



Hochschule für Technik
und Wirtschaft Berlin

University of Applied Sciences

Agent Performance Comparison using Reinforcement Learning in an Interactive Setting

Master Thesis

Name of the Study Programme

International Media and Computing

Fachbereich 4

submitted by

Laura Unverzagt

date:

Berlin, 17.03.2026

First Supervisor: Prof. Dr. Tobias Lenz

Second Supervisor: Prof. Dr.-Ing. David Strippgen

Affidavit

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit in allen Teilen selbständig angefertigt und keine anderen als die in der Arbeit angegebenen Quellen und Hilfsmittel benutzt habe, und dass die Arbeit in gleicher oder ähnlicher Form in noch keiner anderen Prüfung vorgelegen hat. Sämtliche wörtlichen oder sinngemäßen Übernahmen und Zitate sind kenntlich gemacht und nachgewiesen. Ich habe in der Arbeit keine KI-generierten Textpassagen oder Inhalte verwendet. Zudem versichere ich, dass ich keine anderen KI-basierten Tools verwendet habe, deren Nutzung der Prüfer / die Prüferin explizit schriftlich ausgeschlossen hat.

Academic integrity declaration

I hereby declare that I have written all parts of this thesis independently and that I have not used any sources or aids other than those stated in the thesis, and that the thesis has not been presented in the same or a similar form in any other examination. All verbatim or in-sentence copies and quotations have been identified and verified. I have not used any AI-generated text passages or content in the thesis. Furthermore, I assure that I have not used any other AI-based tools whose use has been explicitly excluded in writing by the examiner.



Berlin, 17.03.2026

(Ort, Datum, Unterschrift) | (Place, Date, Signature)

Gemäß Vorlage der HTW-Berlin / According to the template of HTW-Berlin

Contents

0. Abstract	5
1. Introduction	6
Reinforcement Learning	8
Exploration versus Exploitation	9
Model-Based versus Model-Free	9
Monte-Carlo and Temporal Difference Learning (TDL)	10
SARSA	10
Q-Learning	11
Deep Reinforcement Learning (DRL)	11
Multi-Agent Reinforcement Learning (MARL)	12
2. Related work	13
Relevance	13
Q-Learning and SARSA	15
Deep Reinforcement Learning	18
Other Examples of Reinforcement Learning	21
Multi-Agent Reinforcement Learning (MARL)	24
Other Systems using RL	29
Conclusion	30
3. Methodology and Implementation	32
Approach	32
Clementine and Beans	33
MARLClementine and MARLBeans	34
MARLClementinus and MARLBeansus	34
SARSA Algorithm	35
Environment and Infrastructure	36
States and Actions	37
Rewards	41
Reward Values	44
System Architecture and Support Systems	45
Testing Class	45
The JSON Reader	47
NPCs and Stat Scriptable Objects	48

Adaptability and Expandability of the System	49
Testing / Training	50
Exploration versus Exploitation during Training	51
Approach to Comparison and Expectations	52
Clementine and Beans	53
MARLClementine and MARLBeans	54
MARLClementinus and MARLBeansus	55
4. Results and Evaluation	57
Clementine and Beans	59
Comparison with Stationary Tasks	59
Comparison with Moving Tasks	60
MARLClementine and MARLBeans	61
Comparison with Stationary Tasks	61
Comparison with Moving Tasks	66
MARLClementinus and MARLBeansus	70
Comparison with Stationary Tasks	70
Comparison with Moving Tasks	71
5. Discussion	73
6. Citation	74

0. Abstract

This paper presents the comparison of six different Reinforcement Learning based agents using the SARSA algorithm. A short explanation and presentation of the most recent Reinforcement Learning research prepares for the methodology behind the showcased algorithms. Two single-agent algorithms, two decentralised and two centralised multi-agent algorithms are created and tested, using differing state-actions pairs in an adaptable and expandable test setting. Results are evaluated and compared to another, and future outlooks are shared with a short discussion of the outcome of the experiment.

1. Introduction

Whether it is a strategy game, an open world roleplaying game or a 2D platformer, video games of all genres require non-player characters (NPC). The most prevalent approaches to programming of NPCs, behaviour trees and finite state machines, are rigid in their design, difficult in their maintenance for programmers, and predictable for players of the game (Supli and Siqui, 2025).

Academics and companies alike have been searching for different algorithms and approaches to the design of NPCs to increase their adaptability, predictability and immersiveness. Therefore, many machine learning algorithms and theories have become a topic of interest in the context of gaming settings (Souchleris et al., 2023).

A commonly researched approach is Reinforcement Learning (RL). Although often tested and developed in academic settings using games such as ATARI 2600 (Souchleris et al., 2023), it also finds merit and testing grounds in games with professional leagues and thousands of players, with famous examples such as AlphaStar for the game StarCraft II (Vinyals et al., 2019). Even established tech and game companies, like Ubisoft or OpenAI have shown interest in RL and are researching its use in video game settings (Alonso et al., 2020) (OpenAI, 2019a).

Reinforcement Learning algorithms are far and wide spread, and have been developed for many years. Many researchers try to compare and create a profile for each algorithm, to understand which are best suited for each situation. Wender and Watson's (2012) research as an example compares four different Reinforcement Learning algorithm variations.

Many of the examples of current research use complex systems, trained using elaborate computing clusters for hardware, and deep neural networks for learning new strategies, like the AlphaStar agent (Vinyals et al., 2019). Outside of an academic or sponsored setting, this large scale of complexity and hardware requirements would not be feasible for a game studio with often little resources and little budget to spare.

This leads to the question of feasibility of Reinforcement Learning within games, and whether simpler systems without neural networks or multiple CPUs can successfully be implemented into an interactive setting.

Can even simple systems using Reinforcement Learning be used to train compelling NPCs that fulfil their goals? And would they work in games with multiple NPCs that each use Reinforcement Learning agents?

This paper attempts to further the research around these questions and presents an agent performance comparison between six different Reinforcement Learning agents, all using the SARSA algorithm.

Two of these are single-agent algorithms, two consist of a decentralised system and two of a centralised multi-agent system. All algorithms are created and tested using differing state-actions pairs and compared.

This approach has been taken to present the importance of appropriately chosen values in the Reinforcement Learning system, and to show how different agents within the same system can function and learn.

Similar to Glavin and Madden's (2018) research, policies of different learning intervals have been saved and compared to another. The focus of comparison lies primarily on the last learnt policy, considering the success in the completion

of the agent's goal, the spread of learning across the table as well as the rate at which the agent has learnt to complete its goal.

The following texts give a brief overview of the most relevant Reinforcement Learning strategies, followed by the second chapter about relevant research within the field, like the previously mentioned AlphaStar and more. The third chapter dives into the methodology and implementation of the described agents and outlines the system around them. This is followed by the fourth chapter, which presents the learning results of all agents and compares them as described before. Lastly, a short discussion completes the presented research and offers an outlook to possible future research.

Reinforcement Learning

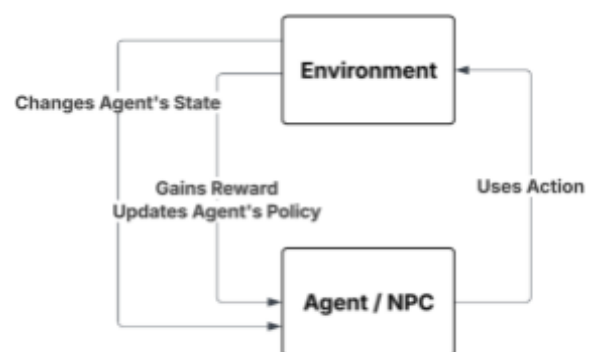
Reinforcement Learning (RL) is an approach to machine learning based on the Markov Decision Process (MDP).

MDP, shortly summarised, is a decision-making model that is used when outcomes are partially random and partially controlled. (Souchleris et al., 2023)

“The four principal components of a reinforcement learning system are policy, reward signal, value function, and optionally, the environment model.”

(Sukmana et al. 2024, p. 158)

An agent, in video game settings often represented as an NPC, is placed in an environment it can interact with. To decide its actions depending on its current state in the environment, it



Img.1 Typical Reinforcement Learning model

follows a policy. To define and learn from actions taken, and therefore update its policies, a reward is given to the agent, often upon entering a new state. The significance of the reward is defined by the value function, which can consider current, previous or future actions in its evaluation.

This cycle of decisions, rewards and improved policies is the underlying principle that defines Reinforcement Learning. The Q-Table in this context is considered the data structure containing policies of all saved state and action pairings.

RL algorithms can be differentiated by their learning approaches. On-policy algorithms such as SARSA learn from their actions and actual interactions with their environment. Meanwhile, off-policy algorithms such as Q-Learning rely on predictions and the outcome of the best action in any given state that could have been taken.

Exploration versus Exploitation

The dilemma in many Reinforcement Learning algorithms is the consideration of exploration versus exploitation.

While it is significant to rely on learnt policies and 'exploit' the best possible action in a given state, the reliance on given information and previously learnt choices bars the possibility of better, unknown opportunities to the algorithm.

Finding a balance between exploring unknown actions or new possibilities and exploiting already known paths is a still investigated area of Reinforcement Learning.

Model-Based versus Model-Free

Most algorithms can be differentiated by their knowledge of the environment and consequences of actions. Model-based algorithms in this context are those

that have access to or build a model of the environment and its dynamics. Model-free algorithms meanwhile are those that have no knowledge or only very little knowledge of the environment and instead learn it from their actions.

Monte-Carlo and Temporal Difference Learning (TDL)

Monte-Carlo methods and Temporal Difference Learning methods are Reinforcement Learning techniques that estimate the most effective policy for an agent.

Monte-Carlo methods observe every step in a run (also called episode) to receive the average reward of all steps taken.

TDL on the other hand observes only singular actions taken before returning a reward.

Both do not require information about the environment of the agent, as both learn from the actions taken and gained rewards.

SARSA

SARSA, which stands for “State-Action-Reward-State-Action”, is a model-free, on-policy Reinforcement Learning algorithm. Following the pattern described in its name, SARSA decides its actions dependant on its current state, before gaining the respective reward and moving to the next state, thereby completing the cycle. As described by Wiering and Otterlo (2012), SARSA algorithms are best used in changing environments, as they learn from the actual changes and choices made. The algorithm is as follows:

$$Q(s,a) \leftarrow Q(s,a) + \alpha * (r + \gamma * Q(s',a') - Q(s,a))$$

Where $Q(s,a)$ is the current value, α is the learning rate, r is the reward, γ is the discount factor (meaning the indicator of importance for future rewards), and $Q(s',a')$ is the next state and action pair.

Q-Learning

In contrast, Q-Learning is a model-free, off-policy algorithm. Unlike SARSA, Q-Learning focuses on the possible future rewards of a taken action. It considers its current state merely to compare its possible future states and future rewards for each, before selecting their action. The great advantage of Q-Learning is its exploration-insensitiveness, as no matter its exploration policy, it always finds the best action for its current state (Wiering and Otterlo, 2012). The algorithm is as follows:

$$Q(s,a) \leftarrow Q(s,a) + \alpha * (r + \gamma * \max Q(s',a') - Q(s,a))$$

Where $Q(s,a)$ is the current value, α is the learning rate, r is the reward, γ is the discount factor, and $\max Q(s',a')$ is the best estimated value in the next state.

Deep Reinforcement Learning (DRL)

DRL is a combination of deep learning and Reinforcement Learning (Yang et al., 2025). Lecun et al. (2015) described deep learning as follows:

Representation learning is a set of methods that allows a machine to be fed with raw data and to automatically discover the representations needed for detection or classification. Deep-learning methods are representation-learning methods with multiple levels of representation [...] (p.1)

But while deep learning relies on supervised learning (Lecun et al., 2015), DRL with its base in RL is capable of building its own model and policies. Using the neural networking theory of deep learning, DRL utilises the reward system of Reinforcement Learning to feed new information into the neural network to improve the policies of the DRL agent (Yang et al., 2025).

Multi-Agent Reinforcement Learning (MARL)

Multi-Agent Reinforcement Learning is a system that expands and often relies on DRL algorithms. The difference to DRL is that MARL interacts with multiple learning algorithms, and therefore multiple agents.

The core of MARL is the interaction of multiple agents within the same environment. This interaction can be done adversely, cooperatively or as a mixture of both.

In a decentralised system, each agent learns its own policy. This means an agent has to consider other agents within the environment and react to their changing behaviour, while often considering the learning processes of other agents' policies. This by nature results in a non-stationary environment for all agents.

