VRIJE
UNIVERSITEIT
BRUSSEL

RL IRL

A feasibility study of using Reinforcement Learning for varying difficulty bots in simple Python games

Lennert Bontinck

June, 2021-2022

Student number: 0568702

Computer Science: AI

Contents

Introduction

This chapter addresses the main concepts of reinforcement learning (RL) and why RL has become more popular. In section 1.4, some important differences in the possible approaches for solving RL problems are highlighted. Afterwards, some of the classical RL problems that are solved to benchmark new state-of-the-art algorithms are introduced. Finally, the feasibility of using RL in real life (IRL) is discussed.

1.1 The reinforcement learning problem

RL is a special type of machine learning (ML) where algorithms are used to take actions in an environment. These algorithms are often referred to as agents which should learn a policy that maps the current state of the environment (S_t) to an action (A_t) that can be taken in that state of the environment. Upon taking an action, or after completing a series of actions, a reward (R_t) is given to the agent in the form of a positive or negative numerical value and the environment state may be updated. Through repeated trial-and-error interactions, experience is collected. This experience can then be used by the agent to learn a policy which chooses the optimal action for a given state. According to Sutton and Barto (2018), it is this trial-and-error search combined with a potentially delayed reward which distinguishes RL from other types of ML. A typical RL cycle is visualised in Figure 1.1.

Since the rewards can be considered a form of feedback given to the algorithm, these types of algorithms are not completely unsupervised. However, classical supervised learning tasks often require far more supervision than RL. Consequently, RL is often seen as a third ML paradigm next to supervised and unsupervised learning. Since the agent can learn in an online setting, there is also often no or minimal data needed upfront. However, training an agent in an online manner, as to have the agent collect experience ($\approx$ data) itself, can take a considerable amount of time, even when using parallelized simulation. It is also noted that offline RL using pre-collected data for training exists.

As a result of not having direct instructions on what to do, it is difficult or even impossible for a RL agent to decide if a learned policy is indeed the optimal policy. Thus, an algorithm should be exploratory enough to evaluate many or even all possible policies over time whilst also being greedy enough so that the collected reward is being maximised according to the currently used policy. Balancing this exploration/exploitation behaviour is one of many challenges in RL. A classical approach to solve this problem is by using epsilon-greedy. In epsilon-greedy, epsilon (ϵ) denotes the probability of taking a random action over the greedy action for any given step (t). Finding a balance between exploration/exploitation becomes even harder when taking into account the time it can take to perform a singular step. This means that mathematical proofs of optimal convergence for exploration strategies relying on infinite steps are even less likely to hold in real-life situations with finite time.

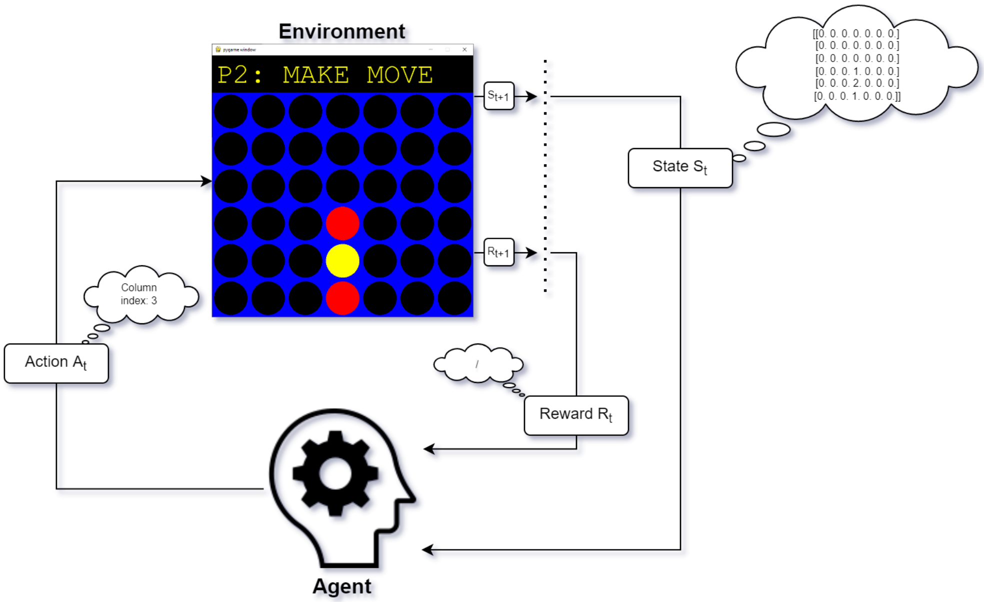

Figure 1.1: Typical RL cycle annotated with examples for a single connect four game step which doesn't yield a reward.

Whilst this definition of the reinforcement learning (RL) problem remains rather abstract, it captures most of the requirements of a good RL algorithm. Indeed, the RL problem is less concrete than some other problems in ML allowing for high creativity in approaches. Some of the many different types of approaches are discussed in section 1.4. The handbook by Sutton and Barto (2018) gives a more detailed introduction to RL if you have some form of computer science background.

1.2 Markov Decision Process

As discussed in section 1.1, the RL problem definition is rather abstract. Whilst this allows for many types of solutions to be created, having a more formal representation of the RL problem can be desired. This is especially handy for mathematical reasoning over RL and proving certain properties of a RL algorithm. To facilitate this need, Markov decision processes (MDPs) are often used as a mathematically idealized form of the RL problem with specific properties. Equation 1.1 gives the mathematical definition of a MDP.

$$\begin{aligned}
\text{MDP} &= \langle S, A, R, \gamma, p \rangle \\
S &= \text{A set of states} \\
A &= \text{A set of actions} \\
R &= \text{A set of rewards} \\
\gamma &= \text{A discount factor, } \gamma \in [0.1] \\
p &= \text{A transition function which desribes the dynamics}
\end{aligned} \tag{1.1}$$

Comparing the formal definition of an MDP to the typical RL cycle shown in Figure 1.1, it becomes apparent that many of the RL problems can indeed be formulated as a MDP. Many variants of MDPs exist, with many of them often assuming certain properties such as the Markov property. Intuitively, the Markov property means that the transition from the current state to the next state is only dependent on the current state and not on any of the previous states the agent was in. For the annotated connect four example in Figure 1.1, the Markov property holds. Indeed, all aspects of the past agent-environment interaction that make a difference in the future are captured in the current state which is simply a representation of the board. Intuitively, when neglecting potential human bias towards certain columns that could be learned, it doesn't matter how to current board emerged in deciding an optimal next move. Sutton and Barto (2018) go over the Markov property and other properties that are often assumed for MDPs in more detail.

As typical RL problems can be represented as MDPs, trying to solve MDPs corresponds to solving the formalised RL problems. Solving a MDP corresponds to acting in the MDP such that the expected discounted return is maximized. The expected discounted return is given in Equation 1.2. From this equation, it becomes visible how important it is to choose a good discount factor since a low γ will result in a *myopic agent* ($\approx$ greedy) and a high γ will result in a *farsighted agent* ($\approx$ exploratory). As a result, not only the reward influences the goal of a RL agent but also the discount factor (γ). It is noted that for some situations the discount factor can be 1, resulting in a special type of expected return: the undiscounted expected return.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{1.2}$$

1.3 Reinforcement learning is becoming more popular

The term artificial intelligence (AI) is used in conventional media a lot, even for systems that shouldn't be classified as AI. In contrast, terms like ML are far less known by the wider public, let alone the term RL. However, the computer program that beat a world champion Go player is likely one of the most recent achievements of AI the wider public will remember. Indeed, with $\approx 10^{360}$ possible game states for the Go game, far bigger than $\approx 10^{123}$ for chess, the AlphaGo system by Silver et al. (2016) that beat a world champion Go player was a monumental milestone. This AlphaGo system is a popular example of RL, although the use of human and domain knowledge combined with provided known rules makes AlphaGo a non-pure form of RL. Follow up versions such as AlphaGo Zero (Silver, Schrittwieser, et al., 2017), AlphaZero (Silver, Hubert, et al., 2017) and MuZero (Schrittwieser et al., 2019) focused on removing these extra dependencies from the algorithm, making them a purer form of RL whilst also having better and more general performance.

All of the above systems were created by DeepMind, a Google-owned company. As the name of the company suggests, all of these systems combine deep learning (DL) with RL to create a special type of RL often referred to as deep reinforcement learning (DRL). Whilst DL was revolutionising for many fields in ML, it was especially groundbreaking in RL research. Indeed, whilst RL has decades' worth of research coming from multiple almost independent threads of history (Sutton & Barto, 2018, Section 1.6), it was only with the introduction of DRL that the field has received high coverage in major publications journals such as Nature (Mnih et al., 2015; Silver et al., 2016; Silver, Schrittwieser, et al., 2017; Sutton & Barto, 2018). Section 1.4 explains in more detail why DRL is so powerful and how it opened many new doors for the RL field.

When looking at the most impressive RL-based systems, it becomes apparent that big companies and institutions such as DeepMind and OpenAI are responsible for the majority of these works. Even state-of-the-art algorithms such as Rainbow by Hessel et al. (2017, DeepMind) and famous libraries such as the Gym library by Brockman et al. (2016, OpenAI) often originate from these bigger companies. For newcomers first learning about RL this may seem intimidating and raise the idea that RL requires a lot of resources to create meaningful systems. Whilst many examples can be given of extensions to existing algorithms and many libraries exist which are provided by smaller institutions or even individuals, complete systems from such smaller institutions with obvious real-life applications are harder to find. Section 1.6 discusses the feasibility of developing such complete systems in more detail.

1.4 Common approaches for reinforcement learning

As discussed in section 1.3, DRL has opened many new doors for the RL field, but many more different approaches for RL exist. A first big division can be made on so called model-based and model-free approaches. This differentiation relates to whether or not the algorithm requires a *perfect model*. Such a perfect model should give exact probabilities and information about all possible state-action pairs for an environment. Whilst often used for proving theoretical aspects of RL through dynamic programming (DP), these types of approaches don't have many other use cases. This is in part because such a perfect model is in contrast with the ideology of RL using delayed rewards. More importantly, providing a perfect model has been proven to be computationally too hard or even impossible in most cases. Contrary to what the name suggests, model-free approaches still require a model. However, the model should only be able to generate experiences and thus fits in more easily with delayed rewards and is possible to be made in many more scenarios. Take for example the simple game of connect four, a model that generates experiences by recognising a win, a loss or a tie is simple, allowing for model-free approaches. Defining probabilities of each of these outcomes at any given state for any given action can become hard very fast, making model-based approaches impractical, if not impossible. Section 3.1 explains how connect four has over *four trillion* possible board layouts ($\approx 10^{13}$), further illustrating how hard providing a perfect model can be.

Another major differentiation that is made between RL algorithms is whether the algorithm is on-policy or off-policy. As discussed in section 1.1, the goal of an agent in RL is to learn a policy which chooses an action for any given state. This policy to be learned by the agent is called the *target policy*. This target policy is thus often the optimal policy, as we want the agent to learn to behave perfectly optimal in an environment. However, for an agent to ensure that it is using the optimal policy, sufficient exploration of all possible policies should be done as was already discussed in section 1.1. Due to this balancing, a conflict of interest arises. The requested agent's target policy is the optimal policy, but to ensure this, the agent has to remain exploratory causing it to never behave optimally the whole time. Whilst strategies such as epsilon-decay exist, where the exploration is reduced over time to limit divergence from the optimal policy, these strategies are not always ideal. What is essentially wanted is that the agent has a notion of two policies. A *target policy* which should be learned, e.g. the optimal policy, and a *behaviour policy* to collect experience, e.g. the learned target policy that takes random actions systematically. The relation between these two policies is what defines an algorithm to be on-policy or off-policy. Off-policy indicates that both policies are different whilst on-policy indicates both policies are equal. On-policy approaches can thus be seen as a special type of off-policy approaches.

A last important notion for this paper is deep reinforcement learning (DRL) approaches and how they differ from tabular approaches. The main rationale behind tabular approaches is to maintain a big table of all possible states and the actions that can be taken in that state and associate a metric value to each such pair. For example, each state-action pair might have an expected reward associated with it. Choosing the action with the highest expected reward then boils down to choosing that state-action pair with the highest value and learning from experience consists of updating that table. Whilst this can be very powerful for low complexity environments, it can quickly blow up in more complex environments. Section 3.1 discusses how making such a table for the relatively simple game of connect four can quickly become non-tractable. DRL doesn't maintain such a table but rather fits a deep learning model on the state and predicts actions from it. This directly shows how DRL enables far more use-cases for RL. Since DL is known to be great at reducing high-dimensional data such as images to low-dimensional features, it can also be faster to train for more complex environments that may still be viable to learn through tabular methods, although this is not always the case. RL algorithms could be even further categorised based on other criteria and whilst these different categorisations are important, they fall outside the scope of this paper.

1.5 Classical reinforcement learning problems

A complete reinforcement learning (RL) system consists of three main components: a conversion mechanism from an environment state to a computer representation of that state, a RL algorithm that maps that environment state to an action and a component responsible for executing the chosen action. In a real life environment system, these three different components often require a multi-disciplinary team to be implemented. Take for example a RL based robot lawn mower. To convert the environment to a computer representation, cameras and other sensors could be used to create a mapping of the environment. This requires expertise from computer vision and other fields to enforce the robot doesn't drive into pools or other forbidden areas, as we don't want to needlessly destroy these products by letting the RL algorithm learn it shouldn't drive into pools. The implementation of the RL algorithm itself requires a specialist in RL. Finally, to perform the suggested action by the algorithm, a robotics engineer might be needed to control such a robot. Whilst some of these requirements could be loosened by re-using existing systems and deep reinforcement learning (DRL) can work directly on captured images, it becomes clear that implementing a complete RL system requires more than just RL expertise.

This is one of the reasons academic research focusing on RL algorithms often make use of simulated environments to limit the focus only on the RL algorithm component. This lowers the entry boundary compared to a complete RL system and enables smaller teams or even individuals to contribute to the RL field by focusing solely on the RL component. Another bonus of simulation is that it is often far faster to complete a step in the environment than a real-life system would take, allowing for far faster learning. Some simulated environments and algorithms even allow for running training in parallel, making the often notoriously long training procedure of RL more manageable. These simulated environments are often games, since the winning criteria of a game are often easy to define but providing exact probabilities for each state is often computationally too hard or even impossible, fitting in perfectly with the delayed reward ideology of RL discussed in section 1.1.

Many libraries that provide simulated game environments exist. This allows the creator of a RL algorithm to easily test the performance on a wide variety of games and since these games are often identical between papers, it allows for easy comparison between algorithms as well. Perhaps the most common simulated environments are Atari 2600 games. The Atari 2600 is a gaming console from the late 70's known for its games with relatively simple rules and winning/losing conditions such as Pong and Breakout. Many frameworks exist that provide some of the most popular Atari 2600 games with specific support for RL. The Gym library for Python by Brockman et al. (2016, OpenAI) is one of the most popular. Most of these frameworks, including the Gym library, are based on Stella which is an Atari 2600 emulator by Anthony et al. (2021). Whilst Gym is one of the most popular environment providing libraries and many Python libraries exist that provide implementations of the most common RL algorithms with Gym support, it should be noted Python isn't the most efficient programming language which isn't ideal in combination with the already long training times that RL algorithms often require. Because of this, the Arcade Learning Environment (ALE) with both C++ and Python support for Atari 2600 games by Bellemare et al. (2013) is also a very popular choice in literature among many others.

Since this paper focuses on the feasibility of using a RL algorithm as a computer opponent in indie games, we focus on Gym-like environments in Python. This is by far the combination of programming language and environment style that has the most available RL libraries, making implementation of common RL algorithms quicker and easier, perhaps at the cost of computational efficiency. Since the Gym environment focuses on providing single agent RL environments, it does not have a standard for defining multi-agent environments in which this paper is interested. Many environment libraries that support multi-agent reinforcement learning (MARL) have been proposed, each having there own way of representing the environments. One of the most used libraries for providing MARL environments is the Petting Zoo one by Terry et al. (2020). The general interaction methods of Petting Zoo environments are very similar to those of the Gym environments, aiding in its popularity. It is also one of the few MARL environment libraries that has some form of support by common RL algorithm providing libraries such as Tianshou by Weng et al. (2021) and Ray RLlib by Liang et al. (2017). Petting Zoo provides multi-agent games from the Atari 2600 amongst other classical multi-player games such as Tic-Tac-Toe and Uno.

1.6 Feasibility of using reinforcement learning in real life

In section 1.5, it was discussed how a complete RL system consists of three main components which often require a multi-disciplinary team to create. Finding the right people for such a team is not only a hard task, but can require significant monetary resources as well. Adding to this, the required computational power and training time of complex RL algorithms are also an issue. The discussed Alpha Go system by Silver et al. (2016) was trained for three weeks using computational power comparable to over 50 high-end graphic cards. This first version of the Go champion beating algorithm also used a lot of domain knowledge, stepping away from the attractive aspect of RL not strictly requiring data or great domain expertise upfront. This required domain knowledge and computational power make the system expensive in training costs alone. In many cases, the system will not perform as wanted from the first time either, so the expensive training procedure has to be repeated for a variety of system iterations. It is also challenging to estimate how good RL will be at a given task upfront, often requiring test trials to get a grasp on potential performance. Whilst unexpected bad performance can have significant value in academic literature, the commercial value is rather limited to investing companies. These properties combined with general challenges in RL are part of the reason RL hasn't seen the widespread adoption other ML approaches have seen. It also limits the most popular ground-breaking examples of RL such as the Go beating algorithm to organisations with sufficient funds. This explains in part why many of the big examples of RL originate from the same popular companies such as OpenAI and DeepMind.

Luckily, new state-of-the-art in RL and DRL has shown to be far more sample efficient and general. Take for example the Rainbow algorithm by Hessel et al. (2017). In their paper, Hessel et al. (2017) studied six extensions to the popular deep Q-network (DQN) based algorithm by Mnih et al. (2015) and combined the strong points into a singular RL algorithm they call Rainbow. Not only does Rainbow significantly outperform these extensions on the Atari 2600 games which were discussed in section 1.5, it does so by requiring far fewer samples. With the distributional DQN extension proposed by Bellemare et al. (2017) being the best out of the six extensions when tested individually, Rainbow used 7 million frames compared to 44 million frames of training required by the distributional DQN approach. The final best-found performance of Rainbow when looking at the median human-normalized score was 223% compared to 164% of distributional DQN. A recent follow-up of AlphaGo named MuZero by Schrittwieser et al. (2019) is also significantly faster in training and didn't require domain knowledge. This also made the system more generalized, as shown by the EfficientZero system proposed by Ye et al. (2021) which is based on MuZero. EfficientZero got human performance as the median for the Atari 2600 games in just two hours of what they call real-time game experience. This shows that new state-of-the-art algorithms can reduce the above-mentioned feasibility challenges with RL to a certain extent.

Indeed, in very recent years, more and more real-life RL systems are being implemented and it is often assumed that RL is still in its early days, especially for real-life applications. Due to RL learning rules and policies itself, it isn't used in real-world environments often where it could potentially harm its surroundings. Rather, it is more often used for determining strategies and policies in computer specific tasks where bad output isn't catastrophic. Popular use cases include providing personalised suggestions such as movie suggestions. It is feasible to provide valuable features for movies and through metrics such as watch time, a reward for the provided suggestion can also be given. Thus, having features and a delayed reward, the main ingredients for RL are in place. Whilst bad suggestions won't be appreciated by the end-user, it doesn't cause any real harm either. However, RL has also been proven useful in scenarios where the output of the RL algorithm can have catastrophic consequences. Planning and strategy determination tasks are one such example. Lazic et al. (2018) describe a system where RL is used to determine a cooling strategy for data servers. Some mechanisms are in place to ensure no overheating can occur. Lazic et al. (2018) found that RL saves significant energy required for cooling whilst the temperatures of the data-servers remained acceptable.

Whilst these use cases are great news for more widespread adoption of RL, these applications are often still aimed at bigger institutions with sufficient funds. True applications for small businesses or even individuals are hard to come by in the current state of RL. However, since many of the commonly used environments for RL, such as the Atari 2600 games, are gaming environments, one possible application addressing indie developers, is the use of RL for providing more human-like computer opponents in video games. The next chapters will address multi-agent reinforcement learning (MARL), a type of RL where two or more agents are present in an environment, as is the case with multiplayer games. The other chapters of this paper study the feasibility of RL to be used by indie game developers for creating a human-like connect four computer opponent by implementing all required components and evaluating them.

Multi-agent reinforcement learning

The previous chapter gave a brief introduction to reinforcement learning (RL). Section 1.5 discussed how common tasks in RL literature includes playing games. However, many of the common RL tasks and the often used Atari 2600 games are single-agent environments or single-agent variants of multi-agent environments. For the feasibility study of this paper, multi-agent reinforcement learning (MARL) is needed. This chapter gives a brief overview of what MARL is and how it differs from the previously discussed RL problem. Some of the challenges in MARL are also addressed. Finally, a brief discussion on existing literature for using RL as means of creating computer opponents in games is discussed in this chapter.

2.1 The difference between single-agent and multi-agent environments

Multi-agent reinforcement learning (MARL) is a generalisation of single-agent RL which was formally defined with MDPs in section 1.2. Contrary to what the name suggests, MARL is not simply any RL setting where more than one agent is present. For example, the Gym library by Brockman et al. (2016) provides the Atari 2600 game Pong and considers it to be a single-agent environment. However, Pong is a tennis themed game where a ball is played back and forward between two players in a 2D setting. But because the opponent player in this Gym implementation of the Pong environment is a fixed strategy, rule-based, entity which does not learn or adapt itself in the environment, it is not considered to be a MARL setting. As such, the opponent agent can be seen as a part of the environment and the learning agent does not have to take into account the presence of this agent in the environment. Feriani and Hossain (2020) argue a RL setting is only considered MARL when a set of agents learn in a shared environment through interaction with the environment and the other agents, either directly or indirectly. Since the computer-opponent in that Pong environment does not learn, the environment is not seen as a MARL environment following this definition. The pong environment provided by the Petting Zoo environment (Terry et al., 2020) allows for using another RL agent as an opponent. In that environment, two agents would be learning and playing against each other, making that Pong environment a MARL environment according to the previously given definition. Intuitively, MARL should not only include multiple acting agents in the environment, but these agents should also be learning in the environment and adapting to it. However, there does not seem to be a shared formal definition of what can and can't be considered MARL. For example, Buşoniu et al. (2010) put less stress on the learning aspect of the agents interacting in a MARL environment.

In section 1.2, it was discussed how RL problems can be formalised as a Markov decision process (MDP). This mathematically idealized form of the RL problem does not allow a direct way of formalising MARL settings. For this reason, a generalisation of MDPs called stochastic games are used to formalise the multi-agent settings. Equation 2.1 gives the mathematical definition of a stochastic game. A further explanation of the mathematical differences between MDPs and stochastic games falls outside the scope of this paper and is not required for the development and understanding of this paper's proposed system. Szajowski (2019) and Buşoniu et al. (2010) provide a great introduction to these more mathematical aspects of MARL and stochastic games.

$$\begin{aligned}
\text{Stochastic game} &= \langle X, \boldsymbol{U}, f, \boldsymbol{\rho} \rangle \\
X &= \text{A set of states} \\
\boldsymbol{U} &= U_1 \times U_2 \times ... \times U_n \\
&= \text{Joined action set of } n \text{ agents} \\
f &= X \times \boldsymbol{U} \times X \in [0.1] \\
&= \text{State transition probability function} \\
\boldsymbol{\rho} &= \{\rho_1, ...\rho_n\} \text{ with } \rho_i = X \times \boldsymbol{U} \times X \in \mathbb{R} \\
&= \text{Reward functions of } n \text{ agents}
\end{aligned} \tag{2.1}$$

2.2 Approaches to multi-agent reinforcement learning

The strategies used for the different agents present in MARL can differ greatly, and the behaviour of the other agents in the environment can influence the learned policy by an individual agent. If the agents share a common goal, where playing as a team is important, the learned policy for playing the game will differ greatly from a setting where agents play against each other. The predictability of the other agents in the environment will also influence the learned policy of an agent. Many other influencing factors can be listed and in general, the type of agents present in a multi-agent environment will influence the learned policy together with the reward and gamma that also influenced the agent's policy in single-agent environments.

One of the most common approaches in MARL is using agents that all share the same RL algorