UNIVERSITY OF DEBRECEN
FACULTY OF INFORMATICS

THESIS

Exploring Reinforcement Learning in AI agents for Video games

Supervisor:	Candidate:
Dr. Harangi Balázs	Kaddah Zohair
Associate Professor	Computer Science BSc

Debrecen

2025

Table of Contents

List of Tables

List of Figures

1. Introduction

Reinforcement Learning (RL) is a form of machine learning that has become a key concept in the field of artificial intelligence particularly through its application in video games. The core idea of reinforcement learning (RL) is learning by interaction, in which agents adjust their behaviors to maximize numerical rewards based on the feedback from their environment. Due to its effectiveness and adaptability, it has made substantial progress in several fields, most notably gaming, where RL algorithms learn complex strategies by trial-and-error interactions with the environment. Reinforcement Learning, in contrast to supervised and unsupervised deep learning, does not need a pre-existing dataset for the agent's learning process. Reinforcement Learning models start in an unfamiliar environment, of which the agent sees only a representation, referred to as the observation or state. The agent then interacts with the environment by applying an action from the action space.

Mnih et al. developed a method known as Deep Q-Learning, which is a unified enhanced version of Q-Learning with a deep convolutional neural network. It effectively learnt the ability to play many ATARI 2600 games at or above human-level performance [1]. Due to the reinforcement learning simple implementation and the nature of them having built in scoring system games are perfect match for them, they are designed in a way to reward players both short term as well as long term in order for the player to achieve a certain goal. This is ideally suited for the reinforcement educational method. Games are also developed with particular control systems that ought to be simple for human players to master, again assisting the Reinforcement Learning method. These days video games have been used for this purpose and mastered by the trained agents, even beating world class human players in some cases. Deep reinforcement learning developed significantly across the years in many genres of games, from Atari [1], to simple first-person-shooter (FPS), to multiplayer online battle arena (MOBA) [35], etc.

In particular, tank games, with their blend of strategic positioning and real-time decision-making, provide rich environments for applying RL principles. The natural dynamics of these games, defined by the need to adjust to unpredictable spawning locations and target positioning, function as a testing ground for reinforcement learning approaches. Agents can be trained to enhance their targeting mechanisms efficiently, demonstrating the applicability of reinforcement learning in handling complex decision-making challenges in gaming environments. It is essential to explore several algorithms, such as Proximal Policy Optimization (PPO), and its fundamental advantages in training efficiency and convergence in order to make the most of RL's potential in creating intelligent agents for the proposed tank game.

The motivation behind this thesis lies in the growing intersection of RL and video games, which not only serves as a testbed for advancing AI but also has practical implications for game development, player experience enhancement, and even real-world applications like autonomous driving and healthcare. By exploring the potential of RL in video games, this research aims to contribute to both the academic understanding of RL techniques and their practical implementation in gaming and beyond.

This thesis will explore reinforcement learning in video games in a systematic manner, divided over six chapters. Starting with basic RL ideas such as Markov Decision Processes, Q-learning, and Deep Q-Networks (Chapter 2), it goes on to investigate why video games — both old school arcade and new school multiplayer online battle arena (MOBA) — are some of the best places to use RL, showcasing notable examples like AlphaGo and OpenAI Five (Chapter 3). Chapter 4 will summarize important RL algorithms; PPO and Actor-Critic based methods; Chapter 5 will present the experimental implementation, which will cover comparisons between training runs along with the episode length analysis. The work is then concluded, with synthesized findings and contributions in Chapter 6.

2. Fundamentals of Reinforcement Learning

2.1 Key concepts in RL

Reinforcement learning (RL) is based on interactions between agents and their environment. The agent looks around the environment and takes actions that aim to maximize the rewards which is its primary goal, the agent is the learner or the decision maker in this situation. The environment provides the agent with states, which describe the current situation, and rewards, which are scalar feedback signals indicating the immediate desirability of the agent's actions, based on the reward received the agent will start taking actions that lead to favorable outcomes. Rewards serve as the fundamental guide for the agent, defining what is good or bad in the short term. This interaction between the agent, environment, and rewards forms the core foundation of reinforcement learning.[7]

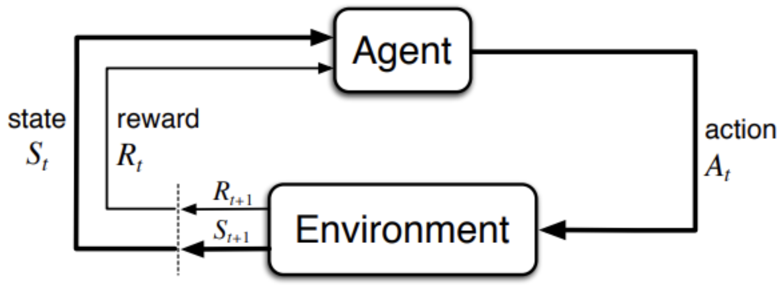


Figure 1 Agent–environment interaction in reinforcement learning. [7]

Markov Decision Process

A key framework for modeling such decision-making processes in RL is the Markov Decision Process (MDP). MDPs provide a structured way to represent decision-making under uncertainty, allowing an agent to determine the best actions that maximize cumulative rewards over time. MDP consists of states, actions, transition probabilities, and rewards. These components work together to model a variety of real-world situations characterized by uncertainty. The core principle of an MDP is its Markov property, which asserts that future states depend solely on the current state and taken action, independent of prior events.[7][8]

States

The set of states S represents all possible configurations or situations that the environment can be in. In a finite MDP, the state space is defined as a finite set $\{s1,s2,...,sN\}$, where N is the size of the state space. Every significant aspect of the problem being modeled can be uniquely described by a state. In a chess game, for instance, a state would stand for the entire board arrangement, including the locations of every piece. In many situations, all of the states in S are regarded as legal, and states can be discrete and represented by different symbols. More complex models, on the other hand, may use features to characterize states, and certain feature combinations may result in states that are illegal.[7][8]

Actions

Actions represent the choices an agent can make in a given state. The set of all possible actions is denoted as A, where $|A| = K$ means there are K possible actions. The selection of actions is crucial as it directly influences the outcomes in terms of rewards. In some cases, not all actions can be taken in every state, so the set of available actions in a specific state s is denoted as $A(s) \subseteq A$.[7][8]

Transition Function

A transition function defines how an agent moves between states when taking actions. It is represented mathematically as $T(s,a,s')$, which gives the probability of reaching state s′ after taking action a in state s. These transition probabilities must sum to 1 for a given state-action pair, ensuring that the agent always transitions to some state.

If an action is not valid in a particular state, the transition probability to any state can be set to zero. MDPs follow the Markov property, meaning the next state depends only on the current state and action, not on past states or actions. This implies that taking the same action in the same state will always result in the same probability distribution over possible next states.

A transition graph visually represents these state changes. State nodes connect to action nodes, which then lead to next states based on probabilities. Each transition is labeled with the probability of moving to the next state and the expected reward associated with that transition.[7][8]

Value functions

Alongside transition functions, reinforcement learning also uses two key value functions: the state-value function and the action-value function. A value function tells how good a situation or action is for the agent by estimating the total rewards it can get in the future when it follows a certain policy. [7]

State-Value function v_π(s)

It represents the expected return when starting from state s and following a given policy π thereafter. The equation implies that the value of a state under a policy π is the sum of the total expected future rewards, adjusted for γ, which represents how much future rewards are valued compared to immediate rewards.

$$v_\pi(s) = E_\pi[\, G_t \mid S_t = s\,] = E_\pi[\sum_{\{k=0\}}^{\{\infty\}} \gamma^k R_{\{t+k+1\}} \mid S_t = s] \quad [7] \qquad (1)$$

Action-Value Function q_π(s,a)

It represents the expected return when starting from state s, taking action a, and following the policy π thereafter. This function gives the expected return for choosing a particular action in a given state.[6]

$$q_\pi(s,a) = E_\pi[G_t \mid S_t = s, A_t = a] = E_\pi[\sum_{\{k=0\}}^{\infty} \gamma^k R_{\{t+k+1\}} \mid S_t = s, A_t = a] \quad [7] \qquad (2)$$

Bellman Equations

The value functions follow recursive relationships defined as Bellman equations, which describe how the value of a state (or action) is connected to the values of its successor states.

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s'} T(s,a,s')\,[R(s,a,s') + \gamma v_\pi(s')] \qquad [7] \qquad (3)$$

Why Are Value Functions Important?

It helps estimate how good a given policy is and it guides the selection of better policies. [7]

Reward Function

The reward function, on the other hand, defines the reward the agent receives for being in a certain state or for taking a specific action. It can be based on the state alone, the action taken in a state, or the transition from one state to another through an action. These different forms of reward functions are interchangeable, but transition-based rewards are especially useful in model-free reinforcement learning, where both the starting and ending states are used to update the agent's understanding.

The reward function plays a key role in setting the goal for learning. In games, for example, an agent may receive a positive reward for winning, a negative reward for losing, and zero reward for a draw. Sometimes, rewards are also given during the game to encourage progress toward the final goal. These rewards guide the agent toward desirable behavior during training.[7][8]

Markov Property

The Markov property in reinforcement learning means that the current state provides all the necessary information for decision-making, without needing to consider the full history of past states. A process follows the Markov property if the probability of transitioning to the next state and receiving a reward depends only on the current state and action. This simplifies learning and makes it more efficient. In many real-world applications, environments may not strictly follow the

Markov property due to hidden or missing information, but reinforcement learning algorithms often aim to approximate these conditions to improve performance. [7]

Policy

A policy in reinforcement learning is a set of rules that tells an agent which action to take in each state. It acts like a decision-making guide for the agent. If the policy is deterministic, it always picks the same action for a given state (written as π(s) = a). If the policy is stochastic, it randomly chooses actions based on those probabilities. When an agent follows a policy, it starts in an initial state, selects an action based on the policy, moves to a new state based on the environment's rules, and receives a reward. This cycle repeats, forming a sequence of states, actions, and rewards. If the task has an end (like reaching a goal), the process restarts in a new state. Otherwise, it continues indefinitely.[7]

Q-learning

Originally proposed by Watkins, Q-learning is a popular reinforcement learning algorithm designed for solving Markov Decision Processes (MDPs), it is used to find the optimal action-selection policy for an agent interacting with an environment. It is widely used because of its simplicity, efficiency, and ability to learn optimal policies without a model of the environment. These interactions are typically structured within the framework of a Markov Decision Process (MDP). [10] [11]

The algorithm maintains a table of Q-values, Q*(s,a) which represents the expected cumulative reward for taking action a in state s and following the optimal policy, These values are updated using the fundamental equation:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\left[r_t + \gamma \backslash max_a Q\left(s_{\{t+1\}}, a\right) - Q(s_t, a_t)\right] \qquad [7] \qquad (4)$$

Where α is the learning rate, γ is the discount factor., rt is the reward received after taking action at., s{t+1} is the next state.[6] Striking the right balance between exploration and exploitation is crucial in Q-learning, as excessive exploitation may trap the agent in suboptimal policies, while excessive exploration can delay convergence to the optimal solution.

[6][10][11]

Exploration vs Exploitation

The RL Approach relies on the two terms exploration and exploitation for self-improvement through feedback received from the environment. The agent must explore new actions to discover their rewards but also has to exploit in order to obtain better action selections in the future. Striking this balance is important as too much exploration leads to resource wastage and inefficient learning while too much exploitation may settle into a suboptimal path and leave better actions unexplored. If you keep tracking the value of all available actions, at any point in time there will always be at least one action that has the highest value is an action that can provide the maximum value. This set of actions is called greedy actions. In case you do select a greedy action we will say that you are taking advantage of what you have at hand. If you opt for one of the nongreedy actions then we say that you are in the process of exploring because with this approach you stand to refine the value of the nongreedy action. From recommendation systems to robotic control, mastering this balance enables AI systems to learn efficiently while avoiding costly stagnation in real-world applications. [7][12][13]

2.2 Deep Reinforcement Learning (DRL)

Deep Reinforcement Learning (DRL) integrates Reinforcement Learning (RL) with Deep Learning (DL), which has emerged as one of the most fascinating research domains in artificial intelligence to date. DRL is concerned with problems where the state or action space is continuous; value and policy functions can be implemented via neural networks, eliminating the need for cumbersome table lookups. As with RL, DRL can also be subdivided into model-based algorithms and model-free algorithms. In model-based RL, the agent builds a model of the environment as an intermediate step. This model is typically a Markov-decision process (MDP), which consists of a state transition model (which predicts the next state), and a reward model (which predicts the expected reward during that transition), enabling computation of optimal value or policy for the MDP. Dyna-Q [15] and MuZero [16] are examples of model-based algorithms. In contrast, model-free algorithms do not learn any dynamics model; instead, they learn policies or value functions from experience, or through trial and error.

[14][15][16]

Neural Networks in RL

Deep Reinforcement Learning uses deep neural networks and their high modelling capacity to model an agent's behavior [17]. Neural networks play a central role in reinforcement learning (RL) by serving as functional approximators that allow agents to estimate the value of various actions in a given state space [7]. This capability is particularly crucial when dealing with high-dimensional inputs, such as images or complex state representations. Deep reinforcement learning (DRL) has emerged from the combination of deep learning techniques, which utilize deep neural networks, and reinforcement learning frameworks, allowing the development of sophisticated algorithms capable of learning from high-dimensional environments effectively. The architecture typically consists of multiple layers that process input data and output predictions regarding action values or policies, thus enabling agents to discern the best course of action based on expected rewards [18]. Variants of neural networks used in RL primarily include convolutional neural networks (CNNs) and recurrent neural networks (RNNs). CNNs are highly effective in dealing with visual data, as they can automatically identify patterns in images, facilitating tasks such as visual recognition and localization in environments [19]. RNNs, particularly Long Short-Term Memory (LSTM) networks, are adept at capturing temporal dependencies in sequential data, making them suitable for tasks where the order of inputs affects the outcome. This temporal aspect is particularly relevant in environments where previous states influence future decisions, helping maintain context within learning algorithms [20]. The development of these neural network architectures has significantly enhanced the agents' performance in complex RL tasks, demonstrating profound success in diverse applications such as gaming and robotic control [21].

Deep Q-Networks

Deep Q-Networks (DQN) represent a significant advancement in reinforcement learning by integrating deep neural networks with Q-learning to handle environments characterized by high-dimensional and raw sensory inputs. The DQN algorithm operates by approximating the Q-value function through a deep learning architecture, enabling the agent to derive action-value estimates directly from the high-dimensional input space. The foundational work by Mnih et al. illustrated that DQNs could achieve human-level performance on various Atari games through the construction of a deep convolutional network that processes visual inputs, demonstrating enhanced efficiency in learning optimal policies compared to standard Q-learning approaches which rely on discrete state representations and manually engineered features [22].

The key features of DQNs include the use of experience replay and target networks, which address some limitations of traditional Q-learning, such as instability and overestimation of Q-values. Experience replay allows the agent to utilize past experiences more effectively by storing them in a replay buffer and sampling from this buffer during training. This randomized sampling helps to break the correlation between consecutive experiences, stabilizing training. Additionally, the implementation of target networks, which are periodically updated copies of the main Q-network, further reduces the variance in Q-value updates, enhancing learning performance [23]. Variants of DQN, such as Double DQN (DDQN) and Dueling DQN, have emerged to mitigate overestimation bias in Q-value estimation, where DDQN separately determines action selection and evaluation, leading to more stable learning dynamics [24][25].

3. Reinforcement Learning in Video Games

3.1 Why Video Games are Ideal for RL Research

Reinforcement Learning (RL) has a rich history in the context of video games, dating back to the early explorations of artificial intelligence in gaming. The foundational algorithms, such as Q-learning, began to gain traction in the 1980s, but significant advancements were not realized until the advent of Deep Reinforcement Learning (DRL) in the 2010s. One of the pivotal moments in this evolution was the introduction of the Deep Q-Network (DQN) by Mnih et al. in 2015, which successfully demonstrated that neural networks could learn to play Atari 2600 games directly from raw pixel data, achieving performance levels comparable to human players [22]. This marked a revolutionary step in RL, showcasing the potential of combining deep learning techniques with traditional RL frameworks. Subsequently, further innovations, such as the Proximal Policy Optimization (PPO) algorithm, enhanced the framework for policy optimization in complex gaming environments, solidifying DRL's applicability [32].

Video games serve as an excellent platform for reinforcement learning (RL) research due to several notable characteristics that align with the fundamental principles of RL. First and foremost, video games typically provide structured environments with clear rules, objectives, and feedback systems. This structure allows RL agents to receive immediate feedback from their actions, enabling them to optimize their decision-making processes through trial and error [33]. The dynamic nature of video games, where different scenarios and challenges can be encountered during gameplay, creates a rich environment for testing RL algorithms in various contexts, including unsupervised learning [34]. For example, games that require strategic thinking, such as "StarCraft II," promote the development of sophisticated multi-agent reinforcement learning strategies that can simulate real-world competition and collaboration among players and AI [35].

3.2 Types of Video Games in RL Research

Reinforcement Learning (RL) has significantly influenced various types of video games, providing a diverse landscape for experimentation and development of intelligent agents. This section discusses four main categories of video games used in RL research: Classic Arcade Games, Real-Time Strategy Games, Multiplayer Online Battle Arenas, and Open-World Games.

Classic Arcade Games

The classic arcade games, and in particular the Atari 2600 games, are mainstays in the world of reinforcement learning (RL) research. Games with simple mechanics but complex control decision requirements like Breakout (paddle control) and Space Invaders (timing and evasion) are ideal for testing RL algorithms in high-dimensional state spaces using heuristic search methods or genetic algorithms. The landmark work of Mnih et al. in 2013 showed that a Deep Q-Network (DQN) could learn by directly acting on the game's pixels fed to it without any pre-processing, subsequently outperforming the vanilla methods and even achieving superhuman performance in some games. DQNs demonstrated the power of merging experience replay with Q-learning, convolutional neural networks, and booster learning. They set the precedent for training agents in rich visual environments using undermined prior knowledge. [1]

Even in their simplest forms, Atari games exhibit important challenges within RL: missing information (partial observability) such as inferring ball trajectory from frames, sparse rewards (e.g., scoring in Pong), and the requirement for long term planning (e.g., Seaquest's oxygen management). Such games continue to play a major role in benchmarking progress in sample efficiency, generalization, and robustness, forming a bridge between simplified problems and modern 3D environments. [1]

Real-Time Strategy Games

Regarding complexity, real-time strategy (RTS) games such as StarCraft II are a considerable step forward for reinforcement learning (RL) over traditional arcade games. Even though Mnih et al.'s DQN was able to master Atari using bare image inputs [1] , RTS games introduce issues like partial observability, hierarchical decision-making, and huge action spaces (e.g., StarCraft II has roughly 524 action functions with spatial arguments). The Asynchronous Advantage Actor-Critic (A3C) algorithm based on Mnih's earlier work has emerged as one of the dominant approaches for handling long-term planning and multi-scale actions in RTS games through parallel actor-learners. [1] [35]

Macro-strategies like resource management and base building against micro-tactics like unit control are typical in games like StarCraft II that often necessitate transfer learning from previous ones involving DefeatRoaches to others such as CollectMineralShards. In this regard, Alghanem & Gopalak's experiments in 2018 demonstrated how sub-human performance could be achieved by A3C with spatial-non-spatial hybrid networks comprising convolutional layers for screen/minimap inputs + fully connected layers for abstract actions but still struggled with full-game scenarios such as BuildMarines thereby pointing out the necessity of memory-augmented architectures including LSTMs and exploration incentives escaping local optima. [35]

Multiplayer Online Battle Arenas

The implementation of reinforcement learning in multiplayer online battle arenas like Dota 2 comes with particular obstacles, including high-dimensional action spaces, long time horizons, and partial observability. Dota 2 games can stretch up to an hour, meaning agents need to plan for tens of thousands of steps while operating under the "fog of war". This limits visibility to areas around allied units. OpenAI Five addressed these problems using self-play with Proximal Policy Optimization (PPO), processing roughly 2 million frames per every 2 seconds across thousands of GPUs. The agent chose to make use of semantic observations instead of raw pixels and offered shaped rewards to help learning towards subgoals such as resource collection and tower destruction. Moreover, coordinating multiple agents was achieved using team spirit techniques, while "surgery" allowed continual adaptation to game updates without resetting progress.

OpenAI Five's scripted aspects like item purchases and limited support for only 17 of 117 heroes are some of the unsolved challenges. Generalization could be expanded alongside the integration of additional actions into reinforcement learning and human-aligned constraints like reaction times improved. The realm of multiplayer online battle arenas holds endless possibilities for testing reinforcement learning in more complex multi-agent settings, especially with practical, non-gaming uses. [36]

Open-World Games

Reinforcement Learning in open-world environments such as “sandbox” games like Minecraft all come with their unique set of challenges and opportunities. Unlike traditional games, open-world environments adopt non-linear gameplay markets which requires a process of free exploration, goal management, and execution toward sparse rewards. While some stratgies such as hierarchical RL (Director) focus on subdividing the larger goal into smaller and manageable tasks, others such as model-based methods (DreamerV3, LS-Imagine) focus on simulating and predicting future steps to enable improve with planning. As an example, LS-Imagine outperformed other algorithms with its jumpy transition mechanics which changes the way agents move through different states with their use of affordance maps to more efficiently traverse large state spaces. When tested with benchmarks like MineDojo, LS-Imagine's results demonstrated superiority over traditional methods.

Prospective advancements are employing language models for task breakdown alongside collaboration of multi-agents to enable more complex objectives. In conjunction these changes may allow better adaptability as well as improve long term planning robotics - spearheading next generation AI technology. Open world gaming will greatly enhance artificial intelligence while simultaneously providing a glimpse into future real world applications. [37]

3.3 Case Studies of RL in Video Games

In the realm of reinforcement learning (RL), several landmark case studies have showcased its efficacy in video game environments, demonstrating the adaptability of various algorithms.

AlphaGo and AlphaStar

AlphaGo, developed by DeepMind, marked a significant achievement in the use of RL for playing the ancient game of Go. The system employed deep reinforcement learning combined with a Monte Carlo tree search algorithm, enabling it to learn optimal strategies through self-play. Notably, AlphaGo defeated world champion Go player Lee Sedol, achieving a level of strategic depth that had previously been considered unattainable for computer programs [38]. This approach was subsequently refined in AlphaGo Zero, which learned solely from self-play without any human data, further illustrating the capabilities of RL to master complex strategies independently [38].

AlphaStar, developed by DeepMind, is the first AI system to defeat a professional *StarCraft II* player, marking a milestone in AI research [39]. While it shares foundational techniques with AlphaGo—such as deep reinforcement learning (RL) and self-play—it diverges significantly by incorporating evolutionary computation (EC) principles to address the complexities of real-time strategy games. Unlike AlphaGo, which relied on Monte Carlo Tree Search (MCTS) for turn-based play, AlphaStar employs Population-Based Training (PBT)—a form of Lamarckian evolution—where neural networks continuously optimize via gradient descent while their hyperparameters evolve competitively [39]. Additionally, AlphaStar uses competitive co-evolution and quality diversity (QD) to maintain a diverse population of strategies, ensuring robustness and adaptability, whereas AlphaGo focused on a single dominant policy. This evolutionary approach, as highlighted by [39], distinguishes AlphaStar as a novel synthesis of RL and EC, extending DeepMind's lineage beyond board games to dynamic, imperfect-information environments like *StarCraft II*.

OpenAI Five

OpenAI Five is a reinforcement learning system developed by OpenAI that achieved superhuman performance in the complex multiplayer online battle arena game *Dota 2*. It represents another landmark case in the application of RL. This system employed a decentralized approach, using multiple agents trained through a variant of RL known as Proximal Policy Optimization (PPO). OpenAI Five demonstrated advanced teamwork and situational awareness, effectively coordinating actions among agents, which was further supported by hierarchical reinforcement learning methodologies. OpenAI Five competed against professional human teams, achieving notable victories and exemplifying RL's capabilities in cooperative and competitive environments.[36]

DeepMind’s Atari Experiments

DeepMind's Atari experiments, as detailed in their seminal 2013 and 2015 papers written by Volodymyr Mnih [1][22], demonstrated the potential of deep reinforcement learning by training an agent to play Atari 2600 games at superhuman levels using only raw pixel input. Their approach combined Q-learning with deep neural networks (DQN) and introduced key innovations like experience replay, which stores and randomly samples past transitions to decorrelate training data, and fixed Q-targets to stabilize learning. The agent achieved remarkable performance across a variety of games, including Breakout, where it surpassed human scores by discovering advanced strategies like tunneling through bricks. However, the method had limitations, particularly in games requiring long-term planning (e.g., Space Invaders), highlighting challenges in credit assignment and exploration. These experiments marked a major milestone in AI, showing that a single algorithm could learn diverse tasks from high-dimensional sensory input without game-specific feature engineering.[1][22]

4. Techniques and Algorithms for RL in Video Games

4.1 Model-Free vs. Model-Based RL

Reinforcement learning has made significant progress in video games, and there are two predominant paradigms: model-free and model-based reinforcement learning. It is crucial to learn the difference between the two approaches in order to select the appropriate algorithms based on the specific needs of a gaming environment.

In the field of reinforcement learning (RL), the distinction between model-free and model-based methods represents a fundamental divide that significantly influences the strategies employed by agents learning to play video games. Each approach has unique characteristics, strengths, and weaknesses that affect an agent's learning efficiency and performance.[14]

Model-Based

Model-based reinforcement learning involves creating an internal model of the environment. This includes learning the transition function, which predicts the next state based on the current state and action, and the reward function, which estimates the reward received when moving from one state to another. With this internal model, the agent can simulate outcomes and plan future actions without needing to interact with the actual environment all the time. This can lead to faster learning and better use of resources. Model-based techniques may include estimating value functions using Bellman equations, predicting future states using deterministic or stochastic models, adjusting policy parameters directly to maximize expected rewards through policy search, and computing returns over time using discounted reward sums. [14]

Model-Free

Model-free reinforcement learning does not attempt to learn an internal model of the environment. Instead, it learns how to act optimally through trial-and-error interactions. This makes model-free methods more flexible in environments that are too complex or unpredictable to model accurately. However, they typically require more experience and interaction to learn effectively.

Model-free techniques generally fall into two categories. The first is policy gradient methods, which optimize the agent's policy directly by adjusting its parameters to maximize the expected return. These methods often use algorithms like REINFORCE or actor-critic. The second is value-based methods, where the agent learns the value of actions in given states, such as in Q-learning, by updating Q-values based on observed rewards and estimated future values. Overall, model-based methods are generally more sample-efficient and allow for planning, while model-free methods are simpler and more robust in complex environments. The choice between them depends on the specific goals, constraints, and complexity of the game being developed. [14]

Factor	**Model-Based RL**	**Model-Free RL**
Environment Interactions	Fewer (uses learned model for planning)	Many (learns purely from experience)
Convergence Speed	Faster (due to efficient planning)	Slower (requires more trial and error)
Prior Knowledge Needed	Yes, requires learning	No model needed, learns directly from rewards
Flexibility	Depends on model accuracy	Adapts well to complex/unknown environments

Table1 Comparison between model-based RL and model-free RL [14]

4.2 Policy Gradient Methods

Policy gradient algorithms are reinforcement learning methods that optimize a policy by moving its parameters toward maximum improvement in performance, on the basis of interaction with the environment instead of purely on value functions. Unlike evolutionary methods, they can exploit explicit feedback from individual experiences and sometimes use value estimates to steer their updates [7]. Such paradigm is particularly well-suited for continuous action spaces and intricate decision-making environments, enabling optimization of possibly more descriptive policies than discrete action environments [26]. Such types of methods have been successful in many areas, from robotics to games, where advanced action-selection mechanisms play a pivotal role in the accomplishment of tasks [27].

One of the significant strengths of policy gradient approaches is their ability to function effectively with deep neural networks, which allows learning of complicated policies that generalize well across a range of states [28]. However, these approaches are usually linked with challenges, primarily high variance in the gradient estimates, which lead to slow convergence [29]. To reduce these issues, different enhancements have been proposed, including the use of baseline functions to reduce variance [27], and the integration of actor-critic architectures that combine the strengths of both policy-based and value-based methods [30]. Furthermore, more advanced techniques such as Proximal Policy Optimization (PPO) and Natural Policy Gradients attempt to stabilize training while offering efficient policy updates, demonstrating the continuous evolution and enhancement of policy gradient techniques in the RL community [31].

DRL Algorithms	**Description**	**Category**
DQN	Deep Q Network	Value-based Off-policy
Double DQN	Double Deep Q Network	Value-based Off-policy
Dueling DQN	Dueling Deep Q Network	Value-based Off-policy
MCTS	Monte Carlo tree search	Value-based On-policy
UCRL-VTR	optimistic planning problem	Value-based Off-policy
DDPG	DQN with Deterministic Policy Gradient	Policy gradient Off-policy
TRPO	Trust Region Policy Optimization	Policy gradient On-policy
PPO	Proximal Policy Optimization	Policy gradient On-policy
ME-TRPO	Model-Ensemble Trust-Region Policy Optimization	Policy gradient On-policy
MB-MPO	Model-Based Meta- Policy-Optimization	Policy gradient On-policy
A3C	Asynchronous Advantage Actor Critic	Actor Critic On-Policy
A2C	Advantage Actor Critic	Actor Critic On-Policy

Table 2 Summary of model-based and model-free DRL algorithms consisting of value-based and policy gradient methods. [14]

REINFORCE Algorithm

The REINFORCE algorithm is a fundamental algorithm in the family of policy gradient reinforcement learning. It works using the policy gradient theorem, which gives a way to find the gradient of the agent's expected returns concerning the agent's policy parameters. In particular, the algorithm generates policies using returns obtained from running policies in the environments. These returns are determined for entire episodes using the equation ($R\_t = \sum_{\{i=t\}}^{T} \gamma^{\{i-t\}} \; r_i$) *(5), where (* r_i) are the rewards received at each time step and (γ) is the discount factor. [39]

Proximal Policy Optimization (PPO)

Proximal Policy Optimization (PPO) is an advancement in reinforcement learning that seeks to improve traditional policy gradients but avoids the complexity of trust region methods such as TRPO. The main contribution is the clipping of the surrogate objective function which prevents the problem of policy update monotonicity loss by restricting how much the new policy can differ from the old policy. Generally, PPO performs a lower bound optimization on a policy performance by taking the minimum between a standard policy gradient objective and one that clips the ratio of probability of old and new policies to be within [1-ε, 1+ε]. This ensures that the ratio is bounded to promote stable training and maintain data efficiency. The data collection process is decoupled from training, wherein data is collected through interaction with the environment followed by multiple epochs of minibatch optimization on the clipped objective. Advantage estimation methods, such as GAE, are used for the optimization process. In comparison to TRPO, the algorithm is easier to implement as it only requires first-order optimization like Adam instead of complex calculations using conjugate gradients. Nevertheless, PPO still retains similar or better outcomes across multiple domains. [31]

PPO has many advantages like better sample efficiency, performance that is stable without the need for significant hyperparameter adjustments, and usefulness in systems both with continuous control and discrete action spaces. The algorithm has two main variants: the primary one with clipped objectives, and the other one with adaptive KL penalties that alter a constraint based on KL divergence. PPO is successful because it manages to balance these three things: the ease of use found in vanilla policy gradients, the stability from region trust methods, and the sample economy found in actor-critic methods. These factors contribute to why PPO is one of the most prominent algorithms used for reinforcement learning, being able to train sophisticated policies like 3D agents in continuous control tasks while being straightforward to implement and tune in practice. [31]

4.3 Actor-Critic Methods

Actor-Critic methods, a subclass of RL algorithms, attempt to merge a policy-based approach with a value-based one to mitigate the high variance associated with pure policy gradient techniques and low bias problems with value-based methods. As with any neural network, the Actor, a neural network, can be trained on policies that maximize rewards and execute actions relevant to the current state. The Actor is trained through policy gradients and updated by the Critic's feedback, which is an estimation model or value function (Q-function/State-value function) that predicts the value of actions: estimate a return on the actions taken.

During operation, the Actor selects an action, which the environment executes, returning a reward and new state. The Critic then evaluates this action by computing the temporal difference (TD) error between predicted and actual rewards, using this signal to refine its own value estimates while also guiding the Actor's policy update via the gradient.

$$\nabla\_\theta J(\theta) = E[\nabla_\theta \log_{\mu\theta}(a \mid s) \cdot Q^{\mu}(s,a)] \qquad [39] \qquad (6)$$

This combination of mechanisms allows lower variance than pure policy gradients through the Critic's stable evaluations, greater stability by balancing policy and value updates, and broad applicability across both continuous and discrete action spaces, making Actor-Critic methods particularly effective for complex RL tasks where neither pure approach would suffice alone. [39]

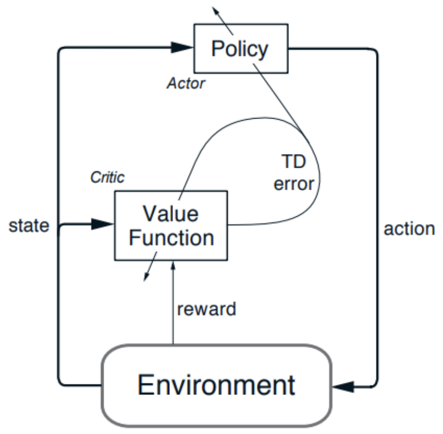


Figure 2 The actor–critic architecture. [7]

5. Experimental Setup and Results

5.1 Selection of Video Game and Environment

For this project, I chose to implement a simple 3D tank game using the Unity game engine. The tank model and environment were obtained from a free asset package on the Unity Asset Store, the environment was developed by Unity Technologies however it did not have anything to do with AI games. While the default environment provided a solid foundation, it was not complicated enough for training a reinforcement learning agent. To counter this, I added more obstacles and varieties of obstacles manually to the scene ex: buildings, trees, and rocks in an attempt to make it hard for the agent to navigate and acquire targets. The core gameplay is for the tank (agent) to find and fire at red box targets that spawn randomly around the environment. To facilitate better decision-making of the agent, I used two tag systems across the environment: an obstacles tag to warn the agent not to move towards them as it moves around the environment and a target's tag to instruct the agent to seek out targets and fire. It is just a basic yet dynamic configuration where one can experiment with reinforcement learning.

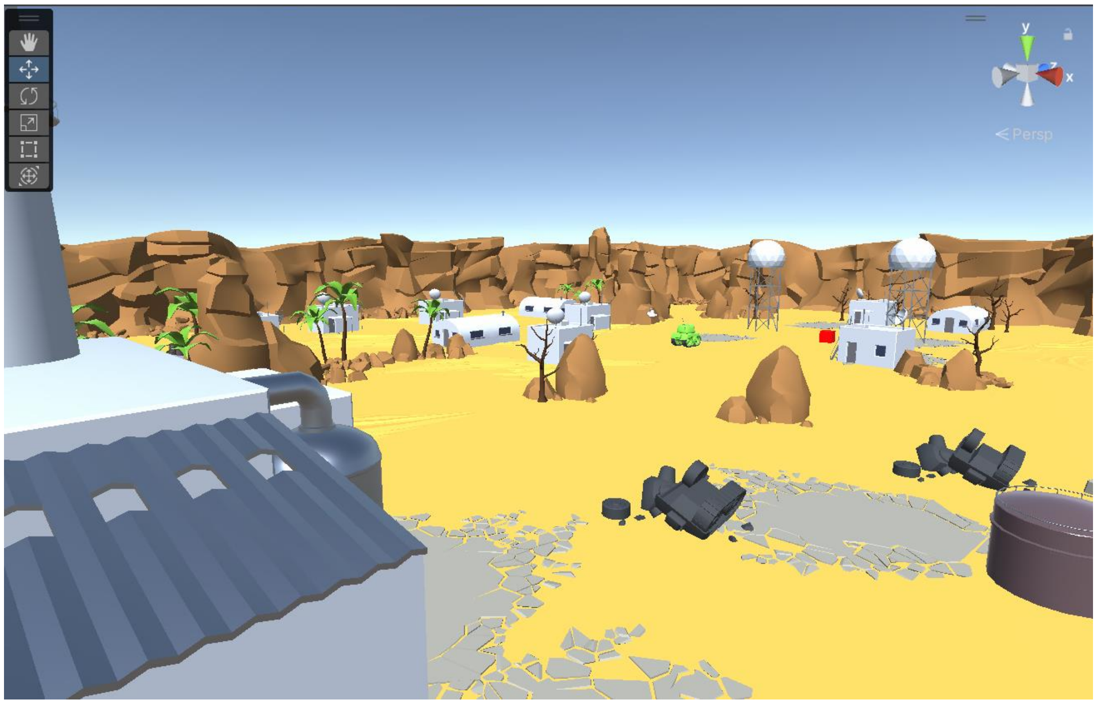
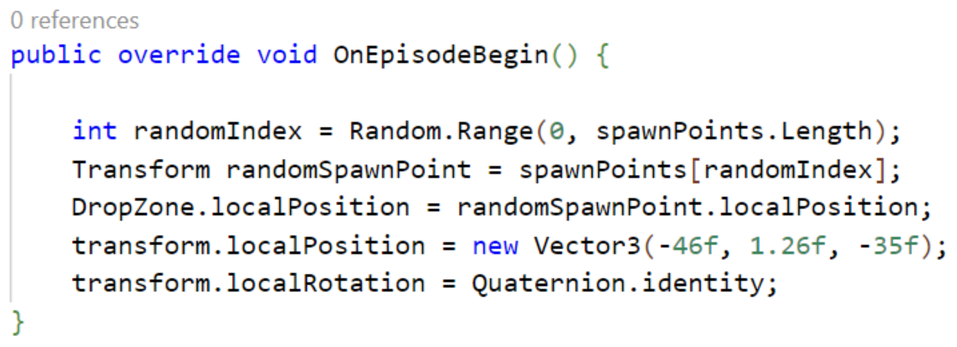


Figure 3 Environment in-use

5.2 Code Implementation

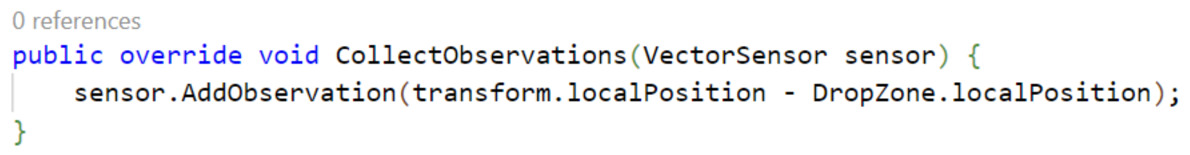
First, I set the speed of the agent's movement and created an array of potential spawn points across the environment where the target could appear. Then, I defined the target object (the DropZone), which the agent should find and shoot, and whose position is randomly initialized at the episode beginning for purposes of generalization. To detect the targets, I created a ray cast slightly above the tank to assist the accuracy and prevent interference from the ground. I also added a shooting effect prefab so the agent visually represents it when shooting. I also established a sphere cast with a cast radius, which determines the distance ahead of the agent that it can "see", the value had to be carefully balanced, as a radius that is too large causes the agent to detect too many obstacles simultaneously, leading to hesitation or inaction. Lastly, I created a static MAX_DISTANCE variable to adjust the reward based on the agent's distance from the target, thereby encouraging the agent to close the distance with the target before shooting.



```
0 references
public override void OnEpisodeBegin() {

    int randomIndex = Random.Range(0, spawnPoints.Length);
    Transform randomSpawnPoint = spawnPoints[randomIndex];
    DropZone.localPosition = randomSpawnPoint.localPosition;
    transform.localPosition = new Vector3(-46f, 1.26f, -35f);
    transform.localRotation = Quaternion.identity;
}
```

Figure 4 OnEpisodeBegin()

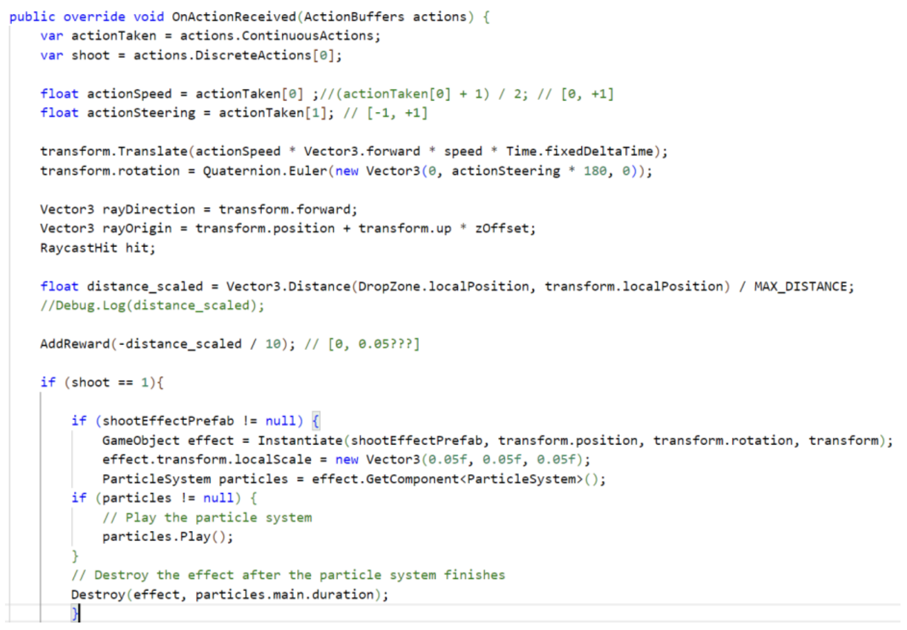
In the OnEpisodeBegin() function, the environment is reset at the start of each training episode. A random target spawn point is chosen from a predefined array of positions, and the DropZone (which represents the target) is placed at that location. Meanwhile, the tank is repositioned to a fixed starting point. This randomization of the target location ensures that the agent will face new challenges for every episode and have to acquire more general techniques, rather than learn one behavior or sequence. This randomness plays a critical role in creating an agent capable of reacting to the variability of an environment.



```
0 references
public override void CollectObservations(VectorSensor sensor) {
    sensor.AddObservation(transform.localPosition - DropZone.localPosition);
}
```

Figure 5 CollectObservations()

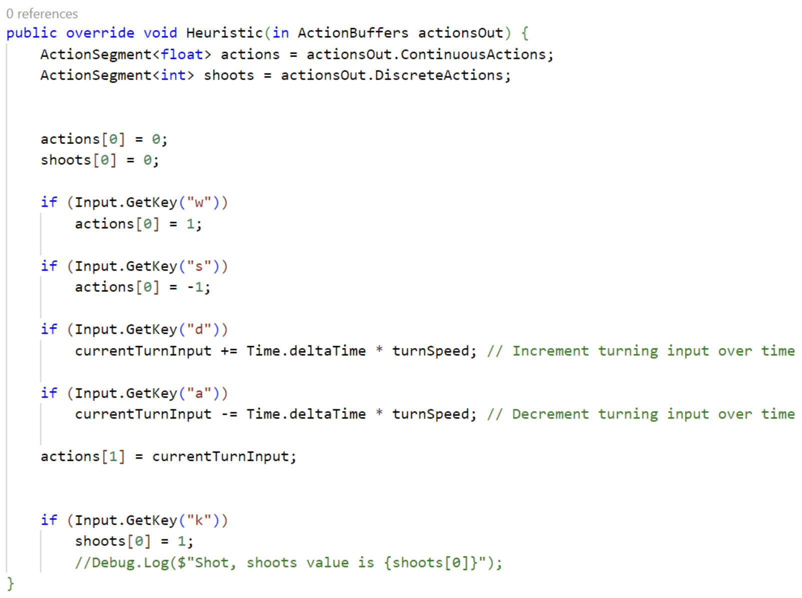
The CollectObservations() function provides the agent with information about the environment that the agent will base its decisions on. Specifically, the agent reads the relative position between the target and the agent. This observation gives the agent a direction vector from the agent's position to the target that the agent can utilize to determine where it needs to go. This brief but explanatory input plays an important role in shaping the agent's action without overwhelming it with unnecessary information.



```
public override void OnActionReceived(ActionBuffers actions) {
    var actionTaken = actions.ContinuousActions;
    var shoot = actions.DiscreteActions[0];

    float actionSpeed = actionTaken[0] ;//(actionTaken[0] + 1) / 2; // [0, +1]
    float actionSteering = actionTaken[1]; // [-1, +1]

    transform.Translate(actionSpeed * Vector3.forward * speed * Time.fixedDeltaTime);
    transform.rotation = Quaternion.Euler(new Vector3(0, actionSteering * 180, 0));

    Vector3 rayDirection = transform.forward;
    Vector3 rayOrigin = transform.position + transform.up * zOffset;
    RaycastHit hit;

    float distance_scaled = Vector3.Distance(DropZone.localPosition, transform.localPosition) / MAX_DISTANCE;
    //Debug.Log(distance_scaled);

    AddReward(-distance_scaled / 10); // [0, 0.05???]

    if (shoot == 1){

        if (shootEffectPrefab != null) {
            GameObject effect = Instantiate(shootEffectPrefab, transform.position, transform.rotation, transform);
            effect.transform.localScale = new Vector3(0.05f, 0.05f, 0.05f);
            ParticleSystem particles = effect.GetComponent<ParticleSystem>();
        if (particles != null) {
            // Play the particle system
            particles.Play();
        }
        // Destroy the effect after the particle system finishes
        Destroy(effect, particles.main.duration);
        }
```

Figure 6 OnActionRecived()

The OnActionReceived() function is where the learning and decision-making process occurs. The agent is provided with two continuous values of action that control its forward movement (actionTaken[0]) and rotation (actionTaken[1]). These are used to control the movement of the tank in the environment. It also receives a discrete action that tells it whether to shoot (shoot == 1). When the agent makes a decision to fire, it sends a SphereCast ahead to find out if there's anything in its way that will serve as a target. When the ray hits an object marked as a "Target," the agent receives a +100 reward, and the episode ends. Parallelly, the agent is constantly receiving a small negative reward based on how close it is to the target, prompting it to narrow the gap eventually. This reward policy directs the agent to close in on the target as quickly as possible and shoot when it's in range.



```
0 references
public override void Heuristic(in ActionBuffers actionsOut) {
    ActionSegment<float> actions = actionsOut.ContinuousActions;
    ActionSegment<int> shoots = actionsOut.DiscreteActions;


    actions[0] = 0;
    shoots[0] = 0;

    if (Input.GetKey("w"))
        actions[0] = 1;

    if (Input.GetKey("s"))
        actions[0] = -1;

    if (Input.GetKey("d"))
        currentTurnInput += Time.deltaTime * turnSpeed; // Increment turning input over time

    if (Input.GetKey("a"))
        currentTurnInput -= Time.deltaTime * turnSpeed; // Decrement turning input over time

    actions[1] = currentTurnInput;


    if (Input.GetKey("k"))
        shoots[0] = 1;
        //Debug.Log($"Shot, shoots value is {shoots[0]}");
}
```

Figure 7 Heuristic()

The Heuristic() function is used to manually control the agent using keyboard input, primarily for testing or debugging purposes. Instead of relying on the AI model's predictions, this method allows us to simulate the agent's actions. In this setup, the W and S keys control forward and backward movement, and A and D keys control the turning whether left or right depending on the input. The K key activates the shooting action. By simulating keyboard inputs to the same action format used by the learning algorithm, this function enables the agent to be tested within the environment under direct human control, making it easier to understand the mechanics or diagnose behavior issues while training.

5.3 Training, Results and Analysis

Training was done using the Proximal Policy Optimization (PPO) algorithm, a reinforcement learning method known for its stability and effectiveness in both continuous and discrete action spaces. Initially, when I began training the agent, the process took an extensive amount of time to complete, even with a relatively small number of steps (1e-6). To improve training efficiency, I implemented parallel training by increasing the number of environments to 9. For example: the agent took over 10 hours to complete just 2 million steps, when I implemented parallel training by scaling to 9 environments the runtime was reduced to 1.4 hours on average for the same step count, almost 6-7x faster than before. This allowed faster data collection and more efficient training.

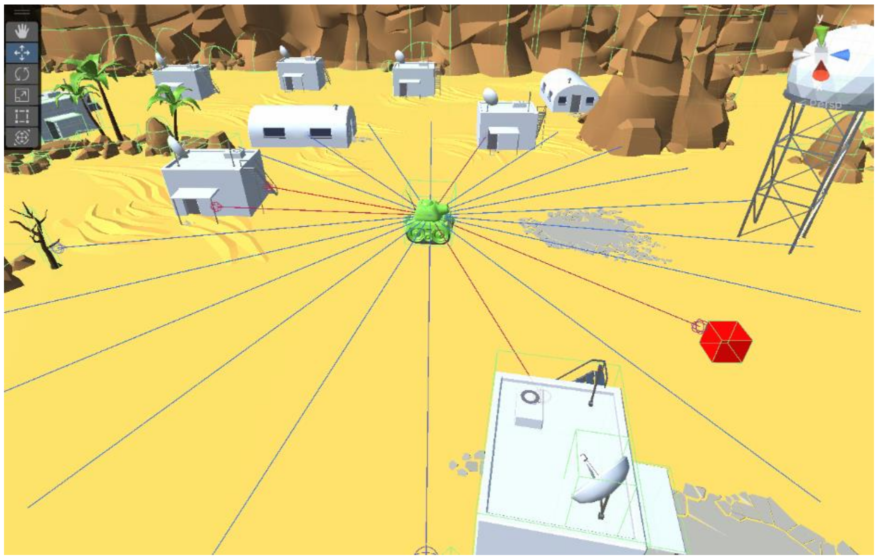


Figure 8 Agent's sensors

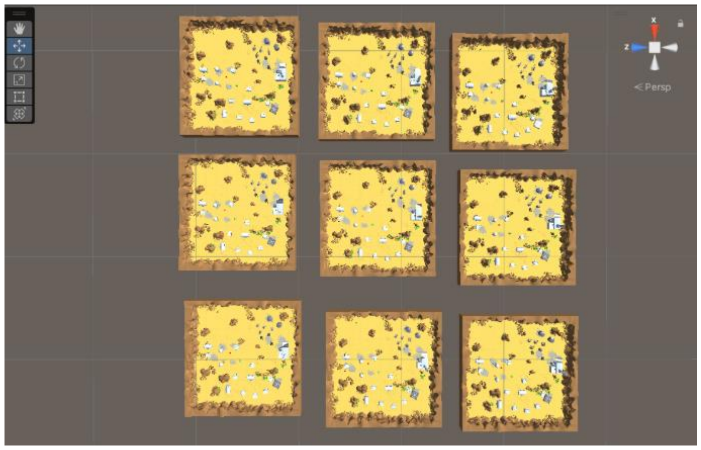


Figure 9 Multiple environments

Comparison 1

The graph displays my first (Run1_9) and best performing (Run6_9) training runs, with Run6_9 achieving significantly higher cumulative rewards due to a well-balanced combination of hyperparameters that optimized learning stability and efficiency compared to the first run.

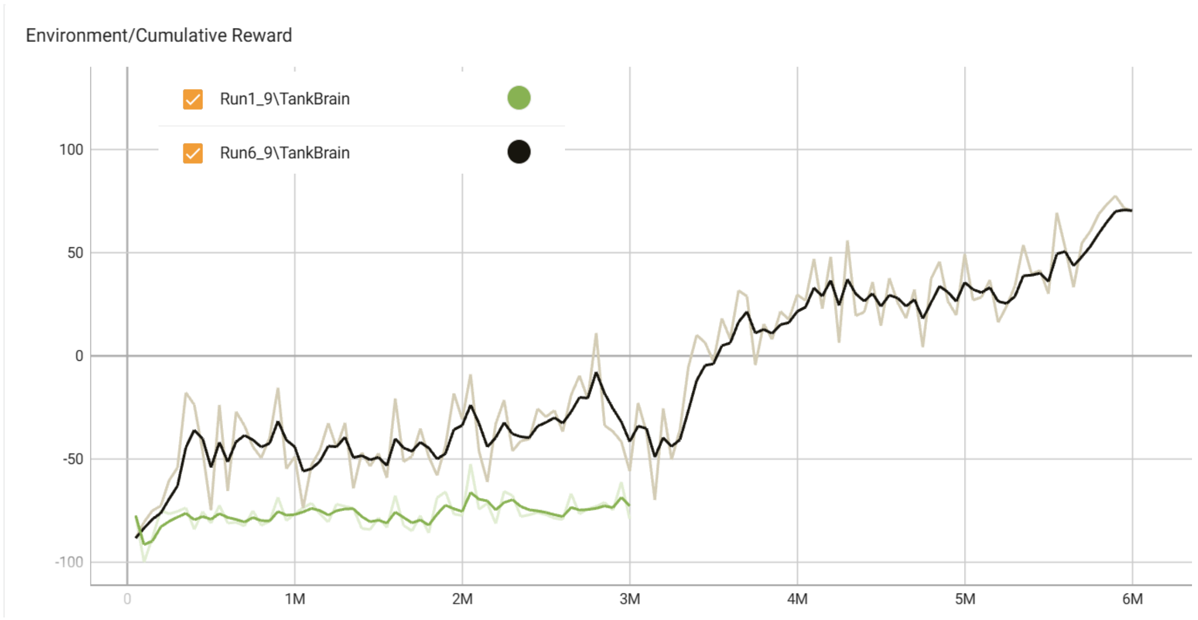
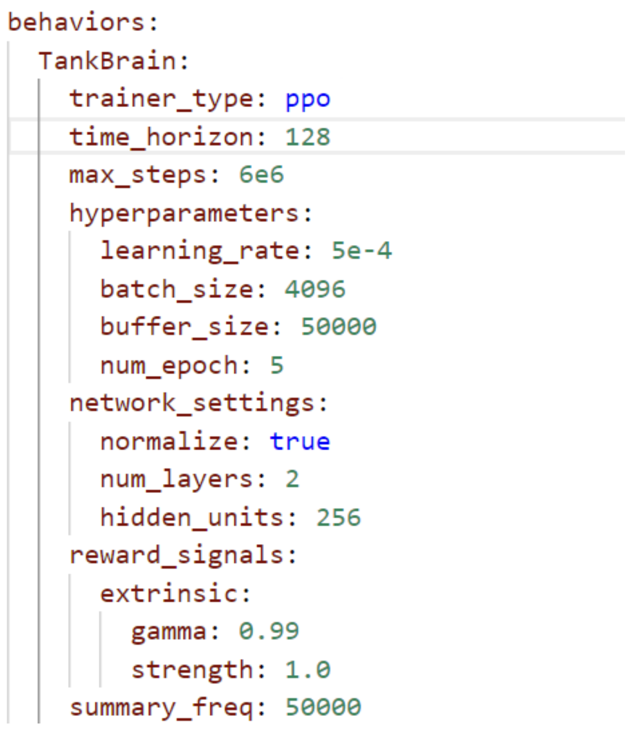
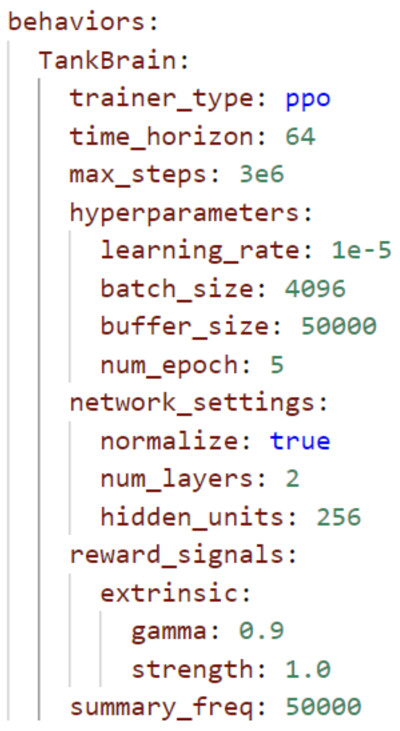


Figure 10 First and Best Training runs Cumulative Reward Graph



```
behaviors:
  TankBrain:
    trainer_type: ppo
    time_horizon: 128
    max_steps: 6e6
    hyperparameters:
      learning_rate: 5e-4
      batch_size: 4096
      buffer_size: 50000
      num_epoch: 5
    network_settings:
      normalize: true
      num_layers: 2
      hidden_units: 256
    reward_signals:
      extrinsic:
        gamma: 0.99
        strength: 1.0
    summary_freq: 50000
```



```
behaviors:
  TankBrain:
    trainer_type: ppo
    time_horizon: 64
    max_steps: 3e6
    hyperparameters:
      learning_rate: 1e-5
      batch_size: 4096
      buffer_size: 50000
      num_epoch: 5
    network_settings:
      normalize: true
      num_layers: 2
      hidden_units: 256
    reward_signals:
      extrinsic:
        gamma: 0.9
        strength: 1.0
    summary_freq: 50000
```

Figure 11 Run6_9 YAML Configuration file *(Best-Performing Model)*

Figure 12 Run1_9 YAML Configuration file

Run6_9 excelled because of its finely adjusted PPO setup, which achieved a balance between learning stability and efficiency. A stable policy update was ensured through the application of a moderate learning rate of 5e-4 and a batch size of 4096. Efficient reuse of data was attained through the 50,000-step experience buffer and 5 training epochs per batch. The 2-layer, 256-unit neural network with input normalization generalized complex behavior without overfitting, and the large time horizon (128 steps) along with a large discount factor (gamma=0.99) allowed the agent to effectively prioritize long-term rewards. As for the comparison with Run1_9, Run6_9's better performance resulted from key changes: a higher learning rate (5e-4 vs 1e-5) enabled faster convergence, an increased time horizon (128 vs 64 steps) and larger discount factor (gamma=0.99 vs 0.9) that improved long-term strategic planning, doubling the training duration (6M vs 3M steps) allowed more thorough policy optimization. These enhancements under the same network structure and batch sizes show how precise hyperparameter optimization can greatly enhance learning efficiency and overall performance.

Comparison 2

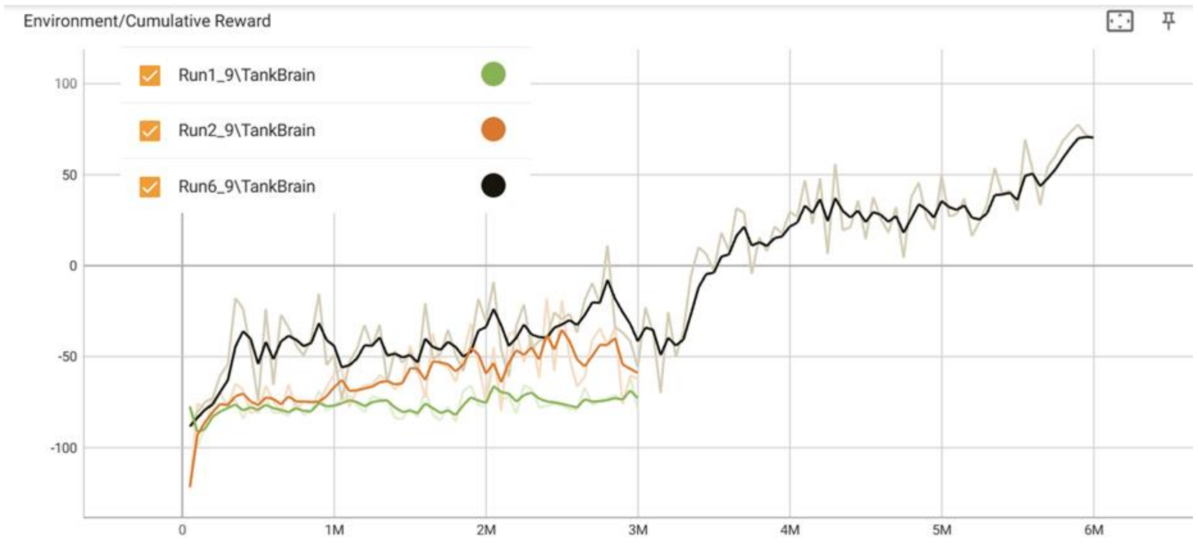
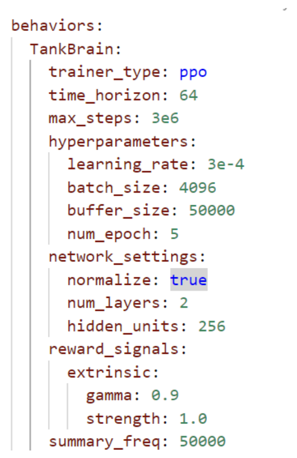


Figure 13 Run 1_9, 2_9, 6_9 Cumulative Reward Graph

Run2_9 demonstrates that learning rate alone does not suffice for peak performance. While its higher learning rate enabled faster convergence and intermediate reward compared to Run1_9, yet it still underperformed compared to Run6_9 due to two underlying deficiencies: short-circuited training time and its focus on immediate rewards. Run6_9 outperformance resulted from its balanced composition of long-term orientation, longer training time, and an optimized learning rate, confirming that bringing out peak PPO performance requires simultaneous tuning of learning dynamics, temporal planning, and training time.



```
behaviors:
  TankBrain:
    trainer_type: ppo
    time_horizon: 64
    max_steps: 3e6
    hyperparameters:
      learning_rate: 3e-4
      batch_size: 4096
      buffer_size: 50000
      num_epoch: 5
    network_settings:
      normalize: true
      num_layers: 2
      hidden_units: 256
    reward_signals:
      extrinsic:
        gamma: 0.9
        strength: 1.0
    summary_freq: 50000
```

Figure 14 Run2_9 YAML Configuration file

Comparison 3

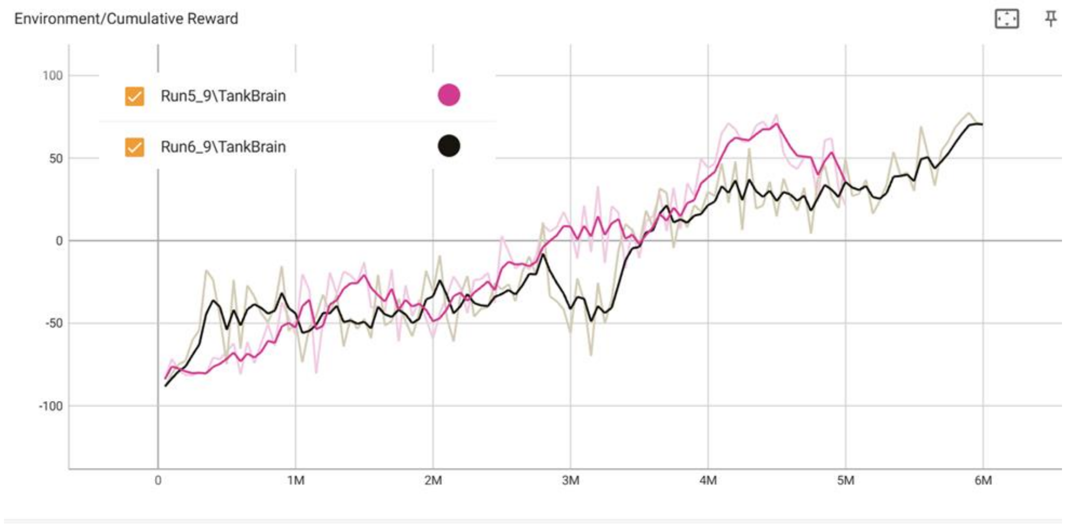


Figure 15 Run 5_9 and Run6_9 Graph

Even though both training runs share identical hyperparameters, it is Run6_9's additional 1M training steps that allowed it to surpass Run5_9's plateau, achieving higher final rewards. This confirms that even with identical hyperparameters, PPO often benefits from extended training to fully exploit learned strategies.

Episode Length

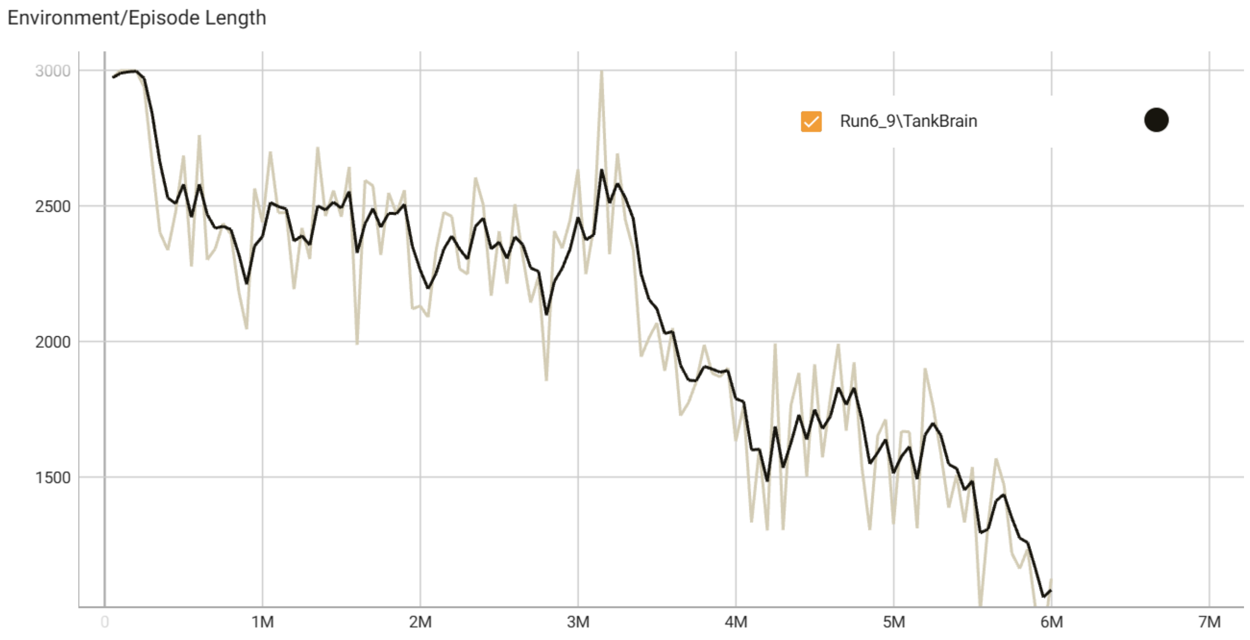


Figure 16 Episode Length of Run6_9 Graph

The Run6_9 Episode Length plot shows the agent's increasing performance, with episode lengths growing steadily from 1500 to 3000 steps over 6M training steps. The trend upward indicates the agent is learning to avoid early termination like collisions or failure and developing more intentional, long-term behavior, in direct alignment with its rising cumulative rewards. Longer episodes helped the agent better use its environment, earn more rewards, and show that its training was effective.

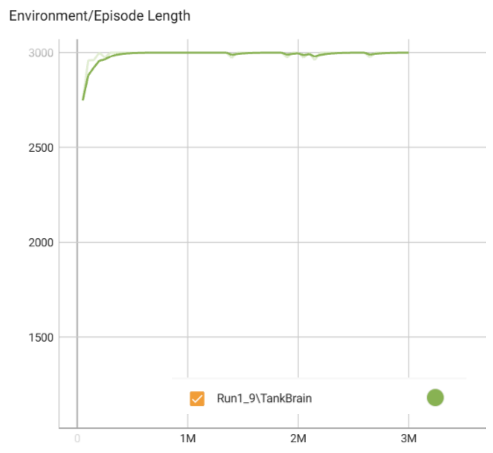


Figure 17 Run1_9 Episode Length Graph

The Run1_9 Episode Length plot indicates plateau performance, in which lengths plateaus at approximately 1500-2000 steps throughout its 3M training steps, indicating inherent limits in the ability of the agent to learn. This lack of progress significantly shorter than Run6_9's 3000-step episodes is directly attributable to Run1_9's poor reward efficiency and results from two basic deficiencies: its extremely low learning rate prevented meaningful policy improvements, while the short-term reward focus failed to incentivize long-term survival strategies. The flat trend line shows how bad hyperparameters can significantly constrain an agent's ability to construct complex behaviors, ultimately resulting in early and frequent task failure.

6. Conclusion

This thesis aimed to develop and evaluate a reinforcement learning-based artificial agent for a 3D tank game world using the Proximal Policy Optimization (PPO) algorithm. Through controlled experimentation, the agent successfully learned navigation and shooting strategies, demonstrating the viability of applying reinforcement learning to dynamic 3D game environments. The results revealed that agent performance significantly depended on hyperparameter optimization, particularly learning rate, discount factor (gamma), and training time. Run6_9, the highest-performing run, accumulated significantly higher rewards and longer episodes than before because it had a very balanced learning rate, high reward value given for long-term rewards, and more training time.

This thesis provided a comprehensive foundation in reinforcement learning principles. It began by introducing key RL concepts, including the Markov Decision Process, Q-learning, and the exploration vs. exploitation dilemma. Then, it delved into deep reinforcement learning (DRL), covering how neural networks, Deep Q-Networks (DQN), and policy gradient methods enable RL to scale to more complex tasks. A special focus was placed on PPO, which formed the core of the agent's learning algorithm in the experimental setup.

Then the thesis explored how reinforcement learning has been applied across various video game genres, ranging from classic arcade games to modern real-time strategy and open-world environments.

This research contributes to game AI in general in the sense that it shows how reinforcement learning can create versatile agents even though the training process was time-consuming and limited by hardware capabilities.

Beyond gaming, the ideas and methods used in this project can also help create smart agents for things like virtual simulations, training programs, or self-controlling robots. Reinforcement learning is a powerful tool for teaching systems how to make good decisions in complex situations.

The research also provides a foundation for many possible future projects, where those projects could be based on exploring multi-agent behaviors or even a more complex environment to further develop the agent.

In my thesis I have used AI to help me organize the content I wanted to write about, it also helped me fix some grammatical mistakes. During my practical work it helped me whenever I faced a technical error or whenever I needed a brief explanation of the results so I could understand the work better.

Acknowledgements

I want to express my deep appreciation and thanks to my supervisor, Dr Harangi Balázs, for all his help and support in this thesis. I am especially appreciative of his assistance when technical difficulties occurred. The project would never have been done without his leadership.

References

1. V. Mnih et al., “Playing Atari with Deep Reinforcement Learning,” *arXiv preprint arXiv:1312.5602*, Dec. 2013. [Online]. Available: https://doi.org/10.48550/arXiv.1312.5602
2. A. Kanervisto, C. Scheller, Y. Schraner, and V. Hautamäki, “Distilling Reinforcement Learning Tricks for Video Games,” *arXiv preprint arXiv:2106.11731*, 2021.
3. D. J. Richter and R. A. Calix, “QPlane: An Open-Source Reinforcement Learning Toolkit for Fixed Wing Aircraft Simulation,” *2021 IEEE Int. Conf. on Systems, Man, and Cybernetics (SMC)*, 2021.
4. M. Arango, “Deep Q-Learning Explained,” *Towards Data Science*, Jun. 20, 2018. [Online]. Available: https://towardsdatascience.com/deep-q-learning-explained-9258354a622
5. Z. Lin et al., “JueWu-MC: Playing Minecraft with Sample-efficient Hierarchical Reinforcement Learning,” *NeurIPS*, 2022.
6. H. Bi, “The Application of Reinforcement Learning Algorithms in Intelligent Games,” *J. Appl. Comput. Eng.*, vol. 54, no. 1, pp. 265–270, 2024. [Online]. Available: https://doi.org/10.54254/2755-2721/54/20241671
7. R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. MIT Press, 2018.
8. M. van Otterlo and M. Wiering, *Reinforcement Learning and Markov Decision Processes*. Springer, 2012.
9. C. Boutilier et al., “Planning and Learning with Stochastic Action Sets,” *J. Artif. Intell. Res.*, vol. 61, pp. 1–33, 2018.
10. C. J. C. H. Watkins and P. Dayan, “Q-Learning,” *Mach. Learn.*, vol. 8, no. 3–4, pp. 279–292, 1992.
11. Y. Hu and G. Yan, “Q-Learning for Single-Agent and Multi-Agent and Its Application,” *2022 2nd Int. Conf. Artif. Intell., Autom., High-Perform. Comput.*, 2022.
12. Q. Cai et al., “A Survey on Deep Reinforcement Learning for Data Processing and Analytics,” *IEEE Trans. Knowl. Data Eng.*, 2021. [Online]. Available: https://doi.org/10.48550/arXiv.2108.04526

13. P. Dayan and T. J. Sejnowski, "Exploration Bonuses and Dual Control," *Mach. Learn.*, vol. 25, no. 1, pp. 5–22, 1996. [Online]. Available: https://doi.org/10.1007/BF00115298
14. N. Le et al., "Deep Reinforcement Learning in Computer Vision: A Comprehensive Survey," *arXiv preprint arXiv:2108.11510*, 2021. [Online]. Available: https://doi.org/10.48550/arXiv.2108.11510
15. R. S. Sutton, "Integrated Architectures for Learning, Planning, and Reacting Based on Approximating Dynamic Programming," *Proc. 7th Int. Conf. Mach. Learn.*, 1990.
16. J. Schrittwieser et al., "Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model," *Nature*, vol. 588, pp. 604–609, 2020. [Online]. Available: https://doi.org/10.48550/arXiv.1911.08265
17. K. Nowosadko, "Self-explaining Neural Networks in Reinforcement Learning," *arXiv preprint arXiv:2403.04212*, 2024.
18. S. Elfwing, E. Uchibe, and K. Doya, "Sigmoid-Weighted Linear Units for Neural Network Function Approximation in Reinforcement Learning," *Neural Netw.*, vol. 107, pp. 3–11, 2018. [Online]. Available: https://doi.org/10.1016/j.neunet.2017.12.012
19. H. Chang and K. Futagami, "Reinforcement Learning with Convolutional Reservoir Computing," *Appl. Intell.*, vol. 50, no. 8, pp. 1–13, 2020. [Online]. Available: https://doi.org/10.1007/s10489-020-01679-3
20. K. Yan et al., "Multi-Step Short-Term Power Consumption Forecasting with a Hybrid Deep Learning Strategy," *Energies*, vol. 11, no. 11, p. 3089, 2018. [Online]. Available: https://doi.org/10.3390/en11113089
21. X. Qu et al., "Minimalistic Attacks: How Little it Takes to Fool a Deep Reinforcement Learning Policy," *arXiv preprint arXiv:1911.03849*, 2020. [Online]. Available: https://doi.org/10.48550/arXiv.1911.03849
22. V. Mnih et al., "Human-Level Control Through Deep Reinforcement Learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015. [Online]. Available: https://doi.org/10.1038/nature14236
23. D. Lee and N. He, "Periodic Q-Learning," *arXiv preprint arXiv:2002.09795*, 2020. [Online]. Available: https://doi.org/10.48550/arXiv.2002.09795

24. M. Sabry and A. Khalifa, "On the Reduction of Variance and Overestimation of Deep Q-Learning," *arXiv preprint arXiv:1910.05983*, 2019. [Online]. Available: https://doi.org/10.48550/arXiv.1910.05983
25. C. Wang et al., "Emergency Load Shedding Strategy for Microgrids Based on Dueling Deep Q-Learning," *IEEE Access*, vol. 9, pp. 19707–19715, 2021. [Online]. Available: https://doi.org/10.1109/ACCESS.2021.3055401
26. P. Ecoffet et al., "Policy Search with Rare Significant Events: Choosing the Right Partner to Cooperate With," *arXiv preprint arXiv:2103.06846*, 2021. [Online]. Available: https://doi.org/10.48550/arXiv.2103.06846
27. J. Guo et al., "Hindsight Value Function for Variance Reduction in Stochastic Dynamic Environment," *Proc. 30th Int. Joint Conf. Artif. Intell.*, pp. 2476–2482, 2021. [Online]. Available: https://doi.org/10.24963/ijcai.2021/341
28. I. Grondman et al., "A Survey of Actor-Critic Reinforcement Learning: Standard and Natural Policy Gradients," *IEEE Trans. Syst., Man, Cybern. C*, vol. 42, no. 6, pp. 1291–1307, 2012. [Online]. Available: https://doi.org/10.1109/TSMCC.2012.2218595
29. J. Bhandari and D. Russo, "Global Optimality Guarantees for Policy Gradient Methods," *arXiv preprint arXiv:1906.01786*, 2019. [Online]. Available: https://doi.org/10.48550/arXiv.1906.01786
30. E. Imani et al., "An Off-Policy Policy Gradient Theorem Using Emphatic Weightings," *arXiv preprint arXiv:1811.09013*, 2018. [Online]. Available: https://doi.org/10.48550/arXiv.1811.09013
31. J. Schulman et al., "Proximal Policy Optimization Algorithms," *arXiv preprint arXiv:1707.06347*, 2017. [Online]. Available: https://doi.org/10.48550/arXiv.1707.06347
32. C. Hu et al., "Reinforcement Learning with Dual-Observation for General Video Game Playing," *IEEE Trans. Games*, vol. 15, no. 2, pp. 202–216, 2023. [Online]. Available: https://doi.org/10.1109/TG.2022.3164242
33. H. Chai, "Reinforcement Learning Methods in Board and MOBA Games," *J. Appl. Comput. Eng.*, vol. 2, no. 1, pp. 883–890, 2023. [Online]. Available: https://doi.org/10.54254/2755-2721/2/20220556

34. B. Alghanem and P. Keerthana, “Asynchronous Advantage Actor-Critic Agent for StarCraft II,” *arXiv preprint arXiv:1807.08217*, 2018. [Online]. Available: https://doi.org/10.48550/arXiv.1807.08217
35. C. Berner et al., “Dota 2 with Large Scale Deep Reinforcement Learning,” *arXiv preprint arXiv:1912.06680*, 2019. [Online]. Available: https://doi.org/10.48550/arXiv.1912.06680
36. J. Li et al., “Open-World Reinforcement Learning over Long Short-Term Imagination,” *arXiv preprint arXiv:2410.03618*, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2410.03618
37. D. Silver et al., “Mastering the Game of Go Without Human Knowledge,” *Nature*, vol. 550, no. 7676, pp. 354–359, 2017. [Online]. Available: https://doi.org/10.1038/nature24270
38. K. Arulkumaran, A. Cully, and J. Togelius, “AlphaStar: An Evolutionary Computation Perspective,” *arXiv preprint arXiv:1902.01724*, 2019. [Online]. Available: https://doi.org/10.48550/arXiv.1902.01724
39. R. Lowe et al., “Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments,” *arXiv preprint arXiv:1706.02275*, 2020. [Online]. Available: https://doi.org/10.48550/arXiv.1706.02275
40. *Unity Asset Store* https://assetstore.unity.com/packages/essentials/tutorial-projects/tanks-complete-project-46209