Meanwhile, in a centralised system, all agents use the same policies instead of their own and gain a joint reward. All agents' actions and states are considered in the evaluation of updated policies.

The consideration of all agents at all times, no matter if centralised or not, is the reason for MARL's success in changing environments (Yang et al., 2025).

2. Related work

This section describes the relevance of Reinforcement Learning in the past, present and future, before diving into case examples and prominent research of the field. The focus of the following is an overview of relevant algorithms, systems and paradigms in the field of RL, with a special focus on gaming contexts.

Examples of Q-Learning and SARSA algorithms are described, followed by relevant research around DRL. A broader spectrum of other RL algorithms is presented, including the controversial AlphaGo agent, before disserting some of the most famous examples of DRL and MARL algorithms like AlphaStar. Other examples of the use of RL strategies within gaming contexts, outside of the addressed algorithms and systems, are showcased after, followed by a summary of the chapter.

Relevance

According to Supli and Siqi (2025), the typical approach to video game AI is the utilisation of behaviour trees or finite state machines. Yet in gaming contexts, “There is a continuous need to study and employ machine learning (AI) techniques and algorithms to research games due to the enormous potential and role of machine learning in game creation and development.” (Supli and Siqi, 2025, p.2).

In their study, Supli and Siqi have shown the advantages of machine learning and RL approaches in the future of video games. They have created a game with the Unity Engine that utilises both behavior trees and machine learning, to evaluate

motivation and player engagement as well as execute a comparative analysis between the behaviour tree and the machine learning AI.

Their results prove that although machine learning might not be suitable for all types of games, specifically dynamic and interactive game genres like strategy games benefit from machine learning approaches and the learning aspect of RL (Supli and Siqui, 2025).

Reinforcement Learning has been a topic of discussion in scientific and gaming contexts for many years. “In the past one and a half decade [sic], the field of reinforcement learning has grown tremendously.” (Wiering and Otterlo, 2012). Already in 2012, scientists realised the potential of RL and continued the development and research of the field.

Souchleris et al. (2023) evaluated the current state of RL and found it used by multiple companies, e.g. Google DeepMind with AlphaGo for the game Go (Silver et al., 2016), or Microsoft using RL algorithms to reach the highest possible score to be accomplished in the game Pacman. The research of Souchleris et al. determines that the terms “reinforcement learning” and “games” have been used together in over 25% of publications between 2012 and 2022, further confirming the relevance of RL algorithms in gaming contexts even nowadays.

Although they predict the development of RL moving further toward general intelligence, Souchleris et al. speculate that RL algorithms will stay relevant within essential gaming contexts (Souchleris et al., 2023). The current trend of research in the RL field shows this relevance increasingly.

RL does not only show relevance in gaming though. Real world applications using RL systems to solve challenges are not unheard of. For example,

self-driving cars can take inspiration from racing game RL implementations and learn driving in the real world (Souchleris et al., 2023).

Q-Learning and SARSA

Both SARSA and Q-Learning have played a large role in the development of Reinforcement Learning. At their core, the algorithms have existed for many years, but have been developed and used in many differing systems and current research still. This chapter shows some of the research around the two algorithms and explains their relevance in newly developed versions and systems that stretch themselves throughout all following chapters of this paper, from inclusions in DRL systems to implementations in later illustrated RL agents.

Reinforcement Learning has reached different gaming spaces within recent years. A good example of an RL implementation, showcasing its relevance, has been made by Sukmana et al. (2024), who created an adaptable RL NPC for their game called “CyberHero”.

CyberHero was specifically created for the purpose of teaching security awareness and password safety, additionally presenting how games can be used for educational purposes. The role of Q-Learning was to expand intelligence and adaptability of NPCs with the goal of increasing the gameplay experience, causing a trickle down effect of maximizing engagement which grows learning possibilities for players.

Their study focused on RL and its implementation in individual NPC behavior and presents how the variability and unpredictability of NPC actions can be increased by Q-Learning. Using Q-Learning, the NPCs were taught to move and shoot within the game's space, showcasing more flexible and engaging

behaviour in comparison to hard coded approaches like the above mentioned finite state machines.

The success of Sukmana et al. (2024) selecting Q-Learning for their research offers the question of preference in algorithms. Every algorithm within the Reinforcement Learning field fulfils a different role, as performance heavily relies on chosen parameters, selected environments, selectable action options and even game genres. The question which algorithm is more efficient, more successful or quicker to learn has been studied by many researchers:

Riady et al. (2024) compared Q-Learning and SARSA, specifically the importance of their chosen discount values and learning rates, as they have done detailed tests for their parameter combinations. They have concluded their tests in a so-called Javanese chess game, a type of two player game on a 3x3 grid with similarities to chess. During their research, they have tested combinations of learning rates (0.1 to 0.01) and discount factors (0.9 to 0.1) for both Q-Learning and SARSA and come to the conclusion that their differing values drastically change the outcome of the learning agents and their win rates. Overall, the comparison of the SARSA and Q-Learning agents showed that their performance did not only rely on their values, but also on their position in the game, as the SARSA agent outperformed the Q-Learning agent with winning rates of 59.6% against 52.8% when playing white, while the Q-Learning agent outperformed the SARSA agent with winning rates of 28.4% against 28% when playing black.

Their research proves the impact and differing results an RL agent can have depending on their algorithms configuration. In the case of Javanese chess, the preference appeared to be on the side of the player playing white, and although SARSA has been slightly outperformed on the black side, it overall has a much

greater win rate on the white side in comparison to Q-Learning. However, this preference is not only reliant on the SARSA algorithm, but also the game or environment it is being tested in.

Interestingly, a similar study comparing both algorithms has been conducted by Wender and Watson (2012). Their research compared both the one step Q-Learning and one step SARSA to each other, but also considered the $Q(\lambda)$ and $SARSA(\lambda)$ algorithms by Watkins (1989) in their comparison, which use eligibility tracing. Their specific focus was on the micro-management of combat units in the real-time strategy game StarCraft: Broodwar, an extension to the game StarCraft which has also been used for research involving the later mentioned AlphaStar agent (Vinyals et al., 2019). Their results are quite interesting, as they show that all four algorithms succeed in the learning process, win easily when using their best policy, have nearly reached the optimal possible policy and have learned nigh equally quickly. Yet despite this, the one step Q-Learning and one step SARSA outperformed the Watkins' algorithms using their best policy.

Their research additionally has shown the trade-off between speed and success during the learning process, as stated by the authors themselves: "Our experiment over half the number of episodes showed that a quicker learning curve is indeed possible but currently comes at the price of a decrease in obtained reward" (Wender and Watson, 2012, p.408).

Their work shows that, in their chosen context, Q-Learning and SARSA algorithms are not that different in their results, even in comparison to their further developed counterparts by Watkins. Comparing algorithms and understanding the importance of their differences is how new systems and further developments in the RL field can be made.

Zhao et al. (2016) presents one of these developments in their research, as they have used SARSA to develop a DRL algorithm called deep SARSA learning. Combining SARSA and deep learning strategies, they tested their new creation in two games from Atari 2600 called Breakout and Seaquest.

Similarly evolved is the deep Q-Learning strategy, a combination of Q-Learning and deep learning. Zhao et al. (2016) compared their creation, and found once again that deep SARSA learning surpasses deep Q-Learning in multiple factors. Although deep SARSA learning achieves higher scores and faster convergence in both games, it takes longer to reach these as deep Q-Learning succeeds in the early stages of learning a game.

Often algorithms combine to new approaches, like in the example of the work by Zhao et al. Both deep SARSA learning and deep Q-Learning derived from machine learning strategies as well as Reinforcement Learning strategies, creating two new algorithms that are part of the field of “Deep Reinforcement Learning”. How relevant DRL has become in current gaming contexts will be further explained in the following section.

Deep Reinforcement Learning

The importance of continuing the Reinforcement Learning strategies and developing areas in the field of DRL have shown themselves in the works of Graepel et al. (2004) and Oh et al. (2022).

Graepel et al. (2004) used a SARSA algorithm to create an AI for the commercial game *Tao Feng*, an at the time well known fighting game. Their goal was to prove that Reinforcement Learning can be used to learn behaviours within fighting games. Yet, their work already struggled at simple steps such as implementation

of their agent, due to the game system's architecture. Issues continued throughout the learning process, and although they were successful in their approach to create an agent capable of learning fighting game movement and actions, their work halted at anything beyond simple learning. Specifically, the competitive aspect of the game has been disregarded in the study. As stated by the authors themselves, "an important aspect of human fighting game play involves timing which is hardly captured by the MDP model and requires explicit representation of time" (Graepel et al., 2004, p.8), the MDP model in question being their agent.

Yet what was not able to be achieved by Graepel et al. in 2004 can be done with technology and algorithms like DRL nowadays.

The study by Oh et al. (2022) also attempted the creation of an AI for a commercial fighting game called *Blade & Soul*. Using DRL instead of a SARSA, their goal was the creation of an agent capable of beating human players. According to Oh. et al., the challenges of fighting games, namely their large pools of actions, the following dependencies tied to actions and the imperfect amount of information, have been responsible for the lack of an agent surpassing humans before their research.

Still, their agent has surpassed these challenges. By creating different rewards for different approaches (aggressive, balanced or defensive), the agent was able to gain a win rate of 62% against professional players.

Although algorithms like SARSA remain relevant and often essential in modern systems, others like DRL have proven their importance. Beyond that, systems and algorithms like DRL have been studied and used not just by universities, but also by industry researchers.

Alonso et al. (2020) from the game company *Ubisoft* have used RL to improve already existing systems for NPCs. They explain the weaknesses of current systems in NPC movement. Many games use so-called navigation meshes (NavMeshs) which are graph representations of a map. This approach allows easy movement of NPCs, but often players have abilities that exceed the information within a NavMesh, leading to unrealistic or predictable NPC movement. The technical inability to adapt the NavMesh or in many cases, the time restraints within video game development do not allow adjustment of NPC movement in many scenarios.

Alonso et al. (2020) used Deep RL to instead teach NPCs to adapt their movement. No matter the movement abilities given to NPCs, in this research the NPCs found the quickest paths from different starting and end points. The NavMeshs were not able to work with the abilities that the RL algorithm was able to implement in its movement.

Their research proves the necessity and feasibility of RL in games in the future, not just for player engagement or interactability with games, like in the research of Sukmana et al. (2024), but also to improve already existing systems from a technical standpoint.

This fact does not diminish similar successes of DRL in areas like adaptability, predictability or even believability (i.e. the ability of the actor to simulate human behaviour), especially in comparison to the established systems like finite state machines or behaviour trees.

Servat and Mohamadi (2023) have continued the trend of DRL in gaming contexts. Although, their goal was not improvement of existing systems or mastery over human skill, but instead comparison to alternate approaches to

NPCs. Focussing on realism, immersion and believability of NPCs, Servat and Mohamadi compared DRL to hard coded NPC approaches.

For this, they created two NPCs, both tasked to hit a target. The first NPC was hard coded, while the second was created using the so-called “Proximal Policy Optimization (PPO)” (Schulman et al., 2018), a form of DRL.

Their most impressive result is the believability of the DRL agent. They have noticed not just less predictability within their agent, but also unexpected advantages caused by RL. Attempting to make it more human and believable caused their agent to miss targets on multiple occasions, but more impressively caused the agent to hesitate before shooting, which is innately human behaviour. The comparison to the hard coded approach, which showed perfect aim and no hesitation, presented the stark difference in human-like behaviour.

Not only are RL agents capable of surpassing human skill, but they are also capable of presenting human behaviour and letting NPCs appear flawed, making them more believable. Combined with systems like Alonso et al.’s (2020), designed to improve existing NPC systems, RL is capable of changing the approach to NPC creation and capabilities of current times.

Other Examples of Reinforcement Learning

The evolution of RL based algorithms and creation of new systems has been explored for a long time.

Merrick and Maher (2006) as an example have created so-called motivated Reinforcement Learning agents (“MRL agents”) in their study, which use intrinsic rewards to calculate an “intrinsic motivation signal” with the purpose of directing NPCs to interesting in-game events. This is done with a focus on

adaptability of NPCs to different environmental changes, specifically considering Massively Multiplayer Online Role-Playing Games (MMORPGs). Therefore, their system was tested in the commercial virtual world called *Second Life*.

Their agent controlled a vendor NPC within the game and showcased new behaviour within its movement between areas of the learning environment. It learned to include new destinations within the environment and due to its adapted movement changed variables of the environment, e.g. the movement of a normally stationary item to a new location due to a colliding movement path. Merrick and Maher (2006) state that the adaptability and evolution possible of their MRL agents provide realism to the gaming experience, as their agents actions and movements can change not just themselves, but also the environment.

Another example of the evolution of RL algorithms is from Ponce and Padilla (2014), who have used a hierarchical Reinforcement Learning method called “MaxQ-Q” to attempt the creation of a more realistic and natural appearance of NPCs. To do so, they consider two main factors: predictability and “natural humanness” (Ponce and Pandilla, 2014).

To evaluate their success, two experiments with follow-up surveys were designed. The NPC, using the MaxQ-Q algorithm, was compared by users to an NPC with a finite state machine, to assess its predictability. Another experiment evaluated the “natural humanness” in multiple test runs by players with the NPC using the MaxQ-Q algorithm.

Their predictability survey results proved a 52.5% less predictable outcome than in comparison to most NPCs so far, “assuming that finite state machines

[are] the most preferred and implemented AI-based video game technique” (Ponce and Padilla, 2014, p.180).

Their survey on the realism of NPCs shows that according to users, the NPC, using MaxQ-Q, appears like a human, does not act randomly and adapts to the user's gameplay.

Once again, it is shown that Reinforcement Learning strategies are proving more human, less predictable and more interesting to users. But RL algorithms can do more than create realistic environments or appear human. Many RL algorithms are part of agents that surpassed human skill levels.

The most notable example of the success of RL algorithms is the human experts surpassing AlphaGo, created by the Google DeepMind Team consisting of Silver et al. (2016). Using two “deep neural networks” called “value network” and “policy network”, they trained their agent with a combination of supervised learning and Reinforcement Learning. Using the Reinforcement Learning strategy called “policy gradient reinforcement learning” (Williams, 1992) (Sutton et al., 1999), they trained the policy network with “self play”.

Self play learning is one of the most used strategies to teach RL algorithms better policies. It is the act of letting an agent “play” against themselves, therefore learning from their own actions. In the case of AlphaGo, the most current network acted as the player, while the opponent was represented by a randomly sampled opponent from previous iterations of AlphaGo.

Their success is shown in their results: AlphaGo “achieved a 99.8% winning rate against other Go programs, and defeated the human European Go champion [Fan Hui] by 5 games to 0” (Silver et al., 2016, p.484). Not only did their agent surpass other Go programs near perfectly, it also outperformed one of the best Go players known at the time.

Many of the previously mentioned approaches in this chapter have used model-free Reinforcement Learning strategies. Yet, there are also model-based approaches shown within gaming contexts.

Kaiser et al. (2024) used image evaluation in their research to predict frames of video games. Instead of the commonly used model-free approach as mentioned above by multiple authors like Alonso et al. (2020), they decided to test a model-based video prediction approach, to test whether a model-based approach could learn to play a game quicker than a model-free RL.

For their testing, they used the Atari Learning Environment (ALE) and created the RL approach “SimPLe”. Their results have shown that their RL approach, due to the raw pixel observation, learns their effective policies quicker than model-based approaches. The drawback of this is lower results in game wins in comparison to model-free methods, as well as less constant performance between different runs (Kaiser et al.).

Although model-based approaches like the research of Kaiser et al. can be quicker to learn and find merit in many areas of study, numerous researchers decide to further the research of model-free approaches due to the advantage of flexibility in inconsistent or changing environments. The mentioned drawbacks of lower win rates and fluctuating performance reliability are additional factors.

Differing problems require differing solutions, and each algorithm within the RL field has its advantages and drawbacks. Systems like the above mentioned AlphaGo require multiple different algorithms, systems and approaches with differing focal points. An unmentioned problem so far is the approach in situations with multiple NPCs in the same often-changing environment. To address this, researchers have taken inspiration and created so-called

Multi-agent Reinforcement Learning agents to address spaces with multiple agents.

Multi-Agent Reinforcement Learning (MARL)

In the context of an explanation of DRL, Yang et al. (2025) described the evolution of MARL as follows: “The need for an adaptive decision-making framework, together with the complexity of addressing multiple interacting learners, has led to the development of multiagent RL [...]” (p. 4)

MARL’s development was “primarily driven by the extension of [D]RL techniques to multi-agent settings” (Yang et al., 2025, p. 6) and carries its import into present studies. Li et al. (2025) describe MARL’s importance and research as follows:

While DRL has demonstrated significant success in video games, existing applications primarily focus on single-agent scenarios. In contrast, most modern video games involve multiple agents interacting cooperatively or competitively. Consequently, researchers and developers have turned to utilize MARL to develop AI systems that can effectively manage and optimize the interactions between multiple agents in games. (p. 874)

As mentioned above, the core of MARL is the interaction of multiple agents that use RL strategies. Often researchers use multiple angles and approaches to create a system containing multiple algorithms that work together to create the MARL. Whether it is DRL algorithms like PPO, TDL approaches or learning strategies like supervised or hierarchical learning, the combination of systems and approaches of their implementation define the efficiency and success of every MARL system in reaching their respective goals.

One of the most notorious examples of a MARL algorithm has been created by Vinyals et al. in 2019. AlphaStar is an agent created to compete against competitive players in the real time strategy game StarCraft.

Although RL algorithms are not the only part of AlphaStar, Reinforcement Learning has played a vital role in its creation. Training of the AlphaStar agents began with supervised learning, before RL algorithms were introduced to the system; a strategy previously encountered in the learning process of AlphaGo. Using the League training MARL algorithm, AlphaStar managed to best most human players, ranking above 99.8% of all active human players within the past months of testing in the game (Vinyals et al., 2019).

Its undeniable success is credited to its combination of algorithms and learning approaches that create the MARL agent, including the off-policy algorithms TD(λ)¹⁶, the DRL V-trace (Espeholt et al., 2018) and a self-imitation algorithm (Oh et al., 2018) called UPGO, made by Vinyals et al.

Alphastar is one of many great examples of successful Reinforcement Learning implementations in the MARL field that have proven their effectiveness and success. But AlphaStar is not the only success story worthy of note, and real time strategy games are not the only genre of games used as a testing field for MARL agents.

The subgenre of multiplayer real-time strategy games called *Multiplayer Online Battle Arena* (MOBA) (Ye et al., 2020) has been infamous for researchers and creators of MARL agents. Noteworthy MOBA games are *League of Legends*, *DOTA 2* and *Honor of Kings*, many of which have been used by researchers as shown below.

The OpenAI team around Berner et al. (2019a) as an example has created the first RL agent using MARL that defeated the world champions of the MOBA

game DOTA 2. Their agent used the PPO DRL algorithm (Schulman et al., 2018) to train the agents' policy. Although the team extracted some of the game's complexity by hard coding actions, their agent already bested some of the best human players of the game in the world during 2018 competition *The International*. The OpenAI team even believed that this performance could have been improved without hard coded scripts. Additionally, they have tested their AI during a multi-day online showcase, where out of 7,257 games only 42 were lost, resulting in a 99.4% win rate against human players of multiple skill levels.

Ye et al. (2020) tackled the topic of MARL similarly to the OpenAI team and used the game Honor of Kings to propose their "learning paradigm", using many RL approaches like DRL, MARL and Monte-Carlo, as well as neural networking approaches. Yet, as described by the authors themselves, "The major difference between our work and OpenAI Five is that the goal of this paper is to develop AI programs towards playing full MOBA games." (Ye et al., 2020, p. 3).

The OpenAI team has limited the total experience of DOTA 2 by reducing the amount of playable characters (17 out of 117), removing specific items within the game and, as previously mentioned, hard coding sections of the game like item or ability selection. These choices were made to avoid game complexity for the agent.

The goal of Ye et al. (2020) was to expand the previous work of the OpenAI team and deal with many of their issues, as an example the scalability-related issues caused by the attempt to increase the hero pool used by OpenAI. (OpenAI, 2019b)

They have tackled these limitations and created systems to implement the previous limitations in their research. While many similar approaches have been taken by Ye et al., like the use of PPO, they managed to expand and suggest a

system (or “learning paradigm”, as described by the authors) usable to many MOBA games in general in the future. With a similarly high win rate as the OpenAI agent of 95.2% in a total of 42 matches against professional players of the game, their approach is considered a success and expansion to previously suggested systems.

Despite many researchers' successes in the creation of RL and specifically MARL systems, many require extensive resources and time for training. Many human surpassing agents, including DRL and MARL agents like AlphaStar or AlphaGo, require supervised learning to improve their policies. Often they use strategies like self play to further their learning process, or they learn from human demonstration.

Zhao et al. have instead suggested another approach called “hierarchical learning” and tested it in a MARL setting using PPO (Schulman et al., 2018) and two further DRL algorithms. Their work showcases how much MARL systems are capable of outside of the human context: “Using proper reward shaping, we trained agents with a variety of offensive and defensive styles. In particular, we trained an agent that can assist the teammate player to achieve better scoring and ball possession.” (p. 6)

The work of Zhao et al. is another example of different training strategies, how MARL systems can learn and work together as well as how research in MARL remains relevant to explore new systems and learning strategies.

The field of MARL is not new, yet many aspects are still unexplored. Due to the history of MARL, many systems rely on DRL. Although skill levels above human capabilities have been reached by multiple algorithms, MARL can still struggle with the larger grasp of a complex system, as the example of OpenAI and the research of Ye et al. (2020) surrounding the totality of a MOBA games’

complexity has shown. MARL is a highly relevant field with many angles to explore in the near future.

Other Systems using RL

Many of the previous studies focused on the aspect of Reinforcement Learning itself, whether this has been to imitate human behaviour, surpass human skill or improve systems' efficiencies and successes. Some researchers instead use the self learning strategies of Reinforcement Learning to support systems within gaming contexts, like balancing strategies or learning aspects for players.

Glavin and Madden (2018) as an example, have created and tested a skill-balancing mechanism for NPCs called Skill Experience Catalogue (SEC), which is based on Reinforcement Learning. During the learning process of the RL using NPC, they store the current policy of different learning intervals for later use as "milestones". Upon completion of the NPCs training, the set milestones will be referred to during games against players. Should players perform better or worse than the AI, the difficulty will be adjusted by choosing different learning stages and referring back to the saved milestones.

Their focus was on only one aspect of the NPC, in their case shooting at targets, but their system is capable of expansion in future research. Glavin and Madden have used RL to improve player experience and have shown that RL can be used to impact players directly during their gaming experience.

Another example of the creative uses of RL was made by Tan et al. (2011). They have created algorithms inspired by Reinforcement Learning to adjust skill levels to the players of a game. Although they did not directly use a RL algorithm, they have been inspired by the idea of the self learning aspects and its

adaptivity. Their focus of the study is scalability and dynamic difficulty scaling of their game AI. Although they are not using known RL algorithms, their study is especially interesting considering their context. RL systems have been tested for believability, humanness, adaptability, performance against humans and effectiveness, among other concepts. This paper instead focuses on adaptation of balancing within a game for the purpose of player satisfaction, similar to Glavin and Madden (2018).

Additionally, their system has been directly inspired by the work by Williams (1992) and the work by Sutton et al. (1999), who are also the creators of the “policy gradient reinforcement learning” algorithm, which has been used in AlphaGo. This showcases how differently the same RL algorithms can be used.

RL systems and RL inspired systems keep relevance within gaming contexts, whether it is in the form of NPCs, player agents or even game balancing systems. Although Supli’s and Siqui’s (2025) closing words describe their own work, their statement speaks truth for all RL using systems:

The method’s real-time learning capabilities also present a promising direction for games where adaptive AI is essential, allowing NPCs to evolve based on player interactions. Ultimately, real-time learning, where NPCs evolve based on ongoing player feedback, could create an ever-evolving, engaging gameplay experience. (p. 11)

Conclusion

The above mentioned studies prove that Reinforcement Learning has become an advantageous strategy for video games that has found an increasing merit in the field. Continuously more companies within and outside of the gaming

industry, like Microsoft or Ubisoft, have studied RL approaches and continued research to improve and develop current machine learning algorithms. Scientists and researchers increasingly often predict real world uses for Reinforcement Learning. Many continue research in safe environments like games, to prove the effectiveness, flexibility and adaptability of RL, improve current algorithms and systems or add their own RL algorithms to the field of research, to show the significance of RL in the future.

Even long explored algorithms like SARSA or Q-Learning remain relevant in current research and contexts. DRL keeps relevance in differing systems across games and continues to change the field of RL research entirely. Areas of research around MARL are still being explored and a “best practice” has not been established.

It is therefore relevant to upkeep new approaches, and keep comparing systems across different environments.

3. Methodology and Implementation

In this section, the approach for the agent performance comparison is being presented. Systems created for this goal are being explained and chosen decisions are being elaborated. The system with all testing scripts, systems and agents can be found online on the GitHub website [here](#).

Approach

As described in the introduction, this paper aims to depict differences and similarities between RL approaches. This is done in a similar approach as Wender and Watson's research (2012) which compared different algorithms in the same environment, as well as the research by Riady et al. (2024) which compared different learning rates and discount factors. This research aims to depict and compare the same algorithm with the same goal, but with differing considerations of states and slightly altered selections of actions, combined with the aspect of a multi-agent environment. It presents that similar systems can have drastically different results.

Additionally, to further understand and compare capabilities of each agent, two differing environments have been implemented during test phases. The first is considered a static environment, while the second is a changing environment. Each agent is trained in both environments, to allow comparability between all agents, as multi-agent approaches by nature create changing environments due to the interaction between agents, while SARSA using single agent approaches do not necessarily require or create a changing environment.

The goal of all agents is keeping needs low, in this experiment specifically the needs *Hunger* and *Thirst*. To do so, objects, henceforth also called *Items*, representing *Food* and *Water* need to be moved to and then consumed in the right manner, by either drinking or eating. This straightforward scenario has been chosen to simulate NPC behaviour, similar to survival games or life simulation games while offering a simple enough procedure for clear comparison of all approaches taken to test the agents.

Three different approaches have been considered and implemented in a total of six differing agents, which have been given names for clear separation and easier description. Additionally, all agents use the SARSA algorithm, which is further explained below.

Clementine and Beans

The first approach was a standard SARSA with differing needs. The problem to be solved by the agent was the fulfillment and training of the agent to reach the lowest possible, numerical values in its needs. The needs of the agent are *Hunger* and *Thirst*, numerical values that grow over time. Thirst, in this case, grows at a 50% increase in comparison to the speed of hunger. For that, two different approaches have been tested and compared: one SARSA with a position based approach, henceforth called “Clementine”, and one SARSA with a distance based approach, henceforth described as “Beans”.

Clementine uses the offered 8x8 grid, resulting in 64 different positions it considers as its states. Beans on the other hand relies on the distance to a resource instead and considers the distance to every need as a state. For comparative purposes, the difference in distance has been segmented into 64 states. Distances are rounded to the next closest state possible.

Both algorithms rely on one NPC that executes chosen actions.

MARLClementine and MARLBeans

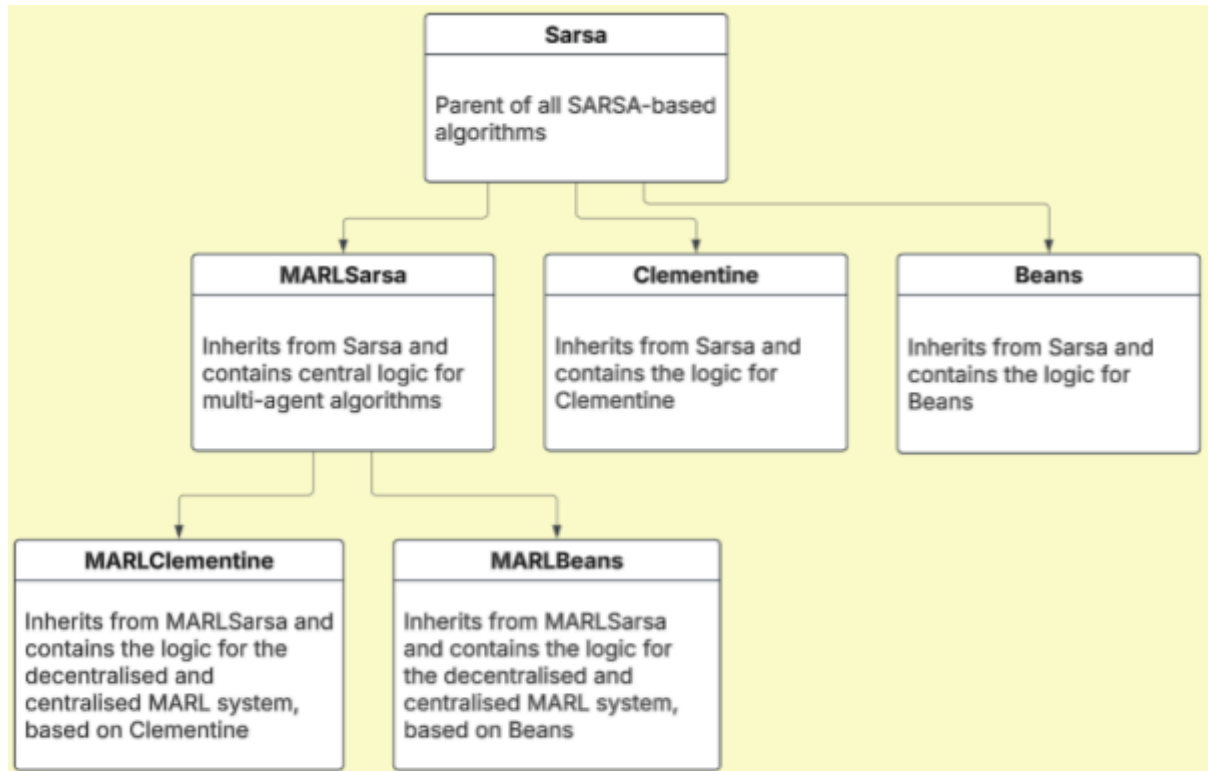
The second approach was a decentralised, multi-agent approach within the environment, using Clementine and Beans as a base for two new multi-agent approaches. They as well had to fulfil their needs, but only considered each agent's own requirements and needs. Therefore, they interacted adversely and were unable to occupy positions taken by another agent.

Unlike Clementine and Beans, both systems utilised three NPCs, each with its own SARSA algorithm, Q-Tables and needs that increased over time.

MARLClementinus and MARLBeansus

Lastly, two centralised systems were created, once again based on Clementine and Beans. Again, the agents' needs had to be fulfilled. This time, all agents of the MARL approach shared their needs and were able to work collaboratively to achieve their goal.

Throughout all three approaches, the position-based and distance-based agents have been compared to each other. Additionally, two further tests per approach have been conducted. The first consisted of a static environment with unmoving and infinite resources for the agents' needs. The second consumed a resource upon usage of the representing action and replenished the resource in a new location within the environment.



Img.2 Child-Parent architecture of the RL algorithms in the system.

SARSA Algorithm

The SARSA algorithm following this formula has been used for all agents:

$$Q(s, a) \leftarrow Q(s, a) + \alpha * (r + \gamma * Q(s', a') - Q(s, a))$$

This algorithm has been chosen as it is very adaptable to changing environments, since it learns from actions taken. This environment is both static as well as changing, due to the multi-agent aspect, as well as the testing phases with position changing tasks. In addition, many factors of the agents differ throughout testing phases. Therefore SARSA offers more flexibility than other algorithms like Q-Learning.

Additionally, unlike neural networks or large scale systems like the ones often used in combination with DRL, SARSA does not require heavy duty hardware like strong CPUs, GPUs or even a physical computing cluster like for Ye et al (2020). This makes the application of the algorithm sustainable and usable for the systems offered.

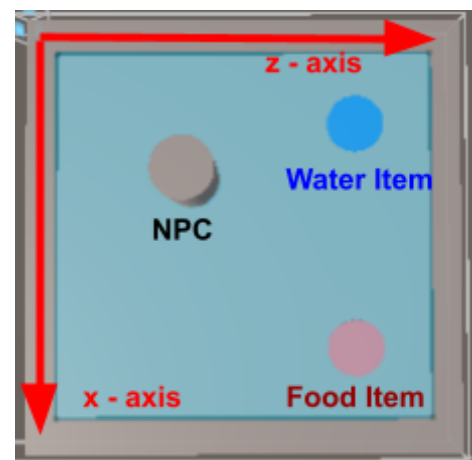
Environment and Infrastructure

For this research, the game engine Unity (Unity Technologies, 2025), version 6000.0.47f1, has been used to create and test all RL agents. The hardware components consisted of a NVIDIA GeForce RTX 3060 GPU as well as an Intel Core i7-10700F CPU.

The Environment used for testing is a 3D space, despite the 2D grid space, for the sake of future expandability and possible addition of game logic or testing functionality in 3D spaces.

The x- and z-axis have been taken for the grid space and are considered up (+x), down (-x), left (+z) and right (-z) in further contexts of the system.

The environment used by the agents for testing is a 8x8 square grid, therefore consisting of 64 differing positions possible for the agents. Each environment contains one resource for their respective need, meaning one *Food Item* and one *Water Item*.



Img.3 Depiction of the environment with an NPC (grey), a Foot item (red) and a Water item (blue)

The static environment has positioned the items as shown in the image. To create a changing environment, items are being moved around the grid whenever an agent uses one.

States and Actions

Every algorithm consists of 64 states per need. This is done since the system relies on the highest need at any given time. Whichever need is highest, will be pursued by the agent and therefore requires its own Q-Table with best approaches to reach that goal. Since every newly added need could become highest, the amount of state always equals 64 times the amount of needs, resulting in 64×2 for this experiment.

Every agent has the same amount of states. This choice was made for the sake of comparability between all algorithms, but states can be extended in the future by increasing the grid size or further separating the divider of distance considered for algorithms based on Beans.

States consider two main factors: the position of the agent and the currently highest need required. For Clementine, its position is defined by the grid, while Bean's position is defined by its distance to the items required to fulfil its needs.

The single agent algorithms each have four basic actions that define their movement, as well as two need actions, resulting in a total of six actions.

Since Clementine relies on the position on the grid for its state, it follows simple movement to the left, right, up and down. It moves by one grid square per action.

Bean's movement actions on the other hand have been changed to rely on the movement toward or away from desired need items. This choice was made since

the horizontal and vertical movement that Clementine utilises stood in no relation to Bean's states. Actions reliant on the direction corresponding to the desired item reflect the impact on the state and therefore the gained reward. This necessitates information of the location of the item, meaning Beans by default has more information about the surrounding environment than Clementine, which relies on its Q-Tables alone.

Bean's movement follows the axis' (up, down, left, right) instead of diagonals or direct lines to desired items, to ensure the same speed in movement as with Clementine. To select which axis to follow, the shortest path toward the item of choice is selected.

Clementine		Beans	
Action ID	Description	Action ID	Description
0	Move up by 1	0	Move toward food item
1	Move down by 1	1	Move toward water item
2	Move left by 1	2	Move away from food item
3	Move right by 1	3	Move away from water item
4	Consume Food	4	Consume Food
5	Consume Water	5	Consume Water

Tab.1: Overview of all actions implemented within Clementine and Beans

The multi-agent systems have inherited the same state logic and actions as their respective parent agents, meaning MARLClementine and MARLClementinus have the same actions and understanding of states as Clementine, while

MARLBeans and MARLBeansus use the same actions as Beans and consider the distance to items in their states.

Additionally, to accommodate for the movement of multiple NPCs, all multi-agent systems have gained a seventh action called *Idle*, which is further explained below.

The agents are only aware of the positions of other NPCs should they have a position next to them, meaning they block the agents' path. Although the centralised system has access to all NPCs and their position, it does not consider them in their selection of choices. This choice was selected to allow closer comparability to the decentralised and adversarial system.

MARLClementine & MARLClementinus		MARLBeans and MARLBeansus	
Action ID	Description	Action ID	Description
0	Move up by 1	0	Move toward food item
1	Move down by 1	1	Move toward water item
2	Move left by 1	2	Move away from food item
3	Move right by 1	3	Move away from water item
4	Idle	4	Idle
5	Consume Food	5	Consume Food
6	Consume Water	6	Consume Water

Tab.2: Overview of all actions implemented within MARLClementine, MARLClementinus, MARLBeans and MARLBeansus

The idle action was added to avoid overlapping of NPCs in the same position. The action itself results in no movement for the NPC. Unlike the other actions of the agents, the idle action can be a reactionary result due to a different action.

To check whether a position is blocked or not is done as an action is taken, and therefore is a conditional variable of the environment. Although an action like moving to the right can be executed, even if the path is blocked by another NPC, it will result in an idle state, without any movement. For the agent to understand the difference between a successful movement to another position versus a failed movement due to a blocked state, the idle action has been implemented.

By verifying whether an action results in an idle state before taking it, the system understands an action's consequences better and can therefore exploit its knowledge of all actions. This allows the agent to not per se understand when a location is blocked, but understand the consequences of an action taken by gaining a reward for remaining idle instead. This can result in the exploitation of differing actions should a position be blocked, as the chosen action defaults to idle and therefore allows for another action with higher reward to be taken.

An alternative approach would be to add the world's state, meaning whether a position is blocked by an object or NPC, within the agent's state system. This was intentionally avoided for multiple reasons:

Adding a boolean whether a position is blocked or not inherently doubles the amount of states for a need, resulting in longer testing phases with more specific requirements for allowing all possible state-action pairs to be tested at least once. Additionally, this adds higher, unneeded complexity to the system and bloats Q-Tables unnecessarily.

Additionally to mention is the fact that agents are capable of crashing into one another, should their movement lead them to the same spot at the same time.

This happens since they do not predict movement. Upon collision, both agents adjust to the closest state position around them.

Rewards

Rewards are being handed out depending on the highest required need. The reward system works identical for all agents, with small differences between the single-agent and multi-agent systems due to the added idle action.

The base of the reward system can be differentiated into three sections:

The first section teaches the agents their movement within the space. Agents gain the lowest possible reward for movement into unreachable areas, e.g. walls, or movements that lead them out of bounds of the testing area.

The second section teaches them the wanted behaviour in regards to their needs and their respective actions. The agents gain discouraging rewards for executing an action like *Eat* or *Drink* on the incorrect item or no item whatsoever. Additionally, the highest award within the system is given, should the correct action be used on the correct item, while the currently highest need corresponds to said action. Should a matching action be used on an item, while the need is not currently the highest, the reward is much lower in comparison.

The third section grants rewards reliant on the distance to required needs, dependent on the currently highest need. This reward is dynamic, as it is higher the closer the agent is to the most desired item. This is done to ensure movement toward those items is rewarded enough for the agent to pursue it.

These distance based rewards are higher than the ones granted by a non-highest needs action on its corresponding item, as otherwise the agent does

not move to other needs and remains idle in position on top of any item that was highest in need at the start of a run. This is partially due to the selection of the SARSA algorithm in combination with its one action learning policy. A certain “punishment” is required to teach that all needs require satisfaction, as the hope of a future, higher reward is not enough enticement for the agent to leave their current position, since it only learns from actually taken actions.

The reward for the idle action has been selected to be very negative and has the same amount as the movement into a wall. This was done to teach NPCs to avoid idle-ing and encourage moving toward needed items.

An alternative system that has been explored was a threshold based system, where needs were only pursued should a certain threshold be passed. In this system, all needs were considered equal, should all have passed the threshold. This alternative approach would have offered a solution reaching the same goal of fulfilling the agents’ needs, but was no longer pursued, as the system based on the highest needs has proven itself as more flexible and adaptable, as well as less complex and easier to expand. The new system and its advantages are further described below.

Although the system based on highest needs has established itself as more effective, the alternative system has shown interesting scenarios that have significantly defined the shape and form of the current system.

One of these scenarios can be compared to Buridan’s Principle by Lamport (2012): “A discrete decision



Img.4 Excerpt from "THE DONKEY" by Elvin Dantes, 2016

based upon an input having a continuous range of values cannot be made within a bounded length of time.” (p. 1057)

As multiple needs surpassed the threshold value chosen in this alternative system, a preference was required, as the agent otherwise halted its movement toward either item. The same was the case should both values be below the threshold. This preference was hard coded during testing, negating the goal of a fully functional, self learning system. This was one of the defining situations for the highest value system, as it avoided hard coded preferences and allowed for adaptable choices by the system itself.

A dynamic reduction or increase of rewards for required needs to ensure the preference of one item could have been used, but a lowest or highest gained reward would have been required to ensure the movement toward desired items stayed relevant. Carefully defined values for movement would have been necessary as well, to avoid the agent becoming stuck between items, falling into Buridan’s Principle once again.

Reward Values

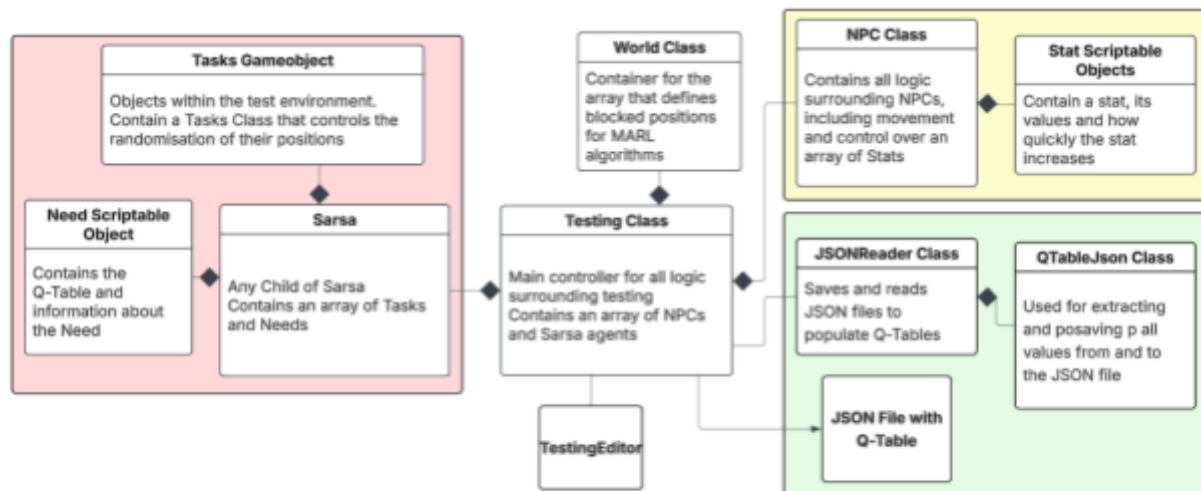
All rewards are between 0 and 1. Additionally, the selected reward values for the test runs have been intentionally extreme, to ensure clear learning of correct behaviour with little room for ambiguities in a gained reward.

Therefore, the rewards for differing actions are as follows:

Variable name	Value	Description
runIntoWallReward	0.00001	Returned for movement into borders of the grid and for the idle action
useCorrectItemReward	0.9999	Returned for using the correct action for a colliding item, should the corresponding Need be highest
useNonHighestNeedItemReward	0.002	Returned for using the correct action for a colliding item, while another Need is higher
useIncorrectItemReward	0.0001	Returned for using an action when not colliding with the corresponding item
distanceReward	0.09	Reward gained for specific distance to highest needed item. Reduced by the distance to the Need times the distanceMultiplier
distanceMultiplier	0.001	Multiplied with the distance to the Need, then used to reduce the distanceReward

Tab.3: Overview of all possible rewards that are given to the SARSA algorithm

System Architecture and Support Systems



Img.5 Depiction of the software architecture. The green area encapsulates components used for JSON extraction and saving. The red area is components connected to the Sarsa algorithms. Yellow depicts the NPC logic.

The model above depicts the structure of the created system within the Unity Engine. The system consists of two large categories: The RL algorithms which contain the full RL logic, and the support system around the agents that control parameters of the agents testing cycles and the world around the agent.

Testing Class

The main controller of the system is the Testing class. It contains most of the required information for the Reinforcement Learning algorithms and can be used to control testing variables like turn numbers, wait times between turns, or even testing modes, as later explained.

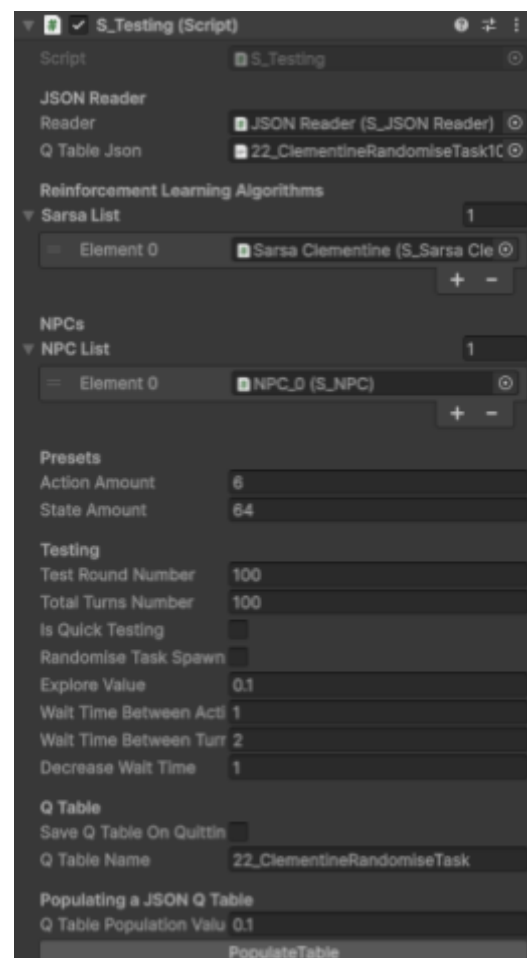
The class was designed to be as user-friendly as possible and allows for full control of the system via the Unity Engine Inspector panel. The first free sections allow control over the used Q-Table in the JSON format, the JSON Reader utilised, the Reinforcement Learning agents (variable in number dependant on choice of the user), as well as the selection and amount of NPCs

(variable in number dependant on choice of the user). Each NPC will be assigned to one RL agent. If more NPCs than RL agents should be implemented, the additional NPCs will be added in a centralised system to the last RL agent within the list. There always needs to be more or equal agents as NPCs within the system.

The *Presets* section allows variability within the action and state amounts. This was implemented for easy variation of testing environments as well as adjusted testing for the multi-agent systems, which have one more action than the single-agent systems.

The *Testing* section is the core of the script and contains variables for all testing purposes. Round numbers (meaning the number of actions used in one full run of the system without resetting), turn numbers (meaning the total number of runs of the system, with resets of the NPCs positions and world in between) and the randomisation of task spawning can be selected as wished. The standard setting is set for 100 rounds with 10 turns and no randomisation. Upon completion of a turn, the Q-Tables of all agents get saved automatically as a JSON.

Additionally, the quick testing can be activated here. Normally, NPCs move



Img.6 Depicting the Testing Script Editor Window within the Unity Inspector

similar as in video games across the environment, as if they would walk or glide from one position to another. Should quick testing be active, NPCs instead teleport to their positions chosen through actions. This choice allowed for quicker and more performance friendly testing. To accommodate this change, the variables “Wait Time Between Actions”, “Wait Time Between Turns” and “Increase Wait Time” define the times between actions, turns and the time taken by a Stat (like Hunger or Thirst) to increase the variable representing their need.

The *Q-Table* section contains the “Save Q Table On Quitting” boolean, which allows for saving of currently used tables during runtime without required completion of a turn. The “Q Table Name” can be set by the user and defines the name of the JSON file containing the information about the Q-Table.

The last section allows for the population of new JSON Q-Tables. The “Population Value” variable is the value used for all state-action pairs in the newly generated Q-Table. The “Populate Table” button allows for immediate creation of the JSON outside of runtime and is created with a Unity Editor script called “TestingEditor”.

Additionally, outside of the variability of the system, the Testing script is responsible for the testing methods for the RL agents. It controls their action amounts, has access to resetting methods and calls methods to save the Q-Tables. It controls the full cycle from taking an action to gaining a reward, and contains the loops responsible for a turn.

The JSON Reader

As mentioned in the previous section, Q-Tables are saved in JSON format. The data saved consists of an integer representing an id for the Q-Table, the name of the NPC used for testing, the value representing the exploration versus

exploitation, an action log of all selected actions within the turn the file represents, the values of all Needs upon completion of the turn, as well as the actual Q-Table consisting of float values.

To read and process saved data, the JSON reader uses a class to contain all values, before handing them to the representative agents. It consists of a method called *ExtractFiles* for extracting data from a JSON file, as well as a method called *SaveFiles* used for saving a Q-Table or creating a new one.

The naming scheme for all JSON files consists of the name chosen in the Testing class, followed by the name of the NPC used by the agent. This was done to separate between decentralised agents using NPCs.

