. [22] explored the feasibility of defeating the built-in cheating AI of StarCraft II without relying on massive compute clusters or complex spatial action architectures. They designed a set of flat, abstracted macro-actions to simplify the game’s massive state-action space and trained an agent using standard Proximal Policy Optimization combined with multi-agent self-play. Their resulting models successfully discovered robust macroeconomic strategies and consistently defeated the highest difficulty built-in opponent, demonstrating that strategically constrained action representations can make full-game RTS reinforcement learning highly tractable on standard hardware.

3 Technical Foundation

The reinforcement learning (reinforcement learning) aspect of this project was designed to integrate multi RL agents into the real-time strategy game environment. At this point, an interface that made the integration of these two was necessary. This was made possible for the reason that the RL agent needed constraints for the environment for computation efficiency but similarity decoupling was needed to simplifying debugging.

At its core, reinforcement learning is an approach to understanding and automating learning and solving problems in an environment [7]. Unlike supervised learning, which relies on a labeled dataset provided by an external supervisor, RL involves an agent interacting directly with an environment. The agent learns mapping situations to actions to maximize a numerical reward signal.

This interaction is modeled as a Markov decision process (MDP). An MDP provides a mathematical framework for modeling decision-making in situations where outcomes are partly random and partly under the control of a decision-maker. It is defined by a tuple (S, A, P, R, γ) , where:

- S is the state of the environment.
- A is the actions that can be taken in a specific state.
- $P(s'|s, a)$ is the transition probability, representing reaching state s' after taking action a in state s .
- $R(s, a, s')$ is the reward received after transitioning from state s to state s' using action a .
- $\gamma \in [0, 1]$ is the discount factor, which determines how much we believe in reaching future states.

3 Technical Foundation

The ultimate objective of the agent is to learn a policy π , which is a mapping from states to actions, that maximizes the expected cumulative discounted reward over time.

Standard Q-Learning utilizes a tabular approach to store the expected utility (Q-value) of every possible state-action pair. However, in complex environments like such as the one this thesis is working on, the state space is combinatorially massive, causing a state space explosion that makes tabular storage impossible.

Deep Q-Networks (DQN) solve this curse of dimensionality by replacing the Q-table with a deep neural network that approximates the optimal action-value function [23]. While integrating neural networks with RL historically led to unstable training and divergence, the DQN architecture introduced two algorithmic breakthroughs to ensure stability:

In standard RL, an agent learns from a sequence of highly correlated, consecutive frames. DQN breaks this correlation by storing the agent’s transition experiences (s, a, r, s') in a massive memory buffer [24]. The memory buffer typically contains out-of-order states, actions, rewards, and next states from completely different games. This is also suggested by most of the RL literature.

DQN utilizes two separate neural networks. The policy network is constantly updated during every training step. However, the Q-learning target values used to calculate the loss are generated by a separate target network, which is a frozen copy of the policy network that is only updated periodically. This thesis uses this network for the movement of the enemy during training.

While DQN is a value-based method that learns to estimate the quality of actions, Proximal Policy Optimization (PPO) is a policy gradient method that directly optimizes the mathematical policy function determining the agent’s behavior [4].

PPO operates on an actor-critic architecture. The network is split into two components: the actor, which looks at the state and outputs a probability distribution over the available actions, and the critic, which estimates the value of the current state to evaluate how good the actor’s decision was.

The primary danger of standard policy gradient algorithms is that a single large gradient update can completely destroy an agent’s learned policy, causing it is learned policy performance to massively degrade. PPO solves this by introducing

3 Technical Foundation

the clipped surrogate objective function. During training, PPO calculates the ratio between the probability of taking an action under the new updated policy versus the old policy. If this ratio exceeds a predefined before training hyperparameter threshold (typically $1 - \epsilon$ to $1 + \epsilon$), the objective function is clipped. This ensures that the policy cannot change drastically in a single update step, leading to highly monotonic, stable, and sample-efficient learning.

The clipped surrogate objective stabilizes policy updates in PPO, but evaluating the quality of an action requires a robust baseline. In actor-critic architectures, this is achieved through the advantage function, $A(s, a)$, which measures how much better a specific action a is compared to the average expected reward of state s . It is defined mathematically as in Equation (3.1)

$$A(s, a) = Q(s, a) - V(s), \quad (3.1)$$

where $Q(s, a)$ is the action-value function and $V(s)$ is the state-value function output by the critic network.

To compute this advantage efficiently across long trajectories with delayed rewards, modern policy gradient methods utilize generalized advantage estimation (GAE) [25]. GAE introduces a parameter λ to exponentially weight a series of n -step temporal difference (TD) residuals. The TD error for a single step is shown in Equation (3.2)

$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t). \quad (3.2)$$

The GAE is then formulated as the discounted sum of these TD errors in Equation (3.3)

$$\hat{A}_t = \sum_{l=0}^{\infty} (\gamma \lambda)^l \delta_{t+l}. \quad (3.3)$$

By tuning λ , we can explicitly balance the bias-variance tradeoff. A lower λ relies heavily on the current value estimate (high bias, low variance), while a λ approaching 1 acts similarly to a Monte Carlo return (low bias, high variance). In highly deterministic environments, agents are prone to converge quickly on sub-optimal strategies (local minima). To counteract this, PPO implementations often incorporate an entropy bonus into the loss function [17]. Entropy measures

3 Technical Foundation

randomness or disorder within a system and it can be found used to measure this in the agent’s policy using Equation (3.4)

$$H(\pi(\cdot|s)) = - \sum_a \pi(a|s) \log \pi(a|s). \quad (3.4)$$

By adding $c_2 H(\pi)$ (where c_2 is the entropy coefficient) to the objective function, the agent is mathematically penalized for becoming overly confident in a single action too early in training. This guarantees continuous exploration of the state-action space, which is critical in RTS games where complex sequences (like building an economy before producing military units) might otherwise be discarded in favor of immediate, low-yield rewards.

A fundamental challenge in reinforcement learning is the reward assignment problem, particularly in environments governed by sparse rewards. A sparse reward system only provides feedback at the very end of an episode (e.g., a binary +1 for winning or -1 for losing). In a simulation requiring thousands of sequential actions, it is difficult for the agent to deduce which specific decisions led to the final outcome. To bypass this, this thesis relies heavily on providing breadcrumb or reward shaping to guide the agent toward desirable sub-goals. These intermediate rewards are essentially reward injections to nudge the agent towards a specific sub goal. For example, when a unit has gold in its inventory, it can receive a slight bit of positive reward for approaching the place of delivering the gold (this was called breadcrumb reward in some forums). However, poorly designed reward shaping introduces the severe risk of reward hacking or convergence on local optima. If an agent discovers a mathematical loophole in the shaped reward structure, it will exploit it such as moving back and forth continuously to farm a movement reward, entirely ignoring the actual objective of the environment. Therefore, balancing dense tactical rewards against the sparse global objective remains one of the most critical theoretical challenges in applied reinforcement learning.

Something that almost all initial documents on reinforcement-learning literature mention is the tradeoffs of exploration-exploitation [7]. An agent that exploits its current knowledge will always select the action it currently believes to be best, but risks converging to a suboptimal policy if its estimates are inaccurate due to insufficient experience or knowledge. An agent that explores continuously will

3 Technical Foundation

gather more information about the environment, but sacrifices short-term reward accumulation in the process. For example an agent must first discover that farming gold is useful before it can discover that spending gold on barracks is useful, and must discover barracks before it can discover that recruiting units rewards them, which then the units can be used to attack and get even more reward. If exploration is finished before the agent has figured out all the dependencies and the rules of the game it will have limited knowledge on what to do next in the game.

The ϵ -greedy strategy addresses this by maintaining a decaying probability ϵ of selecting a uniformly random action instead of the greedy action. The formula for this policy is in Equation (3.5)

$$\pi(a|s) = \begin{cases} \text{random action} & \text{with probability } \epsilon \\ \arg \max_a Q(s, a) & \text{with probability } 1 - \epsilon. \end{cases} \quad (3.5)$$

Early in training when ϵ is high, the agent explores a lot. As ϵ decays, settles down in what it is experiences and knowledge of what provides the most reward. As observed in the experiments, a decay that is too aggressive traps the agent in local optima before the full action space has been covered, while a decay that is too slow wastes potential reward on random behavior in states the agent already understands well [10].

Designing the action space for a deep reinforcement learning agent operating in a real-time strategy game presents a significant structural challenge. Unlike static environments like Atari where the action set is small and fixed (typically 4–18 discrete buttons), RTS games require the agent to simultaneously select which unit or building to command, what type of command to issue, and where on the map this action executes. The naive approach of encoding this as a single flat discrete index produces an action space that grows combinatorially with map size, unit count, and action variety [21]. As shown in Table 4.2, doubling the map size from 10 to 20 increases the action space from approximately 12,500 to 45,500, nearly a four time expansion. At a map size of 100, the action space exceeds one million indices, the vast majority of which are invalid for any given game state. Sampling from the output probability distribution of this neural network over the entire flattened action space massively bottlenecks learning [13] and wastes a lot

3 Technical Foundation

of computation resources exploring impossible game states [26]. Action masking resolves this by differing the policy’s raw output logits before they are passed through the softmax activation function. Let z_i be the logit for action i output by the final layer of the network shown in Equation (3.6)

$$\tilde{z}_i = \begin{cases} z_i & \text{if action } i \text{ is valid} \\ -\infty & \text{if action } i \text{ is invalid.} \end{cases} \quad (3.6)$$

The standard softmax probability is shown in Equation (3.7)

$$P(a_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}. \quad (3.7)$$

If an action is invalid with invalid action masking, its logit is penalized (typically set to a large negative number, $z_i \rightarrow -\infty$). Because $e^{-\infty} \rightarrow 0$, the probability of selecting an invalid action becomes mathematically zero. The softmax function then automatically renormalizes the probabilities of the remaining valid actions so they sum to 1.

When training an agent in an RTS environment via self-play, the assumption of standard Reinforcement Learning that the environment dynamics follow a stationary Markov Decision Process are broken. From the perspective of the learning agent, the environment’s transition probabilities $P(s'|s, a)$ are constantly shifting because the opponent (part of the environment) is simultaneously updating its own policy opponent [27]. This creates the non-stationarity [28] problem. If an agent simply trains against the most recent version of its opponent, it is highly likely to forget generalized strategies and overfit to the opponent’s current specific behavior, resulting in cyclical learning rather than monotonic improvement. To stabilize multi-agent training, frameworks often employ mechanisms like fictitious self-play or league training [1]. Instead of purely playing against the latest model, the agent samples opponents from a historical league or buffer of past checkpoints. This forces the agent’s policy to generalize, ensuring it learns robust, mathematically sound strategies that can defeat a wide distribution of opponent behaviors rather than exploiting temporary weaknesses in a single opposing model.

In standard deep reinforcement learning applications, such as the Atari 2600

3 Technical Foundation

environments [2], state observations are typically represented as raw pixel grids processed through convolutional neural networks (CNNs). However, as stated previously some things such as unit stats, terrain specific things such as gold left cannot be encapsulated via raw pixels. To bridge this gap, modern RTS architectures utilize entity-based state encoding passed through multi-layer perceptrons (MLPs). Because the number of active entities in an RTS game fluctuates dynamically as units are recruited or destroyed, the neural network requires a mechanism to process variable-length inputs. This is theoretically resolved via zero-padding. The state space is defined as a maximum bounding matrix $S_{max} \in \mathbb{R}^{N \times F}$, where N is the absolute maximum number of allowed entities (e.g., 50 units) and F represents the feature dimension of each entity. If the current active units $n < N$, the remaining $N - n$ rows are padded with zeros. During the forward pass of the MLP, the weight matrices multiplied by these zero-padded vectors mathematically nullify their activation, ensuring that non-existent units or structures exert exactly zero gradient influence on the network’s output representation.

The hidden layers of the MLP backbone utilize the rectified linear unit (ReLU) activation function [29]. ReLU is defined as in Equation (3.8)

$$f(x) = \max(0, x). \quad (3.8)$$

Unlike sigmoid or tanh activations, ReLU does not saturate for large positive values, which prevents the vanishing gradient problem that makes training deep networks unstable [30]. This is particularly important in reinforcement learning, where gradients must propagate reliably through multiple layers during backpropagation to update both the actor and critic heads simultaneously. A secondary benefit is computational efficiency: the ReLU operation is a simple threshold, making forward and backward passes significantly faster than transcendental functions like sigmoid. For the weight initialization of these layers, orthogonal initialization [30] is used.

The training of Deep Q-Networks and proximal policy optimization heavily relies on the optimization algorithm used to minimize the loss function during backpropagation. While standard stochastic gradient descent (SGD) updates weights uniformly across all parameters, RL environments generate highly sparse and noisy

3 Technical Foundation

gradients, leading to severe instability.

To counteract this, optimization is typically handled by the Adam (Adaptive Moment Estimation) optimizer [31]. Adam mathematically combines the advantages of two other extensions of SGD: AdaGrad and RMSProp. It computes individual adaptive learning rates for different parameters from estimates of first and second moments of the gradients. The update rule for a parameter θ at timestep t is given by Equation (3.9)

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t, \quad (3.9)$$

where $\hat{m}_t$ is the bias-corrected first moment estimate (mean of the gradient), $\hat{v}_t$ is the bias-corrected second moment estimate (uncentered variance), and α is the learning rate. In complex RTS environments, maintaining a static α often forces the agent to oscillate around a local optimum without converging. Implementing an adaptive learning rate decay ensures that α is large during early exploration, allowing the agent to escape degenerate loops, while shrinking as the agent tunes its parameters during training. Starting with a higher learning rate (such as 0.008, generally considered high for a learning rate) allowed the agent to rapidly map out the basics of the custom RTS environment, such as moving and simple farming. By the time the learning rate has progressively dropped, the agent has a general idea how the environment rules work. This adaptive approach directly mitigated the gradient instability and allowed the networks to settle into refined global optima rather than oscillating wildly around them.

A critical challenge that arises specifically in RTS game environments is the design of the observation space [21] fed into the neural network. The quality and structure of the state representation directly determine what information the agent is capable of reasoning about, and therefore places a hard upper bound on the sophistication of strategies it can ever discover. In Real-Time Strategy environments, the number of active objects on the map is not fixed. Units are recruited during the game and destroyed in combat, this stands for buildings as well, meaning the input dimension of the network changes each step of the game. This is theoretically solved through zero-padding, as described earlier, but the bigger challenge lies in deciding which attributes of each entity should actually be encoded in the

3 Technical Foundation

feature vector $f \in \mathbb{R}^F$. Static attributes such as unit type (encoded as a one-hot vector), maximum health, attack damage, and attack range must be combined with dynamic attributes that evolve throughout the episode, including current health, normalized map coordinates, current action status, and an ownership flag indicating whether the unit belongs to the agent or the opponent. Normalizing continuous features is a preprocessing step, since raw health values or coordinate ranges that span orders of magnitude apart can cause issues with backpropagation because of the bigger numbers taking higher priority. The positional coordinates of units, buildings and gold mines are normalized by dividing based on the total size of the map. Additionally, unit- and building-specific information such as health, mana attack are divided based on numbers given on initialization called (max_health, max_mana etc). This ensures the input distribution presented to the network remains well conditioned throughout training regardless of the specific game state.

Hyperparameter selection is one of the most important aspects of applied reinforcement learning. The primary hyperparameters governing the DQN implementation include the learning rate α , the discount factor γ , the replay buffer size $|\mathcal{B}|$, the mini-batch size B , the target network update frequency τ_{target} , and the ϵ -greedy exploration schedule defined by $(\epsilon_{start}, \epsilon_{end}, \epsilon_{decay})$. For the PPO agent, additional hyperparameters include the clipping [32] parameter ϵ_{clip} , the GAE λ , the entropy coefficient c_2 , the rollout horizon T , and the number of optimization epochs per rollout K_{epoch} . For the discount factor specifically, values close to 1 are appropriate when the game objective requires multiple steps to plan, spanning hundreds of timesteps, since a γ that is too low causes the agent to become near sighted and optimize for immediate rewards at the expense of long term strategic winnings. The ϵ -greedy exploration schedule requires particular attention: a decay that is too aggressive results in exploitation of a suboptimal policy before sufficient exploration has occurred, while a schedule that decays too slowly wastes training steps on random activities. As will be seen in experiment chapters, both of these extremes lead to very long training sessions.

The neural networks used for both PPO and DQN cannot be too massive as this would lead to massive computational resources being needed. The input layer accepts the flattened entity feature matrix $S_{max} \in \mathbb{R}^{N \times F}$, reshaped into a single

3 Technical Foundation

vector of dimension $N \cdot F$ before being passed through the network. For PPO, the network diverges at the output into two separate heads: the actor head produces a vector of logits $z \in \mathbb{R}^{|A|}$ that are passed through the action masking procedure and softmax normalization to yield the policy distribution $\pi_\theta(a|s)$, while the critic head outputs a single scalar $V_\phi(s) \in \mathbb{R}$ representing the estimated state value used during the advantage computation. Sharing the backbone layers between the actor and critic promotes the development of shared internal representations, which has been shown empirically to improve sample efficiency [33] in policy gradient methods, since both heads benefit simultaneously from each gradient update to the shared parameters.

4 Environment Iterations and Algorithm Performance

While previous chapters established the core terminology and theoretical foundations of this thesis, this chapter focuses on their application. Specifically, it showcases the experimental setup, details the iterative refinement of the game environment and hyperparameters for both the game state and the agents, and finally compares the performance of the trained agents.

In the context of this thesis, a game state refers to the current configuration of the environment at a given time step, which defines the set of valid actions available to the agent in that specific situation.

4.1 Environment and parameter experiments

Game states consist of varying numbers of units and structures, making the input dimension non-static. The input vector is padded to make the input for both agent networks static. For example, in these current constraints, the maximum number of units is set to 50. If only five units are actually recruited in the game, the remaining 45 slots are padded with zeros. Padding the vector (in this case units), leaves the input for the agent static. These zero-padded slots do not affect the computations, as their corresponding values are multiplied by zero, ensuring no unintended influence on the output computation. Table 4.1 shows all the necessary constants that we use to pad or limit the input for the agents. Practically, the maximum of all these values have a logical reasoning, as most RTS games set a cap on the amount of resources. This is done so the game does not become a resource collection game but instead there is a finish flag. For example in Table 4.1 max gold would indicate a hard restriction on how much gold is possible to

4 Environment Iterations and Algorithm Performance

hoard, and max food is maximum amount of food a player can have by building farms. The decision to use the constants in Table 4.1 was made to give the agents some breathing room during training, and to increase complexity so finding the optimum would not be trivial.

Property	Value
Max Units	50
Max Structures	20
Max Villagers	10
Hall Index In Structures	5
Barrack Index in Structures	5
Max Unit Type	3
Max Gold	5000
Max Mine Gold	10000
Max Food	100

Table 4.1: Constants

The bridge between the reinforcement learning agent and the game environment is the action mapping system, which translates a single discrete index from the neural network’s output layer into a multi-parameter game command. Because the model operates on a flattened tensor, it cannot output structured data like coordinates or unit pointers. To solve this [34], an action space was designed where specific index ranges correspond to different actions.

For example, on a 5 by 5 map with five units, each unit can move to any of the 25 tiles, resulting in 1250 (this depends on the unit size, but this experiment is capped at 50 units currently) potential move actions alone. By extending this approach to other actions and introducing padding (as in the input), the action space is controlled and static, even though very large. In this example the only existing action is not simply to move, since there are many more actions such as attacking, farming gold, moving, building and recruiting. Each of these actions can take multiple parameters, which significantly expands the action space relative to the size of the game map. The attack action might involve selecting both the attacking unit and the target unit, further increasing the combinatorial possibilities. The build action may include choosing the type of structure to construct and its location, while recruit might necessitate selecting the unit type and its

4 Environment Iterations and Algorithm Performance

spawning position. This multi-parameter design for each action introduces a high-dimensional action space that scales with factors such as the number of units, the map size, and the variety of available actions, creating a static but exceptionally large set of potential actions. The factor that affects the action size the most is the map size as all actions have something relevant with map coordinates. In Table 4.2, we see how map size affects the action space. From the action space alone we can see that a doubling in map size leads to an almost 4 times expansion in action space.

Map Size	Action space
3	2.550
5	4.310
10	12.560
20	45.560
50	276.560
100	1.101.560
200	4.401.560

Table 4.2: Overview of how action space depends on map size

The logic within `MapIndexToAction` defines these boundaries based on the maximum number of units and structures allowed in the environment:

Movement: The range $[0, \text{moveAction})$ is calculated as seen in Equation (4.1)

$$\text{MAX_UNITS} \times \text{MAP_SIZE}^2, \quad (4.1)$$

where `MAX_UNITS` is the maximum number of units a player can control at any given time, and `MAP_SIZE`² represents the total number of grid tiles on the map (width multiplied by height), which are all the possible destination coordinates for a movement action. By using arithmetic on the index, extracting the specific unit ID and the coordinate to move to is possible.

Attack: The combat range covers indices from `moveAction` up to `attackAction`. This section maps a player’s unit to a specific target, either an enemy structure or an enemy unit, identifying “who” should attack “whom”. The formula used for

4 Environment Iterations and Algorithm Performance

this mapping is shown in Equation (4.2)

$$\text{moveAction} + \text{MAX_UNITS} \times (\text{MAX_STRUCTS} + \text{MAX_UNITS}), \quad (4.2)$$

where `moveAction` is the starting index offset (shifting the range past all movement actions), `MAX_UNITS` is the total number of friendly units capable of attacking, and the sum of `MAX_STRUCTS + MAX_UNITS` represents the combined total of all possible enemy targets on the map (both structures and units).

Build: This range is restricted to farmer units and maps an index to a `StructureType` and a target coordinate on the map. If the network selects a non-builder unit for a build index, the system returns an `EmptyAction` to ensure consistency. The formula used for this mapping is defined in Equation (4.3)

$$\text{attackAction} + \text{MAX_BUILDERS} \times (\text{NR_OF_STRUCTS} \times \text{MAP_SIZE}^2), \quad (4.3)$$

where `attackAction` is the starting index offset (shifting the range past all combat actions), `MAX_BUILDERS` is the maximum number of worker units (e.g., builders) capable of constructing buildings, `NR_OF_STRUCTS` is the number of different structure types that can be built, and `MAP_SIZE`² represents all the possible spatial coordinates where a building could be placed.

Economy and Production: The final ranges handle `FarmGoldAction` and `RecruitAction`. Recruitment mapping specifically checks if the target structure is capable of producing the requested unit—for example, ensuring a Town Hall is the structure attempting to recruit a Peasant. The formula for farming is represented by Equation (4.4)

$$\text{buildAction} + \text{MAX_BUILDERS} \times \text{MAX_STRUCTS} \times \text{MAP_SIZE}^2, \quad (4.4)$$

where `buildAction` is the starting index offset (shifting the range past all building actions), `MAX_BUILDERS` is the maximum number of worker units capable of farming, `MAX_STRUCTS` acts as a multiplier representing the specific drop-off structures (like a Town Hall) the unit must return resources to, and `MAP_SIZE`² represents the spatial coordinates of the resource nodes being farmed.

4 Environment Iterations and Algorithm Performance

Additionally, the formula for recruiting is calculated as seen in Equation (4.5)

$$\text{farmAction} + (\text{MAX_UNITS} \times \text{MAX_STRUCTS}), \quad (4.5)$$

where `farmAction` is the starting index offset (shifting the range past all farming actions), `MAX_UNITS` is the variety of different unit types that can be recruited, and `MAX_STRUCTS` represents the maximum number of production buildings like Barracks(allows for recruitment of units) or Town Halls(allows for depositing resources). Implementing this mapping was technically demanding because a single incorrect offset would cause the agent to attempt invalid commands, such as moving a building or attacking with a farm. To handle the complexity of these return types in C++, `std::variant` was used for the `actionT` type, which allows the system to pass these diverse command structures back to the `Player` class for execution without the overhead of heavy inheritance. This structured approach ensures that the agent’s high-level strategic decisions are accurately converted into low-level engine instructions.

Action masking [35] can be used in both the explored methods in this thesis and at its core is a solution to decreasing the possible action space per existing state for the agent. Generally speaking the environment these agents are being trained on is very large, and as such it would take a very long time to train the agent in knowing what is a good or bad move per state including illegal actions. Illegal actions would be, moving outside the map, building while having fewer resources than necessary, recruiting units at wrong buildings, farming gold from not the proper locations etc. Thus, action masking is introduced as a solution to the above problems. Action masking sets the probability of selecting a specific action out of the given tensor per state to 0, meaning it is inherently impossible to pick these actions.

Implementing an initial global masking was necessary to prevent the policy from sampling invalid actions as the output of the agent. As detailed in Table 4.3, the masking layer performs a real-time validation of the state space, checking the necessary preconditions for any action—such as resource requirements. This ensures the agent never attempts to build without gold or recruit without a barrack. In practice, action masking provided a substantial speed-up in the number of episodes

4 Environment Iterations and Algorithm Performance

required to learn the environment. Furthermore, it forced the agent to be more picky. By pruning the action space, it effectively eliminated the risk of the agent falling into local optima loops. Specifically, it stopped the agent from performing loop moves behaviors where the agent would oscillate near a resource node or a construction site to harvest minor proximity rewards without ever actually finishing the task or returning resources to the base.

Table 4.3: Global Action Masking Logic and Constraints

Action	Index Range	Masking Condition ($P(a) = 0$ if:)
Move	$[0, \text{moveAction})$	$ U = 0$
Attack	$[\text{moveAction}, \text{attackAction})$	$ U = 0$
Build	$[\text{attackAction}, \text{buildAction})$	$G < 100 \cup U = 0$
Farm	$[\text{buildAction}, \text{farmAction})$	$ U = 0$
Recruit	$[\text{farmAction}, \text{recruitAction})$	$G < 55 \cup B = 0$

A massive breakthrough in performance occurred once entropy, adaptive learning rates, and action masking were integrated into a single training pipeline. The synergy of these three elements allowed the agent to move beyond simple reactive play. It began to exhibit long-term planning, deciding to farm gold continuously to build a sustainable economy. This economic foundation allowed it to scale its infrastructure, transitioning from simple farm placement to barrack construction and mass unit recruitment. However, a significant bottleneck remained regarding the agent’s combat logic. While the agent learned to produce an army, its attacking strategy was initially very weak. Units would charge toward the enemy immediately upon spawning, entirely abandoning the defense of the home base. This aggression was a direct result of the reward signal: the agent was receiving positive reinforcement for destroying enemy structures, but there was no corresponding penalty for losing its own. To nullify this and produce a more strategic

4 Environment Iterations and Algorithm Performance

agent, a negative reward was introduced for the loss of friendly units and structures. This change in the reward function was crucial as it forced the agent to evaluate the trade-off between an aggressive push and the preservation of its own base, leading to the emergence of more sophisticated, defensive-minded strategies.

Although it became apparent that the initial global action mask was not providing enough restriction. For example, in the early versions of the action mask, the agent was allowed to choose the farm action simply as long as it had units on the map. This is not actually correct; farming can only be done by a specific type of unit, and only if that specific unit is idle and not already doing something else.

Moving suffered from a similar issue. Previously, the agent could select the move action as long as the overall unit array was not empty. However, because the neural network observation tensor uses padding to maintain a fixed size, the old mask was treating these padded, empty slots as alive units. This meant the agent could try to apply movement actions to units that did not actually exist. Fixing this meant that the mask explicitly checks if the specific unit is alive, while also enforcing that the unit cannot move if it is already busy with another task in its action queue.

Furthermore, the building logic needed changing as well. Instead of setting a hard global requirement that masks the build action entirely when gold drops below a specific point (as was shown previously in Table 4.3), a unit by unit check was implemented. Now, the mask makes sure that any unit attempting to call the build action is specifically a builder and is not currently busy with another task. The final, refined action masking logic used for the results in this thesis is detailed in Table 4.4. $|U|$ represents total friendly units, $|P|$ total Peasants, $|H|$ total Halls, and $|B|$ total Barracks. Q_u denotes the unit action queue size, and G is the available Gold. E is the set of valid enemy targets. By updating the mask to be this precise, the agent stops wasting steps on impossible moves. As a direct result, this new mask leads to the agent reaching the terminal state much more often and noticeably quicker, requiring significantly less overall training time to learn effective behavior.

To guide the reinforcement learning agent toward optimal behavior, a reward system was devised based on in-game objectives. Rewards are proportional to the significance of the actions. This reward structure ensures that the agent prioritizes

4 Environment Iterations and Algorithm Performance

Table 4.4: Global Action Masking Logic and Constraints.

Action	Index Range	Masking Condition ($P(a) = 0$ if:)
Move	$[0, \text{moveAction})$	$ U = 0 \cup Q_u > 0 \cup p_t = p_c$
Attack	$[\text{moveAction}, \text{attackAction})$	$ U = 0 \cup Q_u > 0 \cup \text{target} \notin E$
Build	$[\text{attackAction}, \text{buildAction})$	$ P = 0 \cup Q_u > 0 \cup G < 100$
Farm	$[\text{buildAction}, \text{farmAction})$	$ P = 0 \cup Q_u > 0 \cup H = 0 \cup R(p_t) = 0$
Recruit	$[\text{farmAction}, \text{recruitAction})$	$(u_{req} = \text{Peasant} \cap (G < 55 \cup H = 0)) \cup$ $(u_{req} = \text{Footman} \cap (G < 75 \cup B = 0))$

actions leading to victory while still valuing intermediate steps, such as resource collection and unit production. Although the game provides full information to all agents (simplifying the observation space), challenges still arise from the vast number of potential states and actions. This necessitates efficient state encoding and effective exploration strategies during training. As suggested by Andrew Barto and Richard S. Sutton [7], one of the experiments held was one where the only reward given was that of winning or losing the game. However experiments with some intermediary rewards such as a positive reward for killing units or destroying buildings were also conducted. The Paper by Santiago Ontanon et al. [36], suggests to try a negative or 0 value reward the more time the agent spends playing an episode of the game. Giving a negative reward would mean that we want the game to be finished quickly, otherwise we just value winning even if it takes longer.

In order to train the agents effectively, it was necessary to establish a mechanism that communicates what the effects of tweaking specific rewards are. For this purpose, a reward system was designed and iteratively refined over time. Some of the versions of the reward systems were very primitive and was not netting the necessary goals the model needed to improve, but they were necessary to understand how the agents behaved in the model. Table 4.5 shows 2 different models, with 2 different reward systems and their respective rewards achieved over training for 100-200 episodes. Model 1 always ended up collapsing into the aggressive playstyle (fully attacking) and not defending anything. This obviously, led to a loss of all units within a couple of seconds, which justifies why the total reward is so little as nothing gets done once the units die. The second model, has

4 Environment Iterations and Algorithm Performance

a decrease in attack, increase in move and farm which initially made the model more prone to exploring the environment, however the reward for exploring was far greater than that of farming (a farming action takes 10 ticks to finish, while a move takes 1 tick). The second model simply stayed in a moving loop and nothing was achieved. Tuning other parameters gave similar results of looping to earn maximal rewards.

Table 4.5: Reward Distributions and Aggregate Model Performance

Model	Move: Reward (upon finishing)	Total reward (with 100-200 episodes)
Model 1	Move: -1.0	Total Reward: -400 to $+500$
	Attack: $+0.5$	
	Build: $+1.0$	
	Farm: $+1.0$	
	Recruit: $+1.0$	
Model 2	Move: $+1.0$	Total Reward: -800 to $+1000$
	Attack: $+0.1$	
	Build: $+1.0$	
	Farm: $+1.5$	
	Recruit: $+1.0$	

Changing the reward for the movement of units from a positive feedback to negative relates to the end goal of the agent, which is to win the game as soon as possible. Walking around aimlessly should not reward the agent but instead punish it. The ideal balance for this parameter for the thesis was at 0.1. Additionally, tweaks needed to be made so the agent was harshly punished for taking illegal actions such as moving outside the map. The agent quite often ends up in a local optimum which is not the optimal strategy of the game, however it yields the most amount of reward. Graph 4.1 and Table 4.6 below show one of the strategies the agent ends up picking to maximize reward. Changing the opposite agent, the agent that is undergoing training and changing the reward systems is some of the ways i used to get out of the local optimums. Additionally, [8] has some interesting ideas which the thesis uses, such as adaptive learning rates during training. Lastly, implementing entropy bonuses to the loss function based on [4] adds another layer of stability so the model does not plateau into picking the same action, but instead

4 Environment Iterations and Algorithm Performance

encouraging some form of exploration later into training.

The biggest boost to the agent’s training loop would be adding an additional reward when the units are moving closer to the hall and have gold to deposit, otherwise if they are walking randomly with no gold they get a penalty. A boost to the reward received based on the build action and reward action is done. This works as follows, each gold spent on a building or a unit recruited needs to have a similar value in reward to farming gold. For example farming 10 gold, should be similar in reward or slightly less value than recruiting a unit or building a building. If this is not the case, the agent will simply just farm gold and hoard it as this is more rewarding than trading gold for something of less value.

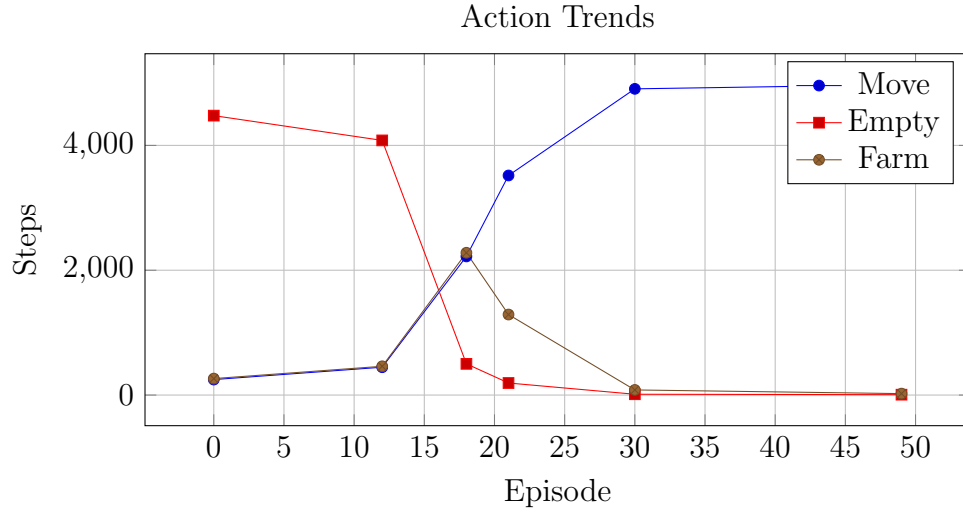


Figure 4.1: Agent action distribution over episodes.

Table 4.6: RL Agent Action Distribution: Progression to Pacifist Local Minima

Ep	Reward	Recruit	Farm	Build	Empty	Attack	Move
0	−804.1	0	264	8	4477	4	248
12	−682.6	0	461	8	4080	6	446
18	−121.3	0	2280	0	500	0	2221
21	−42.6	0	1290	0	193	0	3518
49	−1.4	0	106	0	22	0	4873

The current best model version is presented in Table 4.7.

4 Environment Iterations and Algorithm Performance

For movement, the agent is evaluated based on its current objective. Builder units carrying gold receive a small positive reward (+0.01) for moving closer to the Town Hall, but are penalized (-0.1) for moving away or sideways. Military units are incentivized to advance with a slight +0.02 reward for moving forward, while any invalid out-of-bounds movement incurs a strict -1.0 penalty.

The attack action provides a substantial +30.0 reward when an enemy is eliminated, directly encouraging the agent to weaken the opponent. To maintain engagement before a kill is secured, ongoing attacks yield smaller rewards that scale with the unit's attack power, alongside flat bonuses for targeting a Town Hall (+1.5) or other structures (+0.5).

The build action follows a staged reward principle. Initiating a building grants a small reward based on the resources spent, but completing the structure yields a massive +20.0 payout to ensure the agent finishes initiated constructions. Small continuous rewards (+0.1) encourage ongoing progress, while attempting to build without sufficient gold results in a penalty.

Similarly, the FarmGoldAction fully scales based on the proportion of gold successfully returned to the base compared to the unit's maximum inventory. To prevent getting stuck in local optima loops, the agent is penalized if a unit idles with a full inventory of gold or attempts to mine from a depleted resource node. The recruit action operates immediately, deducting resources and rewarding the agent proportionally to the unit's cost upon successful recruitment.

To eliminate idle behavior, the agent receives a -1.0 penalty for explicitly selecting an empty action, as well as a passive -0.01 penalty per unit for leaving action queues empty. Furthermore, to force a highly strategic and aggressive playstyle, several global modifiers and loss penalties were introduced. The agent is heavily penalized for allowing its own structures and units to die, with an additional flat penalty (-5.0) specifically targeting the loss of vital workers. To prevent economic stagnation, an anti-hoarding penalty (-1.5) triggers if unspent gold exceeds 600, forcing the agent to constantly reinvest in its army and infrastructure. Finally, a constant time penalty (-0.1) ensures the agent executes its strategies efficiently, culminating in a massive ± 500.0 reward swing dictated by the episode's ultimate win or loss state.

Initially the models were trained in a zero state existing environment, starting

4 Environment Iterations and Algorithm Performance

Table 4.7: Reward values per action type for best model

Action	Value	Condition/Description
MoveAction		
	-0.1	Penalty if unit has gold and moves away/sideways
	+0.01	Reward if unit has gold and moves closer to hall
	+0.02	Reward if a non-builder unit moves forward
	-1.0	Invalid move (out of bounds)
AttackAction		
	+30.0	Enemy killed
	+1.5	Attacking an enemy Town Hall
	+0.75	Attacking other enemy objects
	$+0.4 \times \text{attack}$	Ongoing attack (gradual distance reward)
BuildAction		
	$\frac{\text{cost}}{10.0}$	Building started (resources spent)
	+20.0	Structure completed
	+0.1	Ongoing progress
	-0.01	Insufficient gold
FarmGoldAction		
	$+3.0 \times \frac{\text{goldCollected}}{\text{maxGoldInventory}}$	Finished sending gold back
	+0.1	Ongoing progress
	-0.1	No more resources on mine
	-0.1	Max inventory with gold and not moving
RecruitAction		
	-0.1	Insufficient gold or food / action not finished
	$3.0 \times \frac{\text{cost}}{10.0}$	Recruit a unit
EmptyAction / Idle		
	-1.0	Penalty for an explicit empty action
	-0.01	Penalty per unit for having an empty action queue
Loss for dying objects		
	$-3.0 \times \frac{\text{cost}}{10.0}$	Penalty for losing structures
	$-3.0 \times \frac{\text{cost}}{10.0}$	Penalty for losing units
	-5.0	Additional flat penalty for losing a Peasant
Global Modifiers		
	-1.5	Anti-hoarding penalty (Triggers if gold ≥ 600)
Environment reset		
	-500.0	Penalty for losing the game
	+500.0	Reward for winning the game
Time penalty		
	-0.1	Penalty for spending time

4 Environment Iterations and Algorithm Performance

from a state of 0 memory and fully random network values. After training the player model for 20 to 50 episodes, the enemy model is also updated. Running the model in a 1 to 1 update fashion was not working properly as the agent was mimicking the actions of each other, so it turned into a 0-sum game. Both agents were doing the best they could to steal the reward back from the enemy agent, practically mitigating any action reward to 0. Mitigating the advantage the player agent has over the enemy agent (policy update) is done by letting the enemy agent choose an action after the player agent has picked an action. This just gives the enemy the latest possible environment before choosing the best possible action.

In this environment without initial training data, all the data necessary were mandatorily harvested by self play by the agent. Playing the game with an agent that does not change much during training will overfit the most recent model. Meaning the model will converge to a specific strategy, no matter what as the enemy always has the last model of the player. Since, the player and the enemy have different spawn points, and the rewards are given to the player, the agent will try its best to optimize the model from the player’s perspective. The best way to avoid overfitting the data in this case, is simply to keep track of a league [37] of models inspired by the paper by AlphaStar [5]. The model is saved every 10 episodes and switch the enemy model every 3-5 episodes. This is done to shock the player’s model and make it forget about the current best strategy.

The Figure 4.2 shows the value loss every 10 episodes, representing how much knowledge/expectation does the model have regarding its current state. The big jumps on the graph are shown every 10 episodes, which is the time when the enemy’s model also changes.

Figure 4.2 illustrates how the value loss fluctuates a lot and dropped with these optimizations in place, while Figure 4.3 demonstrates the corresponding upward trend in the total reward per episode. Figure 4.4 showcases the number of actions the model picks while training with different opponent models, mainly from the same. The graphs shown above are generated in a 1 by 1 training fashion, player takes a turn to train a model, then the enemy trains a new model. Additionally, changing the model the agent is being trained goes against every 5 episodes. Another experiment was conducted where the player and the enemy train simultaneously and have a change of opponent every 3 episodes. The results for the total

loss during training is shown in Figure 4.5 and the results for fluctuation of the reward are shown in Figure 4.6. These final graphs have bigger differences between their intervals as expected in the loss graph. This happens because the agent is constantly learning to adapt to new enemy strategies. Additionally, the reward graph fluctuates a lot more on the second graph however, the reward reaches higher peaks here compared to the first graph. This is to be expected, as the agent learns to deal with new strategies and ways to earn more reward while mitigating risk.

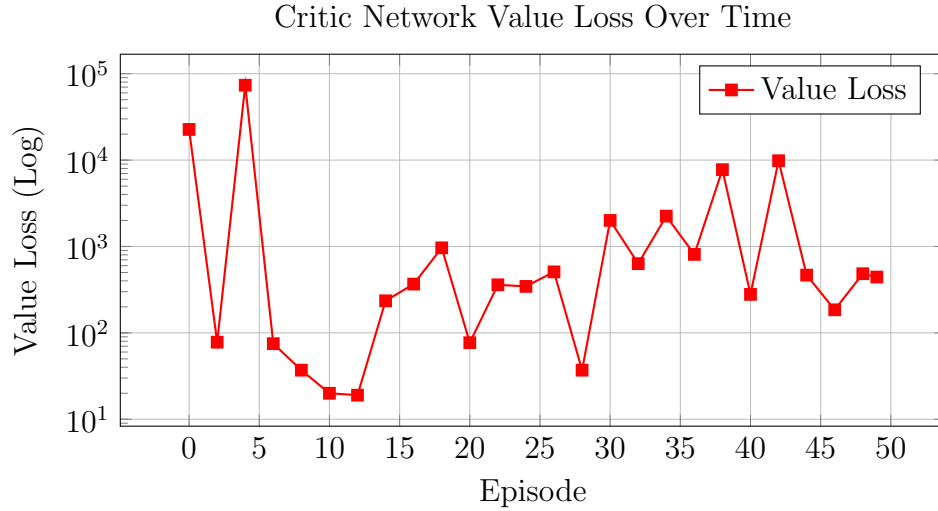


Figure 4.2: Value Loss function in the first 50 episodes.

Achieving the final level of convergence required several mathematical and structural optimizations within the training pipeline. Specifically, implementing normalization for the reward system, the advantage buffer, and the entropy coefficient to stabilize the policy updates and prevent gradient instability.

4.2 Model parameters and performance

Training parameters for PPO were changed constantly throughout the experimentation phase to find the best possible combination. The main goal was to achieve good, stable model behaviour combined with reasonable speed in training, while actively avoiding overly long episodes that would unnecessarily bottleneck the process. Most models were trained with a range of 50 to 200 episodes. The forward

4 Environment Iterations and Algorithm Performance

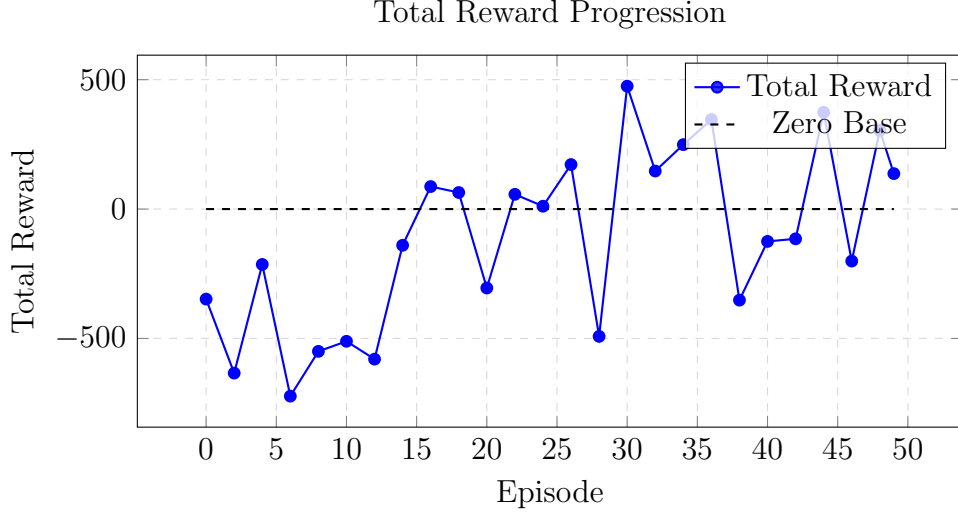


Figure 4.3: Player’s Total Reward over 50 training episodes.

steps were kept in a range of 256 to 2048, with a total of 256 to 10000 steps that can be taken in an episode before forcing the environment to reset. All of these parameter variations and their final values are also represented in Table 4.8. The optimal number of episodes to show meaningful convergence was around 100 to 150. In this setup, the best total step amount turned out to be exactly 5000, with the forward steps sitting right around 1024. Increasing the total steps to more than 5000 did not yield much additional reward or benefit for the model. In the context of this environment, the most interesting and impactful things happen within the first 3000 actions of the game. Adding an additional 2000 steps just to take into consideration the EmptyActions and other explorable actions that the agent needs to figure out without timing out too early. When it came to the optimizer, most of the initial experiments were started with a static learning rate of 10^{-4} . However, a decaying learning rate schedule—which starts relatively high to encourage early exploration and gradually decays down to 10^{-4} worked best for the models. For the clipping parameter was kept at 0.2. This is generally the standard [4] for generalized advantage estimation, and it reliably worked best for the models to ensure the policy updates did not destabilize out of control during training. Finally, multiple gammas across a range of 0.5 to 0.94 were used, and 0.94 was the perfect value for the model. A higher gamma gives much more importance for

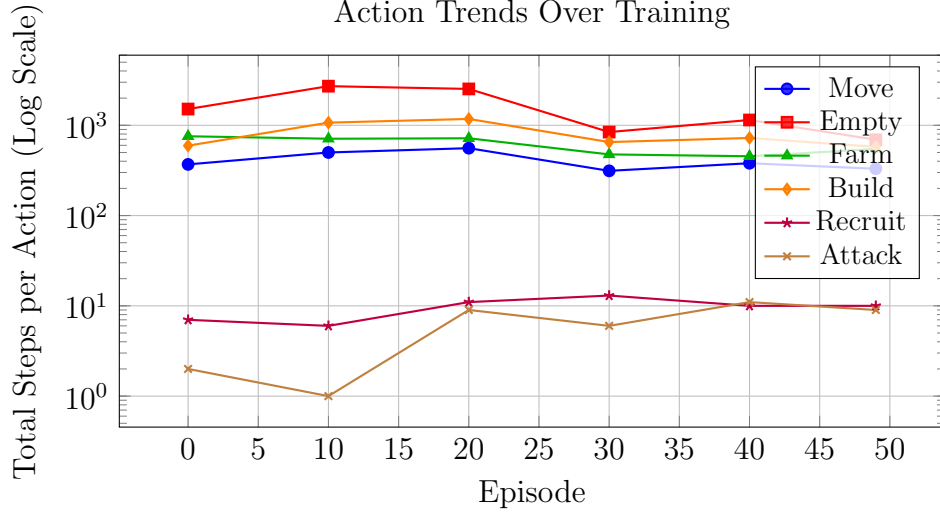


Figure 4.4: Agent action distribution over 50 episodes plotted on a logarithmic scale.

future actions, which makes sense in an RTS game. In this genre, setting up for a larger, delayed reward later is more advantageous than prioritizing an immediate, short-term reward that yields less overall value currently.

Parameter	Value
Episode number	50 to 200
Forward steps	256 to 2024
Total steps	256 to 10000
Learning rate	0.008 (decays)
Gamma	0.94
PPO epsilon	0.2

Table 4.8: Overview of how action space depends on map size

DQN relies heavily on memory buffers—typically to stabilize the learning process and improve sample efficiency. As an agent interacts with an environment, it stores its transitional experiences (the state, action, reward, and next state) into this buffer. During training, the network samples random mini-batches from this memory rather than learning sequentially from consecutive steps. This random sampling is crucial because it breaks the correlations between sequential experiences, which prevents the neural network from getting stuck in local feedback

4 Environment Iterations and Algorithm Performance

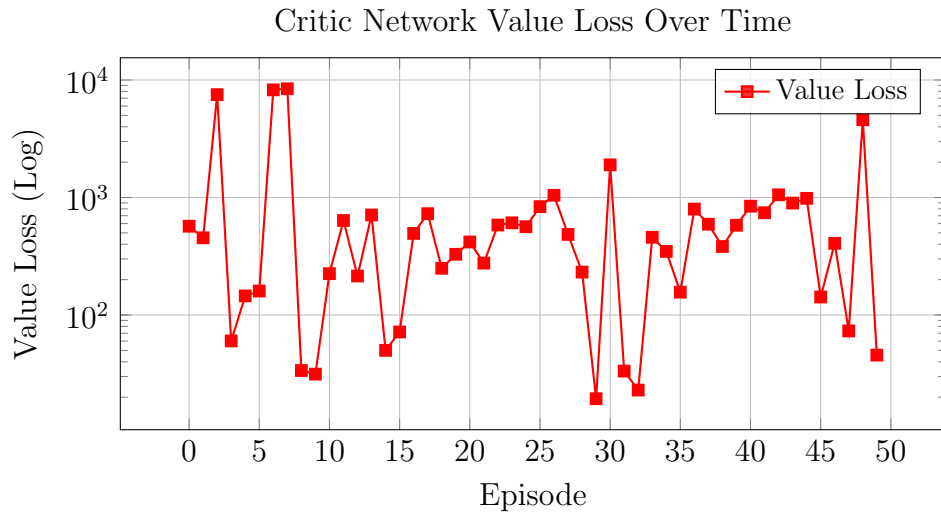


Figure 4.5: Value Loss function over 50 episodes.

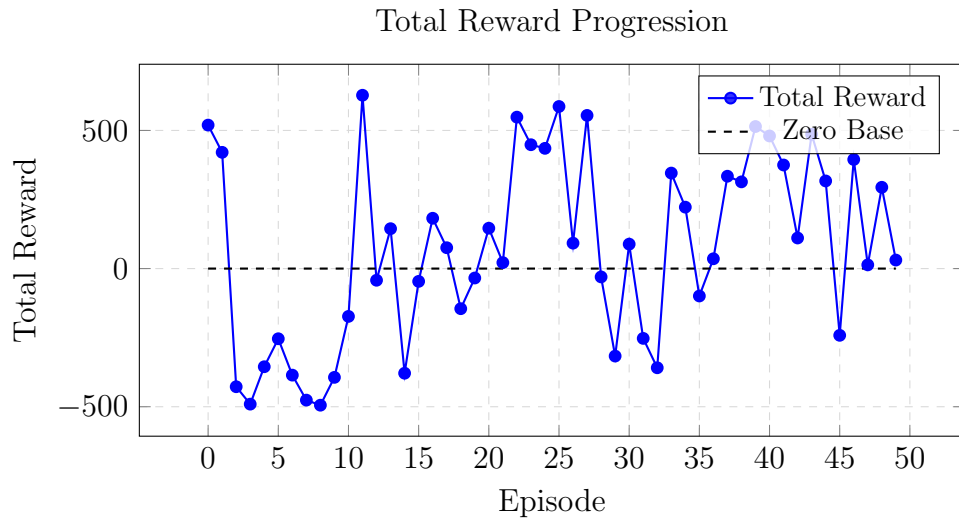


Figure 4.6: Player's Total Reward over 50 training episodes.

4 Environment Iterations and Algorithm Performance

loops. That is why some optimizations are needed for the training process by making the state replay faster for saving, loading and any operation that is needed during training.

Experiments of the replay system were done using randomly generated games with the same size, so the difference in game would not lead to a difference in time to save and load between the different formats.

The test results presented in Table 4.9 and Table 4.10 show that binary serialization is a much faster alternative to a string-based approach. For a buffer of 100,000 actions, the binary method was, on average, 5.2 times faster for saving data and 1.5 times faster for loading it.

While the performance benefits continued with a larger dataset of 200,000 actions, a higher variance was found here. In each run, the string-based test was performed before the binary test, which likely left the memory fragmented. This fragmentation could have impacted the performance of the subsequent binary serialization, leading to the observed variance.

There is also a clear path for making the binary method even faster. The current implementation uses 8 byte primitives for numbers, switching to 5 byte types would cut the data size by around 40% and lead to better I/O performance. This can be done by implementing a union-like variant that initializes enough memory for 4 bytes (either an integer or a float) and one additional byte that represents 0 for an integer and 1 for a float. Given these results and the potential for further improvement, it is clear that a binary serialization protocol and format is the correct choice for the replay system. It significantly reduces latency, which is critical for a system that needs to handle high-throughput I/O.

Training DQN followed a very similar fashion to PPO, and almost all of the hyperparameters and constants used for the PPO model followed suit for the DQN algorithm as well, which are shown in Table 4.11.

Below is a Figure 4.9 of the reward progression for DQN, which showcases the inherent instability [38] of DQN by nature. Because the agent relies heavily on random exploration early on, the rewards swing wildly from massive negatives to huge positive spikes before the Q-values begin to stabilize. Generally speaking, DQN had noticeably faster training speeds than PPO because there simply was not as much complex computation going on under the hood during the action

4 Environment Iterations and Algorithm Performance

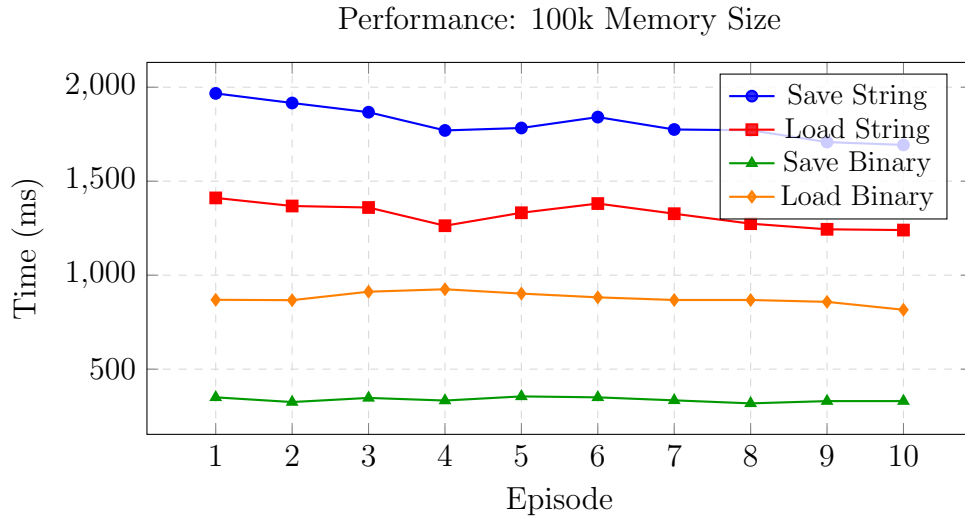


Figure 4.7: System Performance with 100k Memory Size.

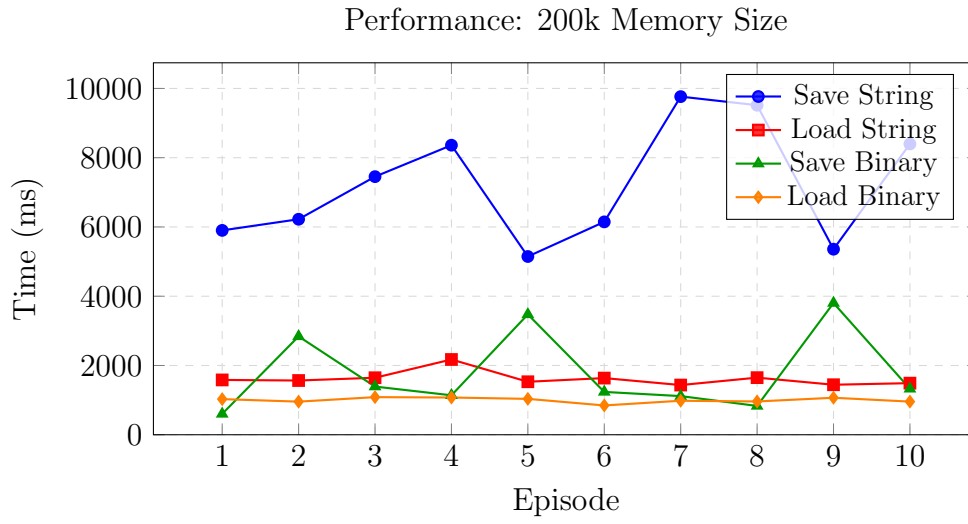


Figure 4.8: System performance metrics for 200k memory size.

4 Environment Iterations and Algorithm Performance

Save string	Load string	Save binary	Load binary
1967	1411	350	869
1916	1368	325	867
1867	1360	347	912
1770	1263	333	925
1783	1332	355	902
1841	1381	350	882
1775	1327	334	868
1771	1274	318	868
1708	1244	330	858
1693	1240	330	816

Table 4.9: Time taken for 100.000 actions

Save string	Load string	Save binary	Load binary
5901	1583	602	1029
6223	1566	2840	956
7453	1646	1391	1086
8361	2173	1139	1075
5147	1530	3475	1036
6147	1636	1237	846
9764	1435	1116	982
9516	1649	832	961
5358	1444	3804	1068
8397	1490	1327	957

Table 4.10: Time taken for 200.000 actions

Parameter	Value
Total steps	5000
Learning rate	3e-4 (static)
Gamma	0.98
Epsilon	1.0 (decays to ~ 0.05)
Batch size	512
Replay memory size	10000
Tau (Target update)	0.005

Table 4.11: Overview of DQN training parameters

selection phase.

The biggest speed-up achieved during this implementation was because of a full

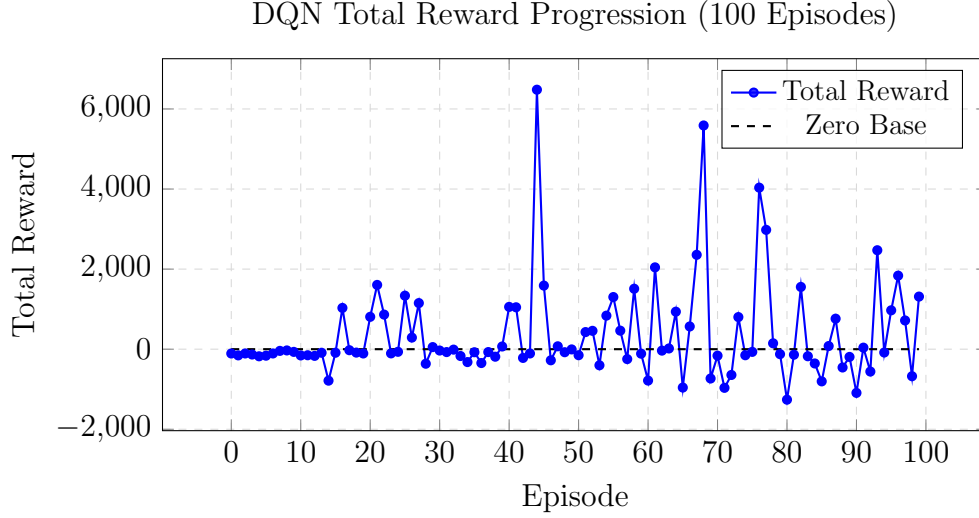


Figure 4.9: Player’s Total Reward Progression over 100 DQN training episodes.

structural transformation to the experience replay buffer. Using raw pointers to represent a transition state for the experience of DQN was simply too error-prone and generally very slow, as it relied on the iterative insertion of vectors of considerable size. Every time the agent took a step, memory management became a bottleneck. Instead, this architecture was changed by saving the experience states directly as flattened tensors. With the state tensor, standard integers representing the action index picked by the agent were also stored, the float reward of the state, and a bool indicating the state termination flag. By skipping the pointer conversions and keeping the data strictly in tensor formats, batch sampling for the neural network updates became faster [39]. This could have technically been optimized even further, but the only place that would benefit from more optimization would be the memory loading phase (the file size produced by DQN held around 10.000 states and the file itself was around 80MB).

4.3 Algorithm performance

To actually evaluate how these models perform after the training loop, the PPO agent against itself in a 50-game evaluation match. In theory, putting the exact same algorithm against itself should result in a 50/50 win rate if the environment

4 *Environment Iterations and Algorithm Performance*

is perfectly balanced. The final result came out to a 42% win rate for the player and a 58% win rate for the enemy. While this is not exactly a perfect 50/50 split, it makes a lot of sense when looking at the environment mechanics. First, the spawn points are not mathematically perfectly symmetrical (the player starts at 2,2 while the enemy starts at MAP_SIZE-4, MAP_SIZE-4), meaning pathing to resources is not identical. Fortunately, this was a planned result and placing the enemy in such a location has a very clear explanation on why the results are biased towards the enemy. The spawn point is simply closer to the gold mines and closer to the middle which gives the enemy more room around the map compared to the player. On top of that, since the enemy agent evaluates the state right after the player in the code loop, it gets a slight reaction advantage. Overall though, getting this close to an even split shows the environment is highly stable and neither side has a severe map advantage.

To investigate the 42% to 58% discrepancy observed in the baseline evaluation, the sequential nature of the environment loop was the focus. Initially, the code strictly evaluated the player’s action first, giving the enemy agent a slight tactical advantage by allowing it to react to the player’s committed state on every single game tick.

To eliminate this first-mover bias, the action evaluation loop was updated to use C++’s random number generator, randomly dictating whether the player or the enemy would take the first action on any given tick. The map spawn points remained unaltered at their original coordinates (player at 2,2 and enemy at MAP_SIZE-4, MAP_SIZE-4).

Running the exact same PPO models through this randomized turn-order environment over another 50 games resulted in a 44% win rate for the player and a 56% win rate for the enemy. This slight tightening of the win rate toward a 50/50 equilibrium indicates that the predictable turn order was indeed having a minor impact on the outcomes. However, the persistent overall advantage for the enemy confirms that the slight mathematical asymmetry in the starting coordinates continues to heavily influence the environment. The remaining 12% discrepancy is a direct reflection of this map bias combined with the fact that 50 game iterations is too low for the law of large numbers [40] to smooth out the variance.

To evaluate the performance and stability of the fully trained DQN models,

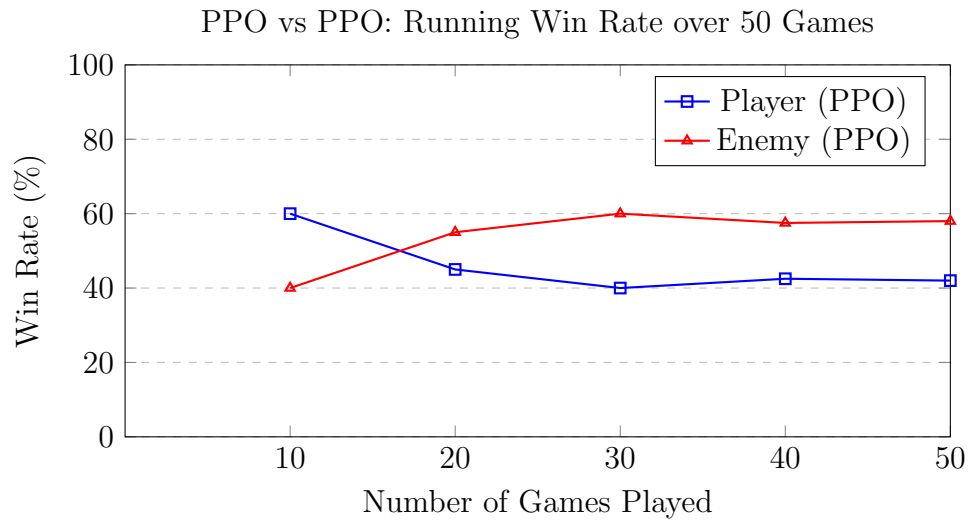


Figure 4.10: The running win rate percentage of the Player and Enemy PPO agents over a 50-game span.

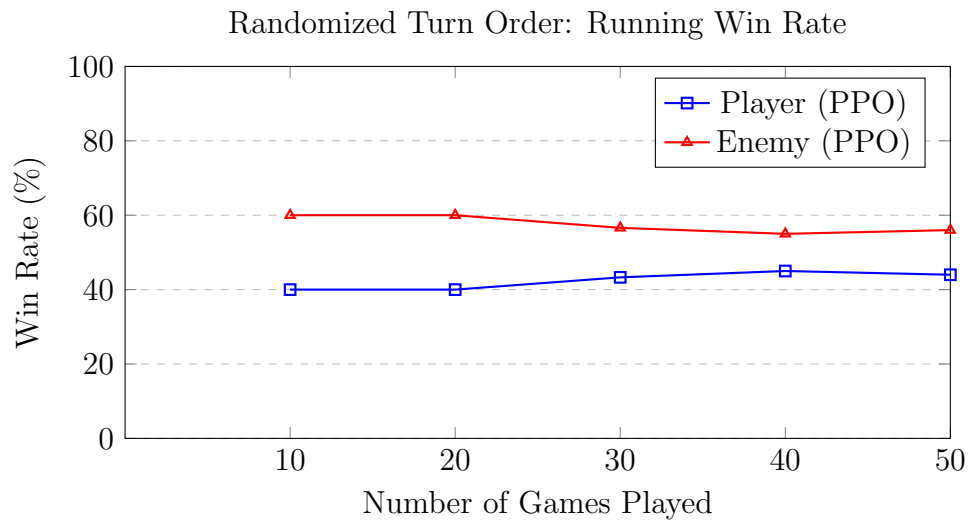


Figure 4.11: The running win rate of the PPO agents after randomizing the turn order.

4 Environment Iterations and Algorithm Performance

the exact same 50-game baseline evaluation that was used for PPO was used. Putting the player’s DQN agent against the enemy’s DQN agent in the standard environment, retaining the exact same asymmetrical spawn points (player at 2,2 and enemy at MAP_SIZE-4, MAP_SIZE-4) and the sequential action evaluation loop.

The final result for the DQN evaluation came out to a perfect 50% win rate for the player and a 50% win rate for the enemy (exactly 25 wins each). This was likely statistical variance, as after running it multiple other times it was not getting a perfect 50% win rate. As discovered during the PPO evaluation, the environment has an inherent mathematical bias towards the enemy due to the spawn coordinates being closer to the gold mines and the center of the map. The reason for this is was likely because the enemy was trained last and for fewer episodes than the player.

The Figure 4.13 illustrates the running win rate at 10-game intervals. While the enemy took an early lead in the first 20 games (mirroring the positional advantage seen in PPO), the player agent completely closed the gap as the match count increased, perfectly stabilizing at a 50/50 equilibrium by game 50. The order of actions is not arbitrary but instead the player goes first and the enemy second.

To further examine how turn order affects off-policy agents, the exact same DQN models were experimented with randomized turn-order over another 50 games, Figure 4.12. This resulted in a 60% win rate for the player and a 40% win rate for the enemy. As suggested, the enemy overfit to the training because it moved second however randomizing the action execution order entirely disrupted this deterministic loop. Without the guarantee of reacting strictly after the player’s committed state on every tick, the enemy’s highly rigid strategy broke down more frequently, artificially inflating the player’s win rate to 60%. This effectively highlights how brittle deterministic algorithms like DQN can be when subtle environmental sequences are altered post-training, contrasting heavily with the smoother distribution seen in PPO.

To definitively compare the two algorithms, a 50-game evaluation pitting the fully trained Player PPO agent against the Enemy DQN agent was necessary, Figure 4.15. The environment retained the exact same asymmetrical spawn points (which inherently favored the enemy).

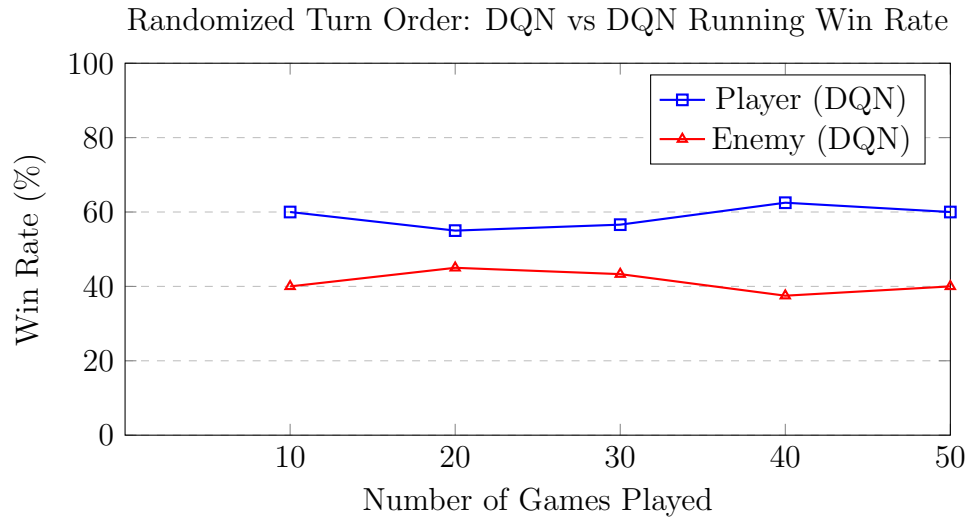


Figure 4.12: The running win rate of the DQN agents after randomizing the turn order.

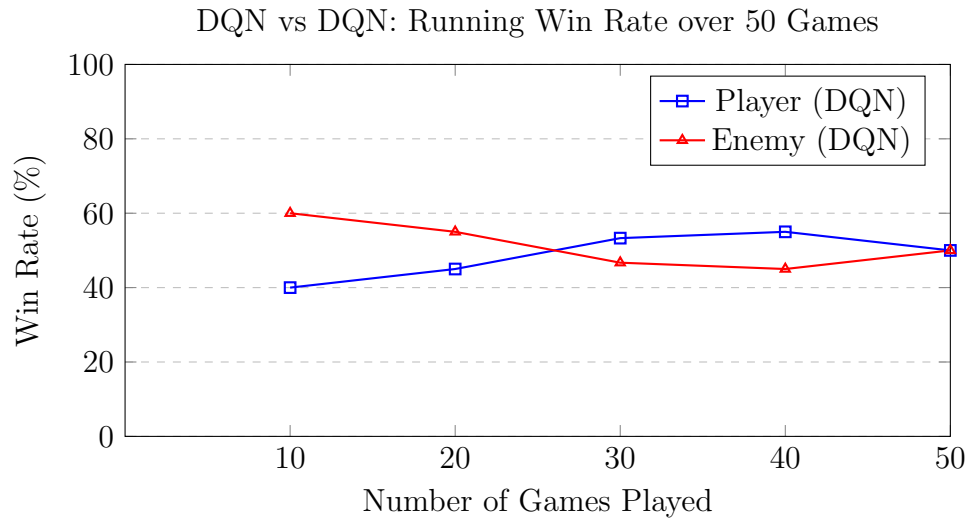


Figure 4.13: The running win rate percentage of the Player and Enemy DQN agents over a 50-game span.

4 *Environment Iterations and Algorithm Performance*

The results were completely one-sided: the PPO agent achieved a massive 96% win rate (48 wins), while the DQN agent secured only a 4% win rate (2 wins). This absolute dominance effectively proves the algorithmic superiority of PPO in this highly complex, multi-parameter RTS environment. While DQN struggled with brittleness and deterministic rigidity, PPO’s stochastic policy and clipped surrogate objective allowed it to remain highly adaptable. The PPO agent was able to dynamically navigate the environment, build a sustainable economy, and execute military strategies that the DQN agent simply could not counter. Crucially, the PPO agent performed so well that it completely nullified and overwhelmed the inherent positional map advantage that the enemy spawn point provided in earlier baselines. This confirms that for massive state-action spaces with delayed rewards, policy gradient methods with entropy exploration significantly outperform traditional value-based off-policy methods.

To eliminate any possibility that PPO’s absolute dominance was merely the result of the first-mover advantage, a subsequent 50-game evaluation experiment was conducted, using the randomized turn-order execution loop, Figure 4.14. The environment and the asymmetrical spawn points remained identical to the previous tests.

The introduction of randomized action execution only served to solidify PPO’s superiority, resulting in a 98% win rate (49 wins) for the player and a mere 2% win rate (1 win) for the DQN enemy. As observed in the DQN versus DQN baseline, randomizing the turn order does not help the deterministic, rigid strategies that off-policy value-based methods rely upon post-training. While the DQN agent struggled to adapt to the unpredictable tick-by-tick execution delays, the PPO agent’s stochastic, policy-driven architecture absorbed the randomness effortlessly. The PPO agent consistently executed its macroeconomic scale-up and tactical combat operations regardless of whether it acted first or second on any given frame. This near-perfect win rate definitively concludes that PPO is vastly more robust, adaptable, and capable of operating under highly variable environmental constraints than DQN in this specific RTS framework.

A significant challenge within reinforcement learning and other computationally intensive fields is the difficulty of ensuring experimental reproducibility [41]. Minor variations in software dependencies, system libraries, or hardware configurations

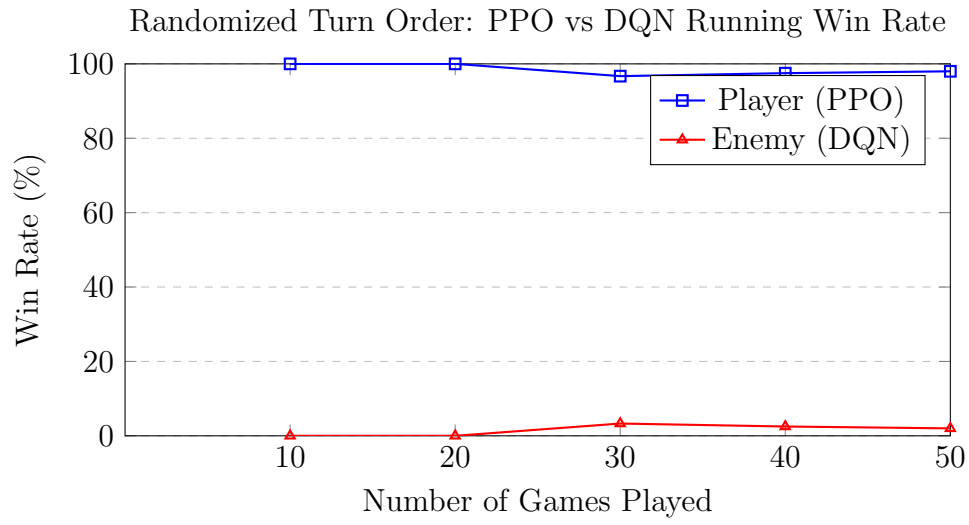


Figure 4.14: The running win rate of PPO versus DQN after randomizing the turn order.

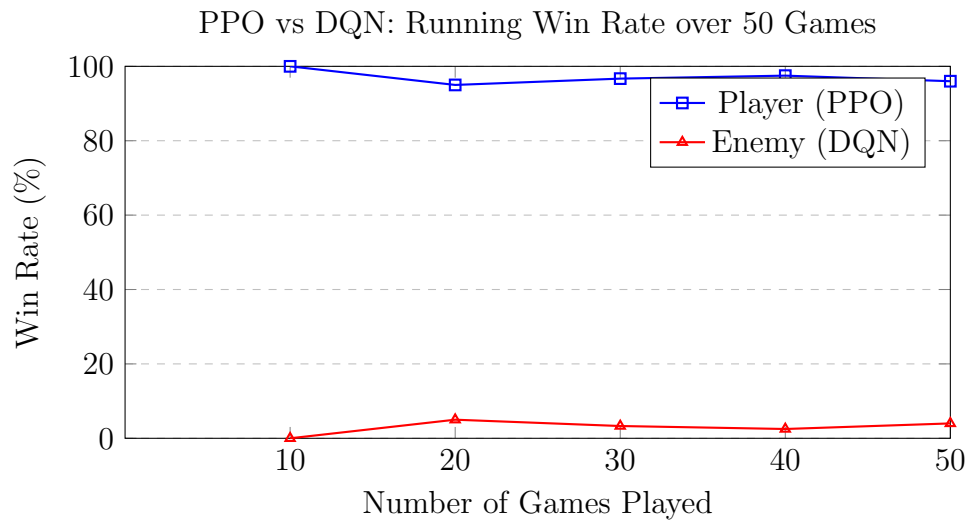


Figure 4.15: The running win rate of the Player (PPO) versus the Enemy (DQN).

can lead to divergent outcomes, hindering the validation and extension of prior work. This thesis addresses this challenge directly by establishing a robust and transparent methodology for reproducing all presented results. The primary mechanism for achieving this is the Nix package manager and its associated ecosystem [42]. For broader accessibility, a secondary, script-based approach is also provided. All source code, configuration files, and documentation are publicly available in the project repository [43].

Reproducibility in this project is primarily achieved through a declarative environment specified using Nix. Nix enforces a purely functional approach to package management, meaning that a given build specification will produce an identical environment regardless of the host machine’s state. This is accomplished using flakes, which are self-contained files that define all project inputs and outputs.

The flake file for this project pins specific versions of all dependencies (inputs) and declares a set of outputs, which include the compiled artifacts, development environments, and execution scripts. This ensures that any interested person can instantiate the exact software environment used for the original experiments.

The core dependencies required to build and run the project are detailed in Table 4.12.

Table 4.12: Project Dependencies

Dependency	Description
Perl	Scripting language for FlameGraph tool
pkg-config	Utility for locating library paths
SDL3 & -TTF	Graphical interface and font rendering
Catch2	C++ unit testing framework
CMake	Build system generator
LibTorch	Tensor and neural network framework
Ninja	Build execution engine
FlameGraph	Performance profiling visualization

The entire system can be built and verified by executing the command `nix build` within the project’s root directory.

5 Evaluation

To help with evaluation of the questions and the whole thesis the players and observers of the game were presented with a UI which allowed them to see the best possible actions currently in the game. Most deep RL agents are basically black boxes: they produce actions from high-dimensional inputs through millions of parameters, giving no real justification for their choices. For some papers this is not necessary, but having the choice or having the knowledge of best possible action in a state is important. In competitive games specifically this is a halt in creativity as the person learning only learns one specific way to deal with a hurdle throughout the game instead of understanding what other actions are possible to counteract what is going on in the game.

The approach taken in this thesis fixes this by making the agent’s raw output layer apparent or rather the probability distribution $\pi_{\theta}(a|s)$ over all valid actions at every state of the game, directly in the GUI shown in Figure 5.5 using SDL3. For example this GUI shows in a list the probability that the agent assigns to picking a specific action out of all possible actions. This exposes how confident the agent is in preferring each available action, shown in the GUI given the current state [44]. The hypothesis is that a player watching these distributions shift in real time can build an intuition for which game states the agent considers decisive, which it treats as equivalent, and which action it consistently prioritizes. Each algorithm has its actions shown on the right side of the UI, with the environment being shown on the left side of the statistics of the actions and the player statistics such as gold, food and units alive below the actions menu.

A significant constraint during this evaluation was securing a larger sample size. Distributing the custom game environment and its accompanying neural network models—which collectively exceed one gigabyte to individual participants’ personal computers was unfeasible due to file size and complex software dependencies. Con-

5 Evaluation

sequently, all evaluation sessions were conducted in person on a dedicated local workstation in Vienna.

The participant group consisted of individuals of different levels. Some players had previously played games such as Starcraft II, League of Legends and Dota 2, some of them had seen and heard of RTS and some of them had no prior knowledge of the game genre itself. This split in results could potentially be also explained by the skill difference in the participants as the more experienced players in this genre would learn very minimally without actually spending hours on the game. The less experienced players had a very quick realization that something interesting they could learn from was happening and they were more interested in observing the agents performance and upcoming actions to take. This suggests that the tool's effectiveness as a teaching instrument may scale with the observer's baseline game knowledge, and that future evaluations would benefit from stratifying participants by experience level to isolate this effect.

Participants were presented with a four-choice questionnaire following the observation session. The Google Forms, scale was selected because it captures graded agreement or disagreements rather than yes/no responses, which is more appropriate for subjective learning assessments. Each question was answered on a scale of strongly disagree, disagree, agree, and strongly agree. The option for a neutral feeling was removed intentionally, as this is the most selected options in case a participant does not understand a question or would rather not give an answer. This option also would not be sufficient since the sample size of this thesis is so small too. The sessions themselves were minimally guided: participants were explained how the game functioned, what the game was about, what the GUI choices represented, what the food and gold metrics indicated and lastly what each letter and color in the map meant. After this participants observed a live match between the PPO and DQN agents using the GUI, with the real-time action probability distributions visible throughout. The game was slowed down with a thread sleep so it was possible to understand what was going on in the game. Additionally, another gameplay was given where the players could choose based on a numeric value 0-9 what action the algorithm could take to play the game. Nothing internal on the agent's side was explained and the only goal was minimal learning of some small strategies or orders of gameplay.

Hypothesis 1 was answered based on the answers received from the Google Forms:

- **I was able to learn new build orders or unit combinations by observing the actions of the AI.**

Evaluation: The results for this question leaned negative. Out of the 7 participants, 4 (57.1%) disagreed that the AI taught them new build orders or unit combinations. Only 3 participants agreed or strongly agreed. This suggests that while the AI plays well, its specific unit combinations or the order in which it builds structures might be too rigid or highly optimized for its own specific reward function, making them difficult to apply to a standard human playstyle. Build order is hard to be taught in a game that does not have a lot of players, that is why this result was received in this hypothesis. Another reason could be that the agents are usually picking strategies that counter the other agent so the build order is not as visible in these scenarios, as everything that is happening is a counter to the opponent's move.

- **I was able to learn new strategies by observing the AI play.**

Evaluation: In contrast to the first question, the feedback here was largely positive. A total of 5 out of 7 players felt they learned new strategies from watching the agent. These strategies mainly were related to early-game strategies or slightly attacking the enemy and zones where to place structures. Generally this was a positive evaluation as noted above, the algorithms were counter reacting to each others' actions in specific states. This is exactly what the executable is trying to achieve, teaching players a way to counter react to bizarre plays.

- **The replay system was a valuable tool for reviewing and improving my strategic gameplay decisions.**

Evaluation: This statement received the strongest positive response from the survey. 5 out of the 7 participants found the replay system valuable, with a majority of 4 (57.1%) strongly agreeing. This highlights that having a visual, step-by-step tool to review past games is a highly effective way for players to analyze their mistakes and understand exactly how the AI managed to gain an advantage over them.

- **The AI exposed me to defensive or offensive maneuvers I had not considered before.**

Evaluation: The responses here were more divided but still leaned slightly positive. 4 participants (57.1%) agreed that the AI showed them novel defensive or offensive maneuvers not considered before, while the remaining 3 (42.9%) disagreed. This split makes sense, as some players have already seen these moves if they are experienced and some players would not understand them.

This correlates highly to the second hypothesis as the maneuvers presented by the agents were of high variance. This is one of the main points of the graphical user interface and the thesis in general.

To determine the statistical significance of these survey responses, a two-tailed Binomial test was conducted for each question, comparing the response distributions against a random chance baseline ($p=0.5$). The resulting p-values were 1.000, 0.453, 0.453, and 1.000, respectively. None of the results achieved the standard threshold for statistical significance ($p \leq 0.05$). However, this lack of significance is a direct mathematical limitation of the small sample size. Achieving a statistically powerful sample size was logistically unfeasible due to the inherent challenges of distributing custom reinforcement learning environments. Providing the software to a wider audience would require bundling the compiled binaries with multiple neural network models exceeding 400 MB in size. Furthermore, configuring the necessary system dependencies would be an issue. Consequently, while the qualitative feedback indicates strong positive trends regarding the tool’s educational value, the findings should be treated as preliminary.

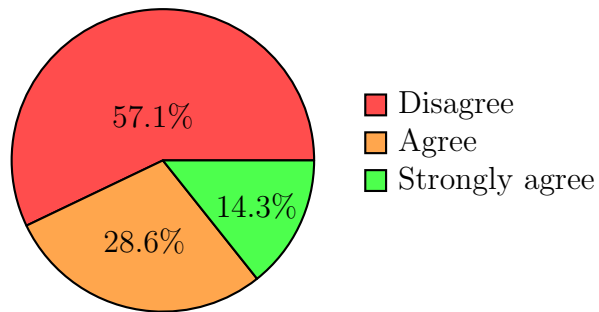


Figure 5.1: Responses evaluating if players were able to learn new build orders or unit combinations ($N = 7$).

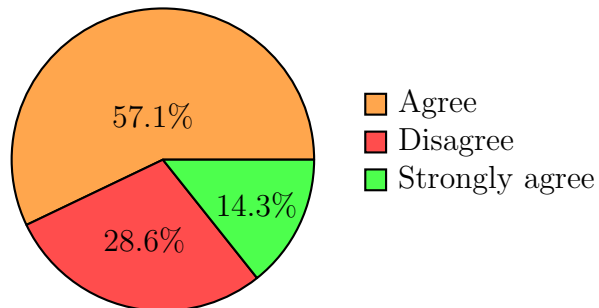


Figure 5.2: Responses evaluating if players were able to learn new strategies by observing the AI play ($N = 7$).

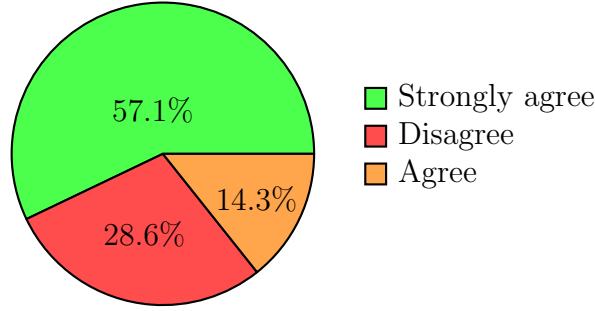


Figure 5.3: Responses evaluating if the replay system was a valuable tool for re-viewing gameplay ($N = 7$).

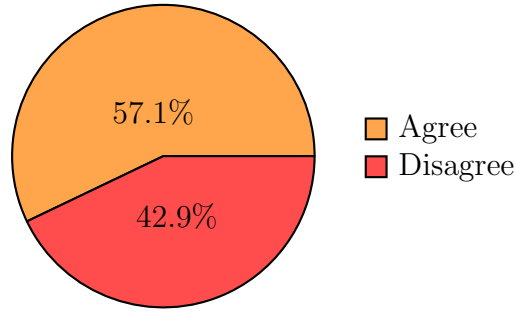


Figure 5.4: Responses evaluating if the AI exposed players to new defensive or offensive maneuvers ($N = 7$).

For the evaluations below, the same setup of variables, learning rates, decays, episode numbers, and forward steps was used. Hypothesis 2:

- **Performance of PPO vs. PPO**

This experiment yielded the most nuanced results regarding environmental balance. Putting the same PPO architecture against itself was thought to result in a 50%-50% win rate. However, the evaluation showed a 42% win rate for the player and a 58% win rate for the enemy for the non randomized turn order experiment. While with the addition of randomization the win rate came closer to the expected win rate, with a change to 44% win

5 Evaluation

rate for the player and 56% win rate for the enemy. This persistent discrepancy leads to the conclusion that in high-dimensional RTS environments, subtle mathematical asymmetries influence outcomes. The enemy’s spawn point was strategically closer to the central gold mines, providing a resource-gathering advantage that the agent successfully exploited. This confirms that map balance is just as critical as algorithmic tuning in competitive RL environments.

- **Performance of DQN vs. DQN**

While the initial sequential evaluation of DQN vs. DQN yielded a 50%/50% win rate, the randomized experiment revealed the underlying instability of the model. Upon randomizing the action execution order, the win rate shifted drastically to 60%/40% in favor of the player. This suggests that the DQN agent had overfit to the deterministic nature of the environment’s internal code loop (the sequential Player then Enemy logic). Because DQN is a rigid, value-based method, its strategy became brittle when the predictable tick-by-tick timing was disrupted. This highlights a fundamental weakness in applying standard DQN to real-time environments where action latency and order can fluctuate.

- **Performance of PPO vs. DQN**

This result confirmed the primary hypothesis regarding algorithmic superiority. The PPO agent achieved a dominant 96% to 98% win rate against the DQN opponent. The success of PPO can be attributed to its stochastic policy and the use of an entropy bonus, which allowed it to navigate the massive action space more effectively than the value-based DQN. PPO demonstrated a superior ability to manage delayed rewards, such as transitioning from early-game resource gathering to late-game military production. The PPO agent’s strategy was robust enough to completely overcome the positional map disadvantage that had hindered it in the PPO vs. PPO trials, proving that policy-gradient methods are significantly better suited for complex, multi-parameter RTS frameworks.

The below image represents what the environment looks like while training or while in play (the image is taken during training). Generally, this GUI is not shown

5 Evaluation

during training in order to save CPU cycles and speed up training, although it is helpful during the debugging process.

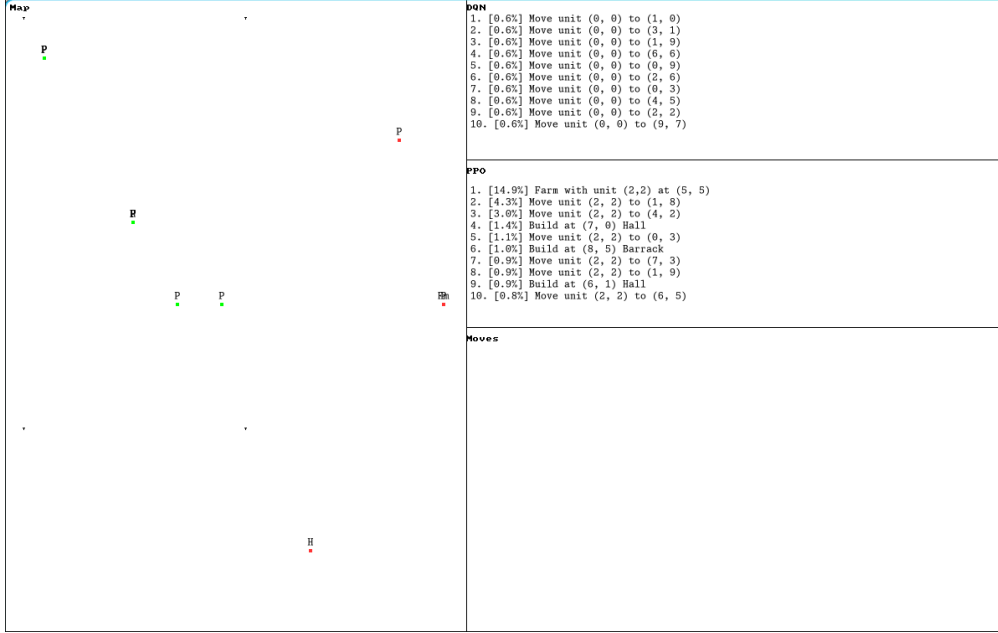


Figure 5.5: Overview of the RL agent’s gameplay and training environment interface.

As stated in previous chapters, the agents use league-play and additionally to have a tool that is accessible for future research, a way to save replays (played games) was necessary. The empirical data confirms that adopting a binary serialization protocol is a necessary optimization for minimizing execution latency in high-throughput reinforcement learning pipelines. For a memory buffer containing 100,000 actions, the binary serialization method demonstrated an average 5.2 times speedup in saving operations and a 1.5 times improvement in loading times compared to the string-based baseline. Scaling the data with which this system was tested, experiments showed that the performance increase continues although the variance also did. The main reason why this was happening was likely heap fragmentation, at least for the string replay system.

Significant computational resources were dedicated to optimizing the training pipelines. During initial testing, DQN completed episodes in under a minute using solely CPU execution, while PPO required slightly more time due to its actor-critic

5 Evaluation

architecture and gradient calculations. Across all training loops, a high discount factor ($\gamma \in [0.9, 0.95]$) was selected. As highlighted in established literature, RTS environments value delayed rewards over immediate gains, necessitating a horizon that accounts for long-term economic scaling. For the DQN architecture specifically, the experience replay buffer was tuned to store 10,000 state transitions. This threshold was selected via testing to ensure sufficient diversity for the value updates while strictly preventing heap fragmentation during local execution.

The computational workload for training the reinforcement learning models and evaluating the agent’s performance was distributed across two distinct hardware environments. The majority of the localized testing, environment debugging, and initial CPU-bound training loops were executed on a local workstation running Arch Linux. The hardware specifications for this local machine are as follows:

- **Processor (CPU):** AMD Ryzen 7 PRO 5850U with Radeon Graphics (8 Cores, 16 Threads, boosting up to 4.5 GHz)
- **System Memory:** 16 GB RAM
- **Operating System:** NixOS for reproducibility, Arch Linux for easy development
- **Operating System:** Ubuntu for the Spark-enabled NVIDIA GPU.

To accelerate the more computationally expensive batch-training phases, a portion of the workload was offloaded to a remote Spark-enabled NVIDIA GPU node. This hybrid approach allowed for rapid, iterative testing within the local engine while leveraging dedicated GPU hardware to expedite the neural network’s convergence during extended training runs.

Some limitations that exist in the thesis are the lack of field of view introduced in [5]. Generally speaking, the models have global knowledge about everything in the game and this would not specifically be something that a real human player would have at all times. However, this does not affect the conclusions of the thesis as both agents had this information available to them. Moreover, the paper by OpenAI and Berner et al. [15], suggests that the limitation of FOG is mainly integrated as a way to equalize the human limitations of reaction time and inability to see

5 *Evaluation*

what's outside their FOG. Some features in a general RTS game are lacking, such as very good game design choices such as good economy model for buildings and units. This does affect the strategy a model thinks is best for the environment, however, this will not affect the performance that the agents have against each other as they both suffer from this limitation.

The biggest difficulties presented in the thesis were generally noted in other chapters as well, however the biggest hurdle in this thesis was definitely finding a suitable reward shape that would make the agents not collapse into a loop or a local optimum.

6 Conclusions

Extensive experimentation was conducted throughout this thesis, giving a mix of highly successful architectural optimizations and some unexpected system-level trade-offs. Initially, a massive, strictly positive speed-up was anticipated when migrating the DQN experience replay from a string-based memory format to a binary representation. While the raw I/O serialization speed did increase significantly, the density of the flattened binary tensors introduced unintended heap fragmentation, occasionally bottlenecking the kernel during continuous allocations. This highlighted a lesson: in high-throughput reinforcement learning, raw read/write speeds can be easily negated by structural fragmentation.

The direct algorithmic comparison between PPO and DQN yielded results that largely aligned with modern RL literature, though observing them in a custom RTS environment provided unique insights. As expected, PPO required significantly more computation and time to train, but it offered a much wider array of hyperparameter optimizations. Because PPO is constrained by its clipped surrogate objective, the network was naturally resistant to sudden network shocks or catastrophic policy collapse. However, this same mathematical complexity made PPO far more adept at identifying and exploiting degenerate reward loops—such as pacifist movements or infinite farming—compared to the more rigid, deterministic DQN.

Consequently, overcoming these local optima became the most time-consuming phase of the research. Designing a reward system that perfectly balanced aggressive and passive behaviors without pushing the agent into an infinite loop or an instant-death rush required constant and iterative tweaking. The reward system served as the behavioral compass for the agents, but it was the implementation of global action masking that provided the most substantial performance boost. By systematically reducing the probability of the illegal moves from the probability

6 Conclusions

distribution, action masking shaved valuable time off the training process and allowed the agents to focus entirely on strategic learning rather than basic ruleset validation.

To further support the agents in the late stages of training, especially as the learning rate decayed, entropy in the probability of the available actions was integrated. While this did not yield a massive boost in raw win rate, it functioned as a vital safety net, injecting just enough mathematical shock to prevent the policy from stagnating into repetitive strategy loops.

Through repeated environment testing, it became clear that the single most impactful variable on training scalability was the map size. Capping the final experiments at a 10 by 10 grid struck the ideal balance—providing a state-action space complex enough to evaluate sophisticated RTS strategies, without making the computational burden prohibitively expensive for the agents to learn.

Beyond the specific algorithmic comparisons, a primary contribution of this thesis is the engineering of a fully reproducible, custom-built C++ real-time strategy environment. Reinforcement learning research is frequently hindered by closed-source commercial game engines or highly fragile dependency chains that make independent verification nearly impossible. By leveraging a declarative package management system alongside a lightweight LibTorch backend, this project resolves that bottleneck. It establishes a transparent, modular, and highly manipulable framework that guarantees exact reproducibility across different hardware setups. Ultimately, this work provides not just an analysis of PPO and DQN, but a robust, open-source foundation that future researchers can seamlessly adopt, modify, and extend whenever a custom, high-performance training environment is required.

Finally, feedback gathered from a sample of users testing the tool was generally positive. The primary concern raised was the clarity of the GUI, which warrants future refinement to be more intuitive. However, testers noted that the agents exhibited creative strategies, particularly in early-game aggression and late-game building placement. A recurring tactic involved continuous harassment with a small number of units, suggesting that the agent had successfully learned to prioritize the disruption of the opponent’s economy over sheer military dominance.

Bibliography

- [1] Oriol Vinyals, Igor Babuschkin, Wojciech M. Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H. Choi, Richard Powell, Timo Ewalds, Petko Georgiev, Junhyuk Oh, Dan Horgan, Manuel Kroiss, Ivo Danihelka, Aja Huang, Laurent Sifre, Trevor Cai, John P. Agapiou, Max Jaderberg, Alexander S. Vezhnevets, Rémi Leblond, Tobias Pohlen, Valentin Dalibard, David Budden, Yury Sulsky, James Molloy, Tom L. Paine, Caglar Gulcehre, Ziyu Wang, Tobias Pfaff, Yuhuai Wu, Roman Ring, Dani Yogatama, Dario Wünsch, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy Lillicrap, Koray Kavukcuoglu, Demis Hassabis, Chris Apps, and David Silver. Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, November 2019.
- [2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.
- [3] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, January 2016.
- [4] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Bibliography

- [5] Michaël Mathieu, Sherjil Ozair, Srivatsan Srinivasan, Caglar Gulcehre, Shang-tong Zhang, Ray Jiang, Tom Le Paine, Richard Powell, Konrad Żołna, Julian Schrittwieser, David Choi, Petko Georgiev, Daniel Toyama, Aja Huang, Roman Ring, Igor Babuschkin, Timo Ewalds, Mahyar Bordbar, Sarah Henderson, Sergio Gómez Colmenarejo, Aäron van den Oord, Wojciech Marian Czarnecki, Nando de Freitas, and Oriol Vinyals. AlphaStar Unplugged: Large-Scale Offline Reinforcement Learning, August 2023.
- [6] Yanyan Li, Yijun Wang, and Yiwei Zhou. Multiagent Deep Reinforcement Learning Algorithms in StarCraft II: A Review. *IEEE Access*, 12:167452–167470, November 2024.
- [7] Richard S. Sutton and Andrew Barto. *Reinforcement Learning: An Introduction*. Adaptive Computation and Machine Learning. The MIT Press, Cambridge, Massachusetts London, England, second edition edition, 2020.
- [8] Amirkeivan Mohtashami, Martin Jaggi, and Sebastian Stich. Special Properties of Gradient Descent with Large Learning Rates, February 2023.
- [9] Andrew Y. Ng, Daishi Harada, and Stuart Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *Proceedings of the Sixteenth International Conference on Machine Learning*, ICML '99, pages 278–287. Morgan Kaufmann Publishers Inc., 1999.
- [10] Shengyi Huang and Santiago Ontañón. Action Guidance: Getting the Best of Sparse Rewards and Shaped Rewards for Real-time Strategy Games. *arXiv*, October 2020.
- [11] Dario Amodei, Chris Olah, Jacob Steinhardt, Paul Christiano, John Schulman, and Dan Mané. Concrete Problems in AI Safety, July 2016.
- [12] Sam Devlin and Daniel Kudenko. Dynamic potential-based reward shaping. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 1*, pages 433–440, New York, New York, June 2012.

Bibliography

- [13] Shengyi Huang and Santiago Ontañón. A Closer Look at Invalid Action Masking in Policy Gradient Algorithms. *The International FLAIRS Conference Proceedings*, 35, May 2022.
- [14] Anssi Kanervisto, Christian Scheller, and Ville Hautamäki. Action Space Shaping in Deep Reinforcement Learning, May 2020.
- [15] OpenAI, Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemyslaw Debiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique P. d O. Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang. Dota 2 with Large Scale Deep Reinforcement Learning, December 2019.
- [16] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur procedures Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *nature*, 550(7676):354–359, 2017.
- [17] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*, pages 1861–1870. PMLR, 2018.
- [18] Trapit Bansal, Jakub Pachocki, Szymon Sidor, Ilya Sutskever, and Igor Mordatch. Emergent Complexity via Multi-Agent Competition. *arXiv*, October 2017.
- [19] Martin Riedmiller, Roland Hafner, Thomas Lampe, Michael Neunert, Jonas Degraeve, Tom Van de Wiele, Volodymyr Mnih, Nicolas Heess, and Jost Tobias Springenberg. Learning by Playing - Solving Sparse Reward Tasks from Scratch. *arXiv*, February 2018.
- [20] Sinan Ibrahim, Mostafa Mostafa, Ali Jnadi, Hadi Salloum, and Pavel Osinenko. Comprehensive Overview of Reward Engineering and Shaping in Advancing Reinforcement Learning Applications. *arXiv*, July 2024.

Bibliography

- [21] Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John Agapiou, Julian Schrittwieser, John Quan, Stephen Gaffney, Stig Petersen, Karen Simonyan, Tom Schaul, Hado van Hasselt, David Silver, Timothy Lillicrap, Kevin Calderone, Paul Keet, Anthony Brunasso, David Lawrence, Anders Ekermo, Jacob Repp, and Rodney Tsing. StarCraft II: A New Challenge for Reinforcement Learning. *arXiv*, August 2017.
- [22] Peng Sun, Xinghai Sun, Lei Han, Jiechao Xiong, Qing Wang, Bo Li, Yang Zheng, Jiaping Ji, Rui Zhang, Zhengying Yin, Jian Peng, Yinqi Qi, Valentin Flunkert, et al. TStarBots: Defeating the Cheating Level Built-in AI in StarCraft II in the Full Game. *arXiv*, September 2018.
- [23] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharmashan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529–533, February 2015.
- [24] Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning*, 8(3):293–321, 1992.
- [25] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015.
- [26] Giovanni Varricchione, Natasha Alechina, Mehdi Dastani, Giuseppe De Giacomo, Brian Logan, and Giuseppe Perelli. Pure-Past Action Masking. *AAAI*, 38(19):21646–21655, March 2024.
- [27] Michael L Littman. Markov games as a framework for multi-agent reinforcement learning. In *Machine learning proceedings 1994*, pages 157–163. Elsevier, 1994.
- [28] Ryan Lowe, Yi Wu, Aviv procedures Tamar, Jean Harb, OpenAI Pieter Abbeel, and Igor Mordatch. Multi-agent actor-critic for mixed

Bibliography

- cooperative-competitive environments. In *Advances in neural information processing systems*, pages 6379–6390, 2017.
- [29] Vinod Nair and Geoffrey E. Hinton. Rectified Linear Units Improve Restricted Boltzmann Machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML)*, pages 807–814, Haifa, Israel, June 2010.
- [30] Andrew M. Saxe, James L. McClelland, and Surya Ganguli. Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. *arXiv*, December 2013.
- [31] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.
- [32] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training Recurrent Neural Networks. *arXiv*, November 2012.
- [33] Alfredo V. Clemente, Humberto N. Castejón, and Arjun Chandra. Efficient Parallel Methods for Deep Reinforcement Learning. *arXiv*, May 2017.
- [34] Warwick Masson, Pravesh Ranchod, and George Konidaris. Reinforcement Learning with Parameterized Actions. *arXiv*, September 2015.
- [35] Adil Zouitine. Masking in Deep Reinforcement Learning, April 2026.
- [36] Santiago Ontañón, Gabriel Synnaeve, Alberto Uriarte, Florian Richoux, David Churchill, and Mike Preuss. A Survey of Real-Time Strategy Game AI Research and Competition in StarCraft. *IEEE Trans. Comput. Intell. AI Games*, 5(4):293–311, October 2013.
- [37] David Balduzzi, Marta Garnelo, Yoram Bachrach, Wojciech M. Czarnecki, Julien Perolat, Max Jaderberg, and Thore Graepel. Open-ended Learning in Symmetric Zero-sum Games. *arXiv*, January 2019.
- [38] Hado van Hasselt, Arthur Guez, and David Silver. Deep Reinforcement Learning with Double Q-learning. *arXiv*, September 2015.

Bibliography

- [39] Matthew Riemer, Gopeshh Subbaraj, Glen Berseth, and Irina Rish. Enabling Realtime Reinforcement Learning at Scale with Staggered Asynchronous Inference. *arXiv*, December 2024.
- [40] Contributors to Wikimedia projects. Law of large numbers - Wikipedia, April 2026. [Online; accessed 18. Apr. 2026].
- [41] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep Reinforcement Learning that Matters. *arXiv*, September 2017.
- [42] Nixos. Available at: <https://nixos.org/>. Accessed: 2025-10-15.
- [43] Klavio Tarka. Saya, 2025. Available at: <https://github.com/Skyfrei/Saya>. Accessed: 2025-04-29.
- [44] Rahul Iyer, Yuezhong Li, Huao Li, Michael Lewis, Ramona Sundar, and Katia Sycara. Transparency and explanation in deep reinforcement learning neural networks. In *Proceedings of the 2018 AAAI/ACM Conference on AI, Ethics, and Society*, pages 144–150, 2018.

German abstract

Real-Time Strategy (RTS) ist ein Genre von Videospielen, das nicht in Runden abläuft; dies impliziert, dass das Spiel standardmaessig in Echtzeit gespielt wird. RTS-Spiele erfordern kontinuierliche, sofortige Entscheidungsfindungen, was sich von Brettspielen wie Schach oder Go unterscheidet, bei denen Spieler Zeit haben, ihren naechsten Zug in einer statischen Umgebung zu berechnen. Die Spieler muessen ihre Wirtschaft verwalten, Ressourcen pluendern, Gebaeude errichten und Einheiten befehligen, waehrend der Gegner genau dasselbe in Echtzeit tut. Dieser Zeit- und Aktionsfluss schafft eine schwierige Umgebung, die durch massive Zustandsraeume, unvollstaendige Informationen (oft durch den Kriegsnebel verborgen, den diese Arbeit nicht behandeln wird) und die Notwendigkeit zur Optimierung von Strategien fuer das kurz- und langfristige Gameplay gekennzeichnet ist. Aufgrund dieser Eigenschaften ist bekannt, dass diese Spiele mit kuenstlicher Intelligenz schwer zu loesen sind. Um diese Umgebungen erfolgreich zu navigieren, kann sich ein Agent nicht einfach auf Brute-Force-Suchalgorithmen verlassen, da der Verzweigungsfaktor moeglicher Aktionen in einem RTS-Spiel ungewoehnlich gross ist. Deshalb erfordern Reinforcement Learning (RL)-Agenten, die gegeneinander antreten, Setups, die optimales Verhalten durch Versuch und Irrtum lehren, waehrend die Suchtiefe pro Spielzustand begrenzt ist. Reinforcement Learning hat eine Geschichte der Eroberung immer komplexerer Spiele, angefangen bei Atari-Titeln bis hin zur Meisterung von Brettspielen wie dem beruehmten GO (Baduk). Der Uebergang von klassischen rundenbasierten Spielen zu kontinuierlichen Multi-Agenten-RTS-Spielen erfordert voellig andere Ansaetze. Die Agenten muessen lernen, ihre unmittelbaren Belohnungen und ihre verzoegerten Belohnungen abzuwaegen, wie zum Beispiel den Gewinn eines kleinen Scharmuetzels, aber den Verlust der Basis, oder die Investition in eine staerkere Wirtschaft, die ein paar Minuten spaeter im Spiel eine staerkere Armee hervorbringen wird.

Bibliography

Diese Arbeit befasst sich mit den wichtigsten Herausforderungen bei der Anwendung von RL in diesem Bereich, einschliesslich der dynamischen Ein-/Ausgabe-Repraesentation, der rechnerischen Effizienz sowie der Integration und dem Vergleich von Reinforcement-Learning-Modellen in einem benutzerdefinierten Echtzeit-Strategie-Framework. Die Ziele dieser Arbeit lassen sich wie folgt zusammenfassen: zu dem Aufbau eines zugänglichen Werkzeugs, das Strategielernen von Reinforcement-Learning-Agenten ermöglicht, die Bereitstellung eines RTS-Werkzeugs, bei dem das Experimentieren nicht an ein kommerzielles Produkt gebunden ist, und die Praesentation einer kleinen Umfrage darueber, wie dieses Werkzeug menschlichen Spielern helfen kann, neue Strategien speziell in RTS-Spielen zu erlernen. Dies geschieht durch die Implementierung von zwei grundlegend unterschiedlichen Reinforcement-Learning-Algorithmen, respektive Deep Q-Networks (DQN) und Proximal Policy Optimization (PPO). DQN repraesentiert einen wertbasierten, Off-Policy-Ansatz, bei dem der Agent lernt, die zukuenftigen Belohnungen spezifischer Aktionen in spezifischen Zustaenden zu schaetzen. PPO ist ein On-Policy-Actor-Critic-Algorithmus, der die Richtlinie (Policy) selbst direkt optimiert und Aktualisierungen glaettet (durch Begrenzung der Aktualisierungsrate), um ein stabiles Lernen zu gewaehrleisten. Der Vergleich und die Implementierung dieser beiden Algorithmen in dieser Umgebung dienen als zentrale Studie fuer diese Arbeit, analysierend, wie sie gegeneinander abschneiden, spezifisch hinsichtlich ihrer Platzkomplexitaet, ihrem rechnerischen Aufwand und der gesamten strategischen Leistung. Reinforcement Learning wurde bereits in mehreren anderen Arbeiten auf RTS-Spiele angewendet, am namhaftesten in Arbeiten, die sich auf StarCraft II konzentrieren, wie das AlphaStar-Projekt und anschliessende gross angelegte Offline-RL-Studien wie AlphaStar Unplugged. Die meisten dieser hochkaraetigen Arbeiten zeigen jedoch keinen direkten Vergleich von zwei voellig unterschiedlichen algorithmischen Paradigmen. Stattdessen fokussieren sie sich auf die Skalierung eines einzelnen Algorithmus durch das Hinzufuegen zusaetzlicher Tweaks, riesiger Mengen an Rechenleistung und komplexer Liga-Trainingssysteme, die den Forscher dazu verpflichten, Zugriff auf eine massive Datenbank von Trainingsdaten zu haben. Ein weiteres Problem in der aktuellen Landschaft der RTS-KI-Forschung ist der statische Zustand der Umgebungen. In grossen kommerziellen Titeln, die fuer die Forschung genutzt werden, kann man die Zustandsdarstel-

Bibliography

lungen, die Kernmechaniken oder die Aktionen, die im Spiel vollzogen werden, nicht fundamental aendern. Dies fuehrt dazu, dass sowohl die Umgebung als auch der Agent etwas fixiert sind, da diese kommerziellen Spiele nicht Open Source sind und stark durch die bereitgestellten APIs eingeschraenkt werden. Diese Arbeit versucht, solchen Einschränkungen zu entkommen. Sie versucht nicht nur, ein Werkzeug bereitzustellen, das zum Strategielernen und zur Beantwortung der Leistungsfrage zwischen diesen beiden Reinforcement-Learning-Agenten verwendet werden kann, sondern auch eine Plattform fuer andere Experimente zu bieten, ohne dass Trainingsdaten notwendig sind. Durch die Nutzung einer Umgebung, in der Forscher Testdaten erstellen und die Spielmechaniken grundlegend aendern koennen, kann dieses Framework manipuliert werden, um spezifische Experimentanforderungen zu erfuehlen. Ueber den technischen Vergleich der Algorithmen hinaus, wie erwaeht, ist ein weiteres wichtiges Ziel dieser Arbeit das Beobachten einer Umfrage, die von Teilnehmern ausgefuehrt wurde, die fragt, ob das Betrachten dieser Simulation dazu fuehren kann, neue Strategien von der KI, die das Spiel spielt, zu lernen? Menschen, die mit Videospielen interagieren, generieren im Allgemeinen etwas, das man Metas oder optimale Spielweisen nennt, durch Stunden des Sammelns von Wissen und des Machens von Fehlern. Jedoch sind diese Metas im Allgemeinen voreingenommen und nur optimal basierend darauf, was funktioniert hat oder basierend auf der Gewinnrate. In Videospielen wie RTS, in denen die Strategien beinahe unbegrenzt sind, koennen Metas nur basierend auf Strategien definiert werden, die getestet und bewiesen wurden. RL-Agenten versuchen, eine Belohnung zu maximieren, was nur durch Versuch und Irrtum von zuvor getesteten Strategien und die Erforschung neuer Strategien getan werden kann. Sie erforschen den Zustands-Aktions-Raum rein mathematisch, was oft dazu fuehrt, dass sie bizarre, aber optimale Strategien entdecken (dies koennte eine gute oder eine schlechte Sache sein). Wie im Experimente-Abschnitt zu sehen sein wird, waehlen die Agenten waehrend des Tests mit dem Belohnungssystem, nur fuer Gold zu farmen ohne es zu benutzen oder rein anzugreifen, selbst wenn die Einheiten sterben. Zu beobachten, ob menschliche Spieler einen Wert aus diesen Simulationen extrahieren koennen, ist eine interessante Idee. Das Erlernen neuer Strategien durch das Beobachten einer KI-Simulation spart immense Mengen an Zeit, die manuell fuer das Experimentieren oder das Auswendigler-

Bibliography

nen neuer Strategien aufgewendet wurden. Zusätzlich lehrt es Spieler neue Wege, auf ungewöhnliche, Out-of-the-Box-Taktiken zu reagieren, denen sie in echten kompetitiven Spielen gegen andere Spieler begegnen könnten. Wenn ein RL-Agent eine bisher ungesehene Einheitszusammensetzung oder ein unkonventionelles Pathing-Manöver entdeckt, das die aktuelle menschliche Meta bricht, kann dieses Wissen direkt zu den Spielern zurückübertragen werden. Um die oben skizzierten Ziele anzugehen, wird diese Arbeit von den folgenden Forschungsfragen geleitet und die erwarteten Ergebnisse als die Hypothesen unten dargelegt: RQ1: Algorithmische Leistung in RTS-Umgebungen. Wie vergleichen sich wertbasierte (DQN) und Policy-Gradient (PPO) Reinforcement-Learning-Algorithmen in Bezug auf Anpassungsfähigkeit, Trainingsstabilität, Leistung und allgemeine Gewinnrate innerhalb einer hochkomplexen, Echtzeit-Strategie-Umgebung? H1: Hypothese eins ist, dass PPO signifikant besser als DQN sowohl in der Gewinnrate als auch in der strategischen Anpassungsfähigkeit abschneiden wird. PPOs stochastische Politik und das geclippte Surrogat-Ziel sind theoretisch besser ausgerüstet, um die massiven Zustands-Aktions-Räume, verzögerten Belohnungen und subtilen Umweltverschiebungen (wie randomisierte Zugfolgen), die RTS-Spielen innewohnen, zu handhaben, wohingegen DQNs deterministische Natur sich als zu spärlich erweisen wird. RQ2: RL-Agenten als Esports-Lerner. Kann eine agentengetriebene Strategie-GUI menschlichen Spielern neue, mathematisch optimale Strategien beibringen? H2: Hypothese zwei besagt, dass das Aussetzen der Echtzeit-Wahrscheinlichkeitsverteilungen des Aktionsraums eines RL-Agenten durch eine dedizierte GUI es menschlichen Beobachtern ermöglichen wird, effektiv Strategien leicht zu identifizieren, zu verstehen und in ihrem eigenen Gameplay zu implementieren.