NPCs and Stat Scriptable Objects

The NPC class controls the NPC and contains their Stats. The Stats in this context represent the numerical increase in a need like Hunger or Thirst.

A NPC consists of an NPC class and a Unity NavMesh Agent. Controlled by the NPC class, the NPC moves either across the NavMesh or teleports across the environment, should quick testing be active. It is additionally responsible for resetting the NPCs position at the end of a turn, as well as controlling the values contained in their assigned Stats. This includes the increase of values over time, as well as the decrease of values should the correct action be used while colliding with its corresponding item.

Stats consist of an enum *ID*, a starting value for their value, the current value that increases over runtime and an *increase* amount, that defines by how much the numerical representation increases in the timespan selected in the Testing class. They are represented as Scriptable Objects to allow for flexibility and an

easier assignment to NPCs for approaches like the centralised multi-agent system.

For this experiment, two Stats have been created as mentioned above: *Hunger* and *Thirst*. Both consist of a starting value of 1, but with 0.4 for Hunger and 0.6 for Thirst, the Thirst value grows at a 50% increase in comparison.

Adaptability and Expandability of the System

Each desire an NPC has or could newly acquire, consists of four different components: an enumerator defined in the *S_Desire* script as an ID, a *Need* scriptable object as the container for its representative Q-Table, a Stat scriptable object for the NPC that contains the current values describing the urge to reach the item that represents the Need in the environment.

Only those four components need to be added or modified to create a new desire. The enumerator functions as the identifier between the other three objects and has to be the same across them. The defining factors for a Stat are described above.

The Need object only requires selection of the ID enum as well as a corresponding log message, that will be saved should the representing action be executed. It additionally is the core of the system, as each Need object contains a Q-Table array that the RL algorithms refer to, should the Need's representing Stat value become the highest. The objects and their variables themselves have been kept as simple and straight forward as possible for easy implementation and expansion of the system.

The system itself has been designed to allow for flexibility. Needs, Stats and RL algorithms can be mixed and matched as desired, enabling the creation of different systems, mixtures of algorithms or shared allocation of stats on NPCs.

This approach in the system's creation permits easy implementation of new features or algorithms and allows for further testing of different NPC and algorithm combinations with little adaptation within the code.

The currently used algorithms Clementine and Beans have differing levels of adaptability themselves. Clementine is the main RL algorithm the system was designed around. To expand Clementine, its code base does not need to be touched. It is versatile and allows for endless additions of Needs, as its movement relies only on the grid environment. The base actions being detached from the system, combined with the endless possible need actions allow for endless expandability, limited by hardware.

Beans on the other hand is not as flexible within the system, as its base actions rely on the position of the item within the map and therefore increase by two whenever a new need has been added.

Part of the system described within the architecture is the so-called *World* script. This currently consists of an array relevant to the blocking logic. The array is a representation of the grid that contains information as to which positions of the grid are currently blocked by an NPC. This array is contained in the *World* class to allow for future expandability, e.g. further non-desired objects blocking the scene.

Testing / Training

As described above, two different testing versions have been implemented: quick test for pure training purposes, as well as the visual testing with more natural NPC movement. All training has been done using quick testing, for faster results. For that, the following values have been used:

- Wait time between actions: 0.2
- Wait time between turns: 0.5
- Increase wait time: 0.2

Visual testing has been used for early analysis of training results and used the values as follows:

- Wait time between actions: 1
- Wait time between turns: 2
- Increase wait time: 1

Each test consisted of 100 actions per turn, and 100 turns in total for each agent. Every agent has had two runs, one for the stationary and one for the changing environment.

Q-Tables of the used agents have been saved after every turn. All agents' Q-Tables have been saved separately, differentiating them by the NPCs used, should multiple agents have been used during a turn. This was done to allow for later comparison of the systems learning paths, but also offered insurance on accurate and error-free test runs that were retraceable should an error have occurred.

Q-Tables have been initialised with a standard value of 0.1 for every state-action pair before beginning the training cycle.

Exploration versus Exploitation during Training

The training of all agents has been split into two phases: the first phase consisted of pure training, where exploration was prioritised over exploitation. This was done to ensure a higher chance of exploring new and untrained paths for the agent. The chosen difference during training was therefore an 80% chance for exploration and a 20% chance for exploitation.

Additionally to the random exploration, a support system encouraging previously untaken actions has been implemented. This was done after it was noticed that learning was not wide spread enough across the values within the Q-Tables, meaning new policies were not explored as frequently as wanted. Teaching the strategy of the correct consumption of an item within the map on occasion was not achieved within the 100 test turns. Therefore, testing took much longer for valuable and usable results to be produced.

At first the system chose the first action that has been untested, but this once again led to longer testing times, since Need actions have been last in the lists of the action arrays. Therefore, the system considered all untaken actions, before randomly selecting one of them.

The second phase of testing consisted of comparative test runs using the previously trained Q-Tables. For this training, a much higher exploitation was considered to understand and compare training results and Q-values learnt. The chosen split was 10% exploration and 90% exploitation for that purpose.

Approach to Comparison and Expectations

This section elaborates on the expected outcomes of all algorithms and expected comparison, which will consider three different factors:

- The success of their goal
- The learning time it required to reach the goal intentionally for the first time
- The spread of learnt state-action pairs across the Q-Table

The goal is defined as the successful and intentional movement toward an item and the usage of the corresponding action (i.e. eat on Food). For this, the

Q-Tables of all agents have been compared in intervals of 10 Turns (meaning 1000 actions).

The factors have been chosen for differing reasons. The success of the goal shows whether the agent was capable of learning the intended actions, while the learning time spent offers a simple overview of required time or performance for larger or extended learning implementations. The spread across the Q-Table can indicate the agent's adaptability to learning and adjusting to changes, but can also be a sign for too wide spread and unintentional learning, depending on the situation and interpretation.

Clementine and Beans

The assumption is that Clementine will perform equally well as Beans in the stationary system with unmoving tasks. Nothing within the system indicates a slower learning pace, and outside of small differences due to randomisation of action selection, both agents are predicted to learn equally fast as well. Beans might possibly learn slower, as it has a lowered spread of possible movement across the board due to the stationary items. The direct pathing toward or away from items allows for limited movement across the grid space and does not directly allow for full exploration of all distance-based states, as some distances are impossible to reach in the non-randomised positioning of items. This will additionally lead to a lesser spread of state-action pairs for Beans. Clementine on the other hand will have a wider learning spread and have more knowledge about surrounding states quicker than Beans.

Meanwhile, Clementine will not be able to perform well in the changing environment, while Beans will thrive in it. Since Beans relies on the distance to an item, it will be able to adapt to the new situation and environment better

than Clementine. This is partially due to the knowledge imbalance between the two. Clementine is not aware of an item's position outside of the gained reward distance calculated, while Beans' state relies on a lowered distance to the items. This leads to the prediction that Beans will learn to reach its goal quicker. Despite this, the prediction is that Clementine will have a wider spread of learnt policies. Beans' spread of learnt policies will be increased in comparison to the stationary environment, as the changing environment enables more possible distances due to the changing positions of items. Therefore Beans' movement across the grid will be spread further.

MARLClementine and MARLBeans

Similar to the previous comparison, the expectation is that MARLClementine will have comparable success in reaching its goal as MARLBeans within the stationary system. The assumption is that MARLClementines agents will once again gain a larger spread of state-action pairs, as well as executed actions during its learning phase. This assumption is made due to the pathing of MARLBeans NPCs. All NPCs require to take similar, if not identical paths, resulting in more blocking of each other. Therefore, the idle action will be more prevalent in MARLBeans than in MARLClementine. Due to this mutual blocking, MARLBeans will not be able to easily reach its goal and take longer to learn successful policies than MARLClementine.

The changing system is predicted to have the opposite effect. Similar as with Clementine, MARLClementine will have no relation to the position of newly spawned items. Therefore, it will take longer to learn and have little success reaching its goal in comparison to MARLBeans. Due to the changing positions of items, the MARLBeans agents will have an easier time avoiding one another, as

they are able to use the space around them. Task randomisation will therefore be quick using MARLBeans, as items will be consumed frequently.

Both MARLClementine and MARLBeans are predicted to learn more state-action pairs than their single-agent counterparts, due to the increased amount of possible actions.

MARLClementinus and MARLBeansus

MARLClementinus and MARLBeansus are both expected to reach their goal with little test runs required within the stationary setting. This is due to the same algorithm having multiple NPCs and therefore multiple actions per turn, resulting in quicker learnt policies. The blocking of NPCs is expected for both algorithms, which could slow down learning by a little. Overall, the expected behaviour is quite similar to the behaviour of Clementine and Beans. MARLClementinus is expected to learn a little more efficiently and quicker than MARLBeansus. Additionally, the learning spread is expected to be wider than with MARLBeansus, though both algorithms are expected to have a wider spread than their single-agent counterparts due to the increased number of NPCs and therefore actions per turn.

Since both agents do not consider surrounding NPCs unless they block movement, the learning rate of differing policies is expected to be impacted by this. Despite this, learning is expected to be faster and wider spread than the single-agents.

Within the changing environment, MARLBeansus is once again expected to surpass MARLClementinus, specifically in regards to reaching the goal successfully. The spread of learnt policies will be similar across both algorithms,

while MARLBeansus will learn goal reaching policies quicker than MARLClementinus.

In summary, all agents based on Clementine are expected to slightly outperform the agents based on Beans within the stationary environment, while the assumption for the changing environment is the opposite. This is suspected due to the previously mentioned imbalance in information, specifically the fact that the Beans family is aware of distances to items, and therefore aware of their position as their actions are based on such.

4. Results and Evaluation

To evaluate results, a tool has been created in the Unity project. All visual results can be seen in [GitHub](#) using the tool in the Evaluation Scene. The JSON files containing results and Q-Tables can be found and read under “Assets/Jsons” within the GitHub project. Within, the “EVALUATION” folder contains test runs with exploration values of 0.1.

The tool depicts the highest action for each state within the Q-Table. For all algorithms based on Clementine, this will be a grid representation, while all algorithms based on Beans will depict this as an array of distances toward items. Actions shown within the tool correlate to their respective IDs described in the “States and Actions” chapter.

Clementine - Water Need							
0	0	0	0	0	0	0	0
0	2	2	2	2	2	2	1
0	0	2	2	2	2	2	1
0	2	2	2	2	1	1	1
0	0	2	1	1	1	1	1
0	0	2	1	1	3	1	1
0	0	5	1	1	1	1	1
0	3	3	3	3	3	3	3

Img.7 Evaluation table using the stationary item JSON from turn 100 of Clementine.

Above you see an example of Clementine's evaluation table, specifically the Q-Table of the Water Need. All values represent the highest action for a state,

and therefore the highest action for a position on the grid. This view allows visualisation of the learning process and the most valued actions according to their results. The drinking action, marked with a blue square in the image below, indicates that the correct action has been taught to the algorithm to sustain itself. Surrounding values show the preferred movement actions of the algorithm.

Beans - Water Need							
0	0	0	0	0	0	0	0
5	1	1	1	1	1	1	1
2	1	1	2	1	1	1	0
0	1	1	1	1	0	0	0
0	0	1	0	0	0	1	1
0	1	0	1	1	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Img.8 Evaluation table using the stationary item JSON from turn 100 of Beans.

For Beans, the states represent the distance to the highest needed item. Therefore, these actions are sorted as an array, reaching from the lowest to highest distance. Consequently, the position of the drinking action is in a high row, once again indicating the correct learning of this action.

Clementine and Beans

Comparison with Stationary Tasks

As predicted, both Clementine and Beans have been successful in their learning and been able to sustain their needs. Using the most recent Q-Table for each, both Clementine and Beans have been able to end a turn with values lower than substituted by a single action; Beans slightly outperforming Clementine. (see Tab.4) Additionally, both are visibly able to move toward the Food and Water item on the shortest path possible.

Already in round 10, Clementine was able to consume the correct type of item by using its associated action. Beans as well was able to learn both eating and drinking on the correct item within its first ten rounds. As expected, both algorithms learned quickly and at a similar pace, although, unlike in the predictions, Beans has not been slower in its learning.

Additionally, as predicted, Clementine was superior in its spread of state-action pairs. With roughly 100 more state-action pairs, it far surpassed Bean's learnt policies. (see Tab.4) This shows how the grid based states and actions learn more widely, thanks to the movement across the grid.

Algorithm	Food Need Values	Water Need Values	Learned State-Action Pairs	Unlearned State-Action Pairs
Clementine	3	6	563	205
Beans	2	4	411	357

Tab.4: Comparison of Clementine's and Beans' results for stationary tasks

Comparison with Moving Tasks

Beans and Clementine do not directly act as predicted. Clementine, as expected, is unable to find Food or Water, resulting in end values of around 40 and 60 for Food and Water. It was not able to find either during its turn. Beans on the other hand acted in an unexpected manner. It was not able to find Water or Food items actively, but through random movement managed to drink and eat on the correct item a few times during its run. This resulted in values around 40 for both Food and Water toward the end.

Interestingly, Beans was able to find Water and Food when using the Q-Table from the stationary item training, managing to act as predicted with the table. This shows that controlled learning can lead to better results. Beans has been stuck during its turn a few times, moving back and forth, but was able to gain Food and Water values around 17 and 23 around the end of the turn. With the Q-Table values from the stationary training, Beans was visibly able to find Food and Water items, although slower than in the stationary system. The same cannot be said about Clementine. Even using the better trained table from the stationary test, it was only able to find the Water item and Food item before randomisation, as has been predicted previously. Due to the grid based states, it cannot adapt quickly enough and remains in position to attempt to drink in the same spot the water was at.

Although neither algorithm was able to reach their goal with their intended training with randomised tasks, both learned the importance of drinking and eating the correct item. Clementine was able to drink and eat on the correct item around turn 20, having learned eating before drinking, while Beans was able to do both around turn 30, with drinking as the first learnt, correctly used action around turn 10.

The spread across all learnt values is slightly higher than in comparison to the stationary Q-Table. Clementine had similar amounts of state-action pairs as with the stationary tasks, while Beans slightly increased its number of learnt policies in comparison to before. (see Tab.5)

Overall, the number of state-action pairs in the results confirm the prediction that Clementine has been able to learn more different action-state pairs than Beans. (see Tab.5) It additionally shows that despite similar amounts of taught state-action pairs, the stationary Q-Table of Beans has values of higher quality, as this table is able to guide Beans to its goal despite randomised task spawning.

Algorithm	Food Need Values	Water Need Values	Learned State-Action Pairs	Unlearned State-Action Pairs
Clementine	40	60	540	228
Beans	40	40	426	342
Beans (stationary)	17	23	/	/

Tab.5: Comparison of Clementine's and Beans' results for randomised tasks

MARLClementine and MARLBeans

Comparison with Stationary Tasks

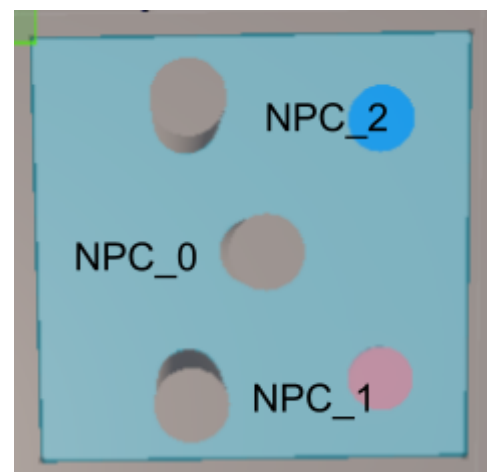
Since the decentralised system consists of three differently learning NPCs, all three Q-Table results have been evaluated for both algorithms. The Q-Table of one NPC has been given to all three for the evaluation run with 0.1 exploration as a value.

MARLClementine interacted differently than expected. The first Q-Table of NPC_0 showed movement around the center of the grid, surprisingly without any of the NPCs truly blocking each other. It took roughly 50 rounds before the NPCs huddled around water and began sustaining themselves, moving between water and food items.

Q-Table NPC_0	Food Need Values	Water Need Values
0	5	3
1	11	19
2	3	9

Tab.6: Comparison of MARLClementine's results for stationary tasks using NPC_0's Q-Table

For NPC_1's Q Table, the NPCs did not reach any of the items. Instead, they blocked each other around the center of the grid, creating a huddle around one another. Therefore, all three NPCs ended with values of 40 for Food and 60 for Water. The explanation for this phenomenon is to be found in the spread of actions. Seen in Tab.8, NPC_1 has more than 100 unknown state-action pairs in comparison to the other NPCs, resulting in less learned policies to use. The cause for this can be the random movement, but more likely is their positioning. NPC_1 is furthest away from the first required item: Water. The distance and



Img.9 Positions of the NPCs in the environment

the potential blocking of the other two NPCs encourage the lack of learnt policies and repetition of similar state-action pairs during the training phase.

NPC	Learned State-Action Pairs	Unlearned State-Action Pairs
0	591	305
1	471	425
2	604	292

Tab.7: Comparison of MARLClementine's State-Action Pairs for stationary tasks

NPC	Food Need Values	Water Need Values
0	40	60
1	40	60
2	40	60

Tab.8: Comparison of MARLClementine's results for stationary tasks using NPC_1's Q-Table

The last NPC's Q-Table proves this result with its demonstrated success. Within 10 rounds, the NPCs move toward both items. Although they block each other between both, trying to reach either, they manage to untangle themselves and gain impressively low values, creating the best results of all decentralised MARL systems using stationary tasks. (see Tab.9)

NPC	Food Need Values	Water Need Values
0	3	2
1	1	5
2	4	3

Tab.9: Comparison of MARLClementine's results for stationary tasks using NPC_2's Q-Table

The results using the MARLBeans algorithm have been unexpected as well.

Each different Q-Table has given different results. For the Q-Table of NPC_0, all NPCs began to spread at first. All gathered around the water item, then around the food item. After one NPC became stuck in its position for around 10 rounds, the NPCs managed to avoid each other nearly entirely, as they took offset, straight paths to the items they required. This resulted in the best spread of values across the three tables tested for the stationary MARLBeans test. (see Tab.10)

NPC	Food Need Values	Water Need Values
0	4	1
1	8	3
2	12	18

Tab.10: Comparison of MARLBeans' results for stationary tasks using NPC_0's Q-Table

The Q-Table of NPC_1 has caused direct movement to the right, aligning all NPCs with the Food and Water item. Within 10 rounds, the NPCs began to

cluster around the Water item, even more intensely than with NPC_0. Due to heavy blocking of each other, most could not move from their position. The Food item has rarely been reached by any of the three NPCs. (see Tab.11)

NPC	Food Need Values	Water Need Values
0	30	26
1	10	1
2	20	17

Tab.11: Comparison of MARLBeans' results for stationary tasks using NPC_1's Q-Table

Lastly, for the Q-Table of NPC_2, the turn began with nigh random movement across the grid. It took around 20 rounds before the first NPC reached the Water item. NPC_2 was stuck between the same two states for half the turn. Upon reaching round 50, all three NPCs managed to move to both needed items, resulting in good values across the board, but also the worst statement pair of this section for the unmoving NPC_2. (see Tab.12)

NPC	Food Need Values	Water Need Values
0	2	5
1	2	6
2	40	30

Tab.12: Comparison of MARLBeans' results for stationary tasks using NPC_2's Q-Table

Although the NPCs do cause blocking of paths and hinder each other, the results have been more positive than expected.

The spread of learnt state-action pairs across the Q-Table has been fairly even for all that have used MARLBeans. The results show roughly 100 more learnt actions in comparison to Beans, (see Tab.13) confirming the theory that MARLBeans learns more state-action pairs in the multi-agent setting.

NPC	Learned State-Action Pairs	Unlearned State-Action Pairs
0	549	347
1	533	363
2	521	375

Tab.13: Comparison of MARLBeans' State-Action Pairs for stationary tasks

Unlike expected, the spread of state-action pairs has been uneven in MARLClementine and consistent within MARLBeans. Yet, as predicted, the best performing agent is part of the Clementine family (MARLClementine, NPC_2).

Comparison with Moving Tasks

As expected, the MARLClementine algorithm is not able to sustain the NPCs' needs or reach their goal.

For the first Q-Table of NPC_0, all NPCs block each other's paths and move in a circle, right next to the Food item, without ever reaching it. All NPCs end the turn with the maximum values for Food and Water: 40 and 60.

NPC_1's Q-Table values do not fare better. Although not rotating around another, the NPCs do not reach the items.

The last Q-Table of NPC_2 does not break the pattern. As predicted before testing, all three MARLClementine algorithms fail to reach their goal and fail to move to any item even once.

Despite this, a noticeable result is once again the spread of state-action pairs, further confirming the statement that the distance to items is very relevant for the MARLClementine algorithm. While NPC_0 and NPC_2 had almost identical amounts of unlearned state-action pairs, NPC_2 left around 100 states more unexplored. (see Tab.14)

NPC	Learned State-Action Pairs	Unlearned State-Action Pairs
0	543	353
1	452	444
2	542	354

Tab.14: Comparison of MARLClementine's State-Action Pairs for moving tasks

The results for MARLBeans, following previous expectations, differ in comparison:

For NPC_0's Q-Table, the NPCs try to race toward water, often blocking each other's paths. Due to the slow pace toward the Water item, the Food item never gets approached, as it never becomes highest in its value. (see Tab.15)

NPC	Food Need Values	Water Need Values
0	40	53
1	40	42
2	40	54

Tab.15: Comparison of MARLBeans' results for moving tasks using NPC_0's Q-Table

A similar situation happened with the Q-Table of NPC_1, although much more indirect. Instead of blocking one another, NPCs could not find the path toward the Water item. On occasion they succeeded in reaching that goal, resulting in similar values as the previous table. (see Tab.16)

NPC	Food Need Values	Water Need Values
0	40	37
1	40	60
2	40	51

Tab.16: Comparison of MARLBeans' results for moving tasks using NPC_1's Q-Table

The best of the three Q-Tables is the last, as it guided the NPCs best toward the items. Although in a similar situation, the NPCs gained better values than from the other Q-Tables. (see Tab.17) Additionally, they were the only ones to use the correct action on a Food item.

NPC	Food Need Values	Water Need Values
0	30	27
1	30	50
2	40	40

Tab.17: Comparison of MARLBeans' results for moving tasks using NPC_1's Q-Table

Since the single-agent demonstrated superior movement in the random task environment using the stationary task Q-Table, this has been tested here as well. For this, the Q-Table MARLBeans stationary tasks Q-Table of NPC_2 has been used. The results show similar values as the previous Q-Table, indicating that both approaches would benefit from further training.

These results show that with supervised training and more turns for training, the MARLBeans agents could improve their learning and achieve better success in reaching the goal of fulfilling their needs. Meanwhile the MARLClementine does not function with randomised tasks, as predicted due to its state-definition. Additionally, as expected, the agents work adversarially and block each other, lowering every NPC's success.

The spread of state-action pairs of the MARLBeans agents is similar to each other, keeping the state-action pair amount for MARLBeans even across stationary and randomised tasks. (see Tab.18)

NPC	Learned State-Action Pairs	Unlearned State-Action Pairs
0	347	549
1	363	533
2	375	521

Tab.18: Comparison of MARLBeans' State-Action Pairs for moving tasks

MARLClementinus and MARLBeansus

Comparison with Stationary Tasks

Both centralised systems have surprised and defied predictions.

MARLClementinus, as expected, managed to reach its goal with ease. Often its NPCs tend to block one another, as all of them move toward the same goals to fulfil their common needs. Despite this, the algorithm managed to reach some of the lowest need values in the entire system. (see Tab.19)

Similarly to Clementine, it additionally was one of the algorithms with the most state-action pairs in the entire system.

MARLBeansus on the other hand surprised with its performance. Although the NPCs had begun their turn with split movement, they ended up huddled around the water item. Nearly rotating around one another, with the water item in their center, they did not learn during their training to use the drinking action, resulting in very little improvement on their stats. Since every centralised system contains three NPCs that share their stats, stats increase much faster as well, resulting in a maximum of 120 for Food and 180 for Water. (see Tab.19)

This outcome to the algorithm is especially surprising, since it has learnt an adequate amount of state-action pairs, especially in comparison to Beans.

Algorithm	Food Need Values	Water Need Values	Learned State-Action Pairs	Unlearned State-Action Pairs
MARLClementinus	2	7	623	273
MARLBeansus	120	150	530	366

Tab.19: Comparison of MARLClementinus' and MARLBeansus' results for stationary tasks

In addition, the MARLBeansus algorithm has not learnt the Eat action whatsoever, defying all predictions. As shown in the image below, the ID for eating (4) is not represented once as the highest valued action within MARLBeansus' Q-Table.

MARLBeansus - Food Need							
0	0	0	0	0	0	0	0
1	1	1	1	1	2	1	1
1	2	1	1	1	1	2	2
1	1	0	1	1	0	0	1
0	0	0	1	0	1	0	1
0	1	1	1	1	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Img.10 Evaluation table using the stationary item JSON from turn 100 of MARLBeansus.

Comparison with Moving Tasks

Once again, MARLClementinus is not capable of approaching an item or using its corresponding action, similar to Clementine in the single-agent approach. Its movement is erratic and without sense, since it has no relation between its state and the position on the grid.

MARLBeansus on the other hand is capable of following items and using their actions. NPCs don't often block one another in their movement. Despite this, they do not move often and directly enough toward their needed items, resulting in low values in comparison to other algorithms. (see Tab.20) This shows that further training is required to solidify already existing behaviour.

Both algorithms have learnt expected state-action pair amounts, although MARLClementinus has learnt more than anticipated in comparison to MARLBeansus:

Algorithm	Food Need Values	Water Need Values	Learned State-Action Pairs	Unlearned State-Action Pairs
MARLClementinus	120	180	604	292
MARLBeansus	110	155	524	372

Tab.20: Comparison of MARLClementinus' and MARLBeansus' results for moving tasks

5. Discussion

This paper has presented an approach and implementation of an agent performance comparison between six different Reinforcement Learning agents using the SARSA algorithm. This included both single- and multi-agent approaches in static and changing environments, using a flexible and expandable system architecture. An extensive overview of current research in the RL field has been showcased and the comparison of the different algorithms has been evaluated.

Further comparisons like this paper has presented will define and prove feasibility of different RL algorithms for gaming contexts in the future.

The system presented here offers different approaches for the future. Extension of the NPC logic that allows further actions to be paired and matched can be implemented. An example could be an action to “push away” other NPCs.

The system developed as part of this thesis is planned to be extended in the future. A further developed game environment is planned that allows the testing of NPCs against players.

In summary, it can be said that Reinforcement Learning has much to offer for the future, thus it is important to continue research within the field to allow such algorithms to enter the mainstream of game development and enrich gaming experiences.

6. Citation

- Alonso, E., Peter, M., Goumard, D., & Romoff, J. (2020). *Deep Reinforcement Learning for Navigation in AAA Video Games*.
<https://doi.org/10.48550/arXiv.2011.04764>
- Astley, R. (2009, October 25). *Rick Astley—Never Gonna Give You Up (Official Video) (4K Remaster) [1920x1080]*.
<https://www.youtube.com/watch?v=dQw4w9WgXcQ>
- Dantes, E. (2016). *THE DONKEY [Digital Artwork]*.
<https://web.archive.org/web/20221007091106/https://elvindantes.com/post/140747355253/the-donkey-based-on-buridans-donkeyass-and>
- Espeholt, L., Soyer, H., Munos, R., Simonyan, K., Mnih, V., Ward, T., Doron, Y., Firoiu, V., Harley, T., Dunning, I., Legg, S., & Kavukcuoglu, K. (2018). IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures. *Proceedings of the 35th International Conference on Machine Learning*, 80, 1407–1416.
- Glavin, F. G., & Madden, M. G. (2018). *Skilled Experience Catalogue: A Skill-Balancing Mechanism for Non-Player Characters using Reinforcement Learning*. 2018 IEEE Conference on Computational Intelligence and Games (CIG), Netherlands. <https://doi.org/10.1109/CIG.2018.8490405>

- Graepel, T., Herbrich, R., & Gold, J. (2004). Learning to Fight. *Proceedings of the International Conference on Computer Games: Artificial Intelligence, Design and Education*.
- Kaiser, L., Babaeizadeh, M., Milos, P., Osinski, B., Campbell, R. H., Czechowski, K., Erhan, D., Finn, C., Kozakowski, P., Levine, S., Mohiuddin, A., Sepassi, R., Tucker, G., & Michalewski, H. (2024). *Model-Based Reinforcement Learning for Atari*.
<https://doi.org/10.48550/arXiv.1903.00374>
- Klüwer, T., Xu, F., Adolphs, P., & Uszkoreit, H. (2012). *Evaluation of the KomParse Conversational Non-Player Characters in a Commercial Virtual World*. 3535–3542.
- Lamport, L. (2012). Buridan's Principle. *Foundations of Physics*, 42(8), 1056–1066. <https://doi.org/10.1007/s10701-012-9647-7>,
- Lecun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature*, 521(7553), 436–444. <https://doi.org/10.1038/nature14539>
- Li, Z., Ji, Q., Ling, X., & Liu, Q. (2025). A Comprehensive Review of Multiagent Reinforcement Learning in Video Games. *IEEE Transactions on Games*, 17(4), 873–892. <https://doi.org/10.1109/TG.2025.3588809>

- Merrick, K., & Maher, M. L. (2006). Motivated reinforcement learning for non-player characters in persistent computer game worlds. *Association for Computing Machinery, ACE '06: Proceedings of the 2006 ACM SIGCHI International Conference on Advances in computer entertainment technology*, 3-es.
- Oh, I., Rho, S., Moon, S., Son, S., Lee, H., & Chung, J. (2022, June). Creating Pro-Level AI for a Real-Time Fighting Game Using Deep Reinforcement Learning. *IEEE Transactions on Games*, 14(2), 212–220.
<https://doi.org/10.1109/TG.2021.3049539>
- Oh, J., Guo, Y., Singh, S., & Lee, H. (2018). Self-Imitation Learning. *Proceedings of the 35th International Conference on Machine Learning*, 3878–3887.
<https://proceedings.mlr.press/v80/oh18b.html>
- OpenAI, Berner, C., Brockman, G., Chan, B., Cheung, V., Dębiak, P., Dennison, C., Farhi, D., Fischer, Q., Hashme, S., Hesse, C., Józefowicz, R., Gray, S., Olsson, C., Pachocki, J., Petrov, M., Pinto, H. P. d O., Raiman, J., Salimans, T., ... Zhang, S. (2019a). *Dota 2 with Large Scale Deep Reinforcement Learning*. arXiv.Org. <https://arxiv.org/abs/1912.06680v1>
- OpenAI. (2019b). *OpenAI Five defeats Dota 2 world champions*.
<https://openai.com/index/openai-five-defeats-dota-2-world-champions/>

- Ponce, H., & Padilla, R. (2014). A Hierarchical Reinforcement Learning Based Artificial Intelligence for Non-Player Characters in Video Games. *Nature-Inspired Computation and Machine Learning*, 172–183. https://doi.org/10.1007/978-3-319-13650-9_16
- Riady, M. A., Suyanto, & Sihombing, P. (2024). Agent Performance Comparison of the Q-Learning Algorithm and SARSA Algorithm in Javanese Chess Game. *2024 Ninth International Conference on Informatics and Computing (ICIC)*, 1–6. <https://doi.org/10.1109/ICIC64337.2024.10957223>
- Schulman, J., Moritz, P., Levine, S., Jordan, M., & Abbeel, P. (2018). *High-Dimensional Continuous Control Using Generalized Advantage Estimation* (arXiv:1506.02438). arXiv. <https://doi.org/10.48550/arXiv.1506.02438>
- Servat, A., & Mohamadi, H. S. (2023). *Immersive Game Worlds: Using Deep Reinforcement Learning for Lifelike Non-Player Characters*. 1–5. <https://doi.org/10.1109/ISGS61252.2023.10559740>
- Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Driessche, G. van den, Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., & Hassabis, D. (2016).

Mastering the game of Go with deep neural networks and tree search.
529(7587), 484–489. <https://doi.org/10.1038/nature16961>

Souchleris, K., Sidiropoulos, G. K., & Papakostas, G. A. (2023, February 14). Reinforcement Learning in Game Industry—Review, Prospects and Challenges. *Applied Sciences, Deep Reinforcement Learning for Robots and Agents*. <https://doi.org/10.3390/app13042443>

Sukmana, S. E., Khairy, M. S., Ratsanjani, M. H., Rahmad, C., Maskur, & S, M. A. N. S. (2024). *Reinforcement Learning for AI NPC Literacy Educational Game*. 158–162. <https://doi.org/10.1109/IEIT64341.2024.10762983>

Supli, A. A., & Siqui, X. (2025). Leveraging deep reinforcement learning for dynamic NPC behavior and enhanced player experience in unity3d. *Entertainment Computing*, 55, 101007. <https://doi.org/10.1016/j.entcom.2025.101007>

Sutton, R. S., McAllester, D., Singh, S., & Mansour, Y. (1999). Policy Gradient Methods for Reinforcement Learning with Function Approximation. *Advances in Neural Information Processing Systems*, 12. https://proceedings.neurips.cc/paper_files/paper/1999/hash/464d828b85b0bed98e80ade0a5c43b0f-Abstract.html

- Tan, C. H., Tan, K. C., & Tay, A. (2011, December). Dynamic Game Difficulty Scaling Using Adaptive Behavior-Based AI. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(4), 289–301.
<https://doi.org/10.1109/TCIAIG.2011.2158434>
- Unity Technologies. (2025). *Unity* (Version 6000.0.47f1) [Computer software]. <https://unity.com/de>
- Vinyals, O., Babuschkin, I., Czarnecki, W. M., Mathieu, M., Dudzik, A., Chung, J., Choi, D. H., Powell, R., Ewalds, T., Georgiev, P., Oh, J., Horgan, D., Kroiss, M., Danihelka, I., Huang, A., Sifre, L., Cai, T., Agapiou, J. P., Jaderberg, M., ... Silver, D. (2019). Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature*, 575(7782), 350–354.
<https://doi.org/10.1038/s41586-019-1724-z>
- Watkins, C. (1989). *Learning from Delayed Rewards* [Ph.D. dissertation, University of Cambridge].
<https://www.cs.rhul.ac.uk/~chrisw/thesis.html>
- Wender, S., & Watson, I. (2012). Applying reinforcement learning to small scale combat in the real-time strategy game StarCraft:Broodwar. 2012 *IEEE Conference on Computational Intelligence and Games (CIG)*, 402–408.
<https://doi.org/10.1109/CIG.2012.6374183>

- Wiering, M., & Otterlo, M. (2012). *Reinforcement Learning: State-of-the-Art*. Springer. <https://doi.org/10.1007/978-3-642-27645-3>
- Williams, R. J. (1992). Simple statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning. *Machine Learning*, 8, 229–256.
- Yang, Y., Ma, C., Ding, Z., McAleer, S., Jin, C., Wang, J., & Sandholm, T. (2025). *Game-Theoretic Multiagent Reinforcement Learning*. <https://doi.org/10.48550/arXiv.2011.00583>
- Ye, D., Chen, G., Zhang, W., Chen, S., Yuan, B., Liu, B., Chen, J., Liu, Z., Qiu, F., Yu, H., Yin, Y., Shi, B., Wang, L., Shi, T., Fu, Q., Yang, W., Huang, L., & Liu, W. (2020). Towards Playing Full MOBA Games with Deep Reinforcement Learning. *Advances in Neural Information Processing Systems*, 33, 621–632. <https://proceedings.neurips.cc/paper/2020/hash/06d5ae105ea1bea4d800bc96491876e9-Abstract.html>
- Zhao, D., Wang, H., Shao, K., & Zhu, Y. (2016). Deep reinforcement learning with experience replay based on SARSA. *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*, 1–6. <https://doi.org/10.1109/SSCI.2016.7849837>

Zhao, Y., Borovikov, I., Rupert, J., Somers, C., & Beirami, A. (2019). *On Multi-Agent Learning in Team Sports Games* (arXiv:1906.10124). arXiv. <https://doi.org/10.48550/arXiv.1906.10124>

Test Environment in Unity on Github

https://github.com/LiloTM/MasterThesis_SarsaComparison.git

Affidavit

HTW Berlin. (n.d.). *Eigenständigkeitserklärung_KI-Verbot*. HTW Berlin.
Retrieved 14 March 2026, from <https://www.htw-berlin.de/lehre/lehre-gestalten/ki-in-lehre-und-pruefungen/eigenstaendigkeit-erklaeren-ki-verzeichnis/>

HTW Berlin. (n.d.). *Eigenständigkeitserklärung_KI-Verbot_engl*. HTW Berlin.
Retrieved 14 March 2026, from <https://www.htw-berlin.de/lehre/lehre-gestalten/ki-in-lehre-und-pruefungen/eigenstaendigkeit-erklaeren-ki-verzeichnis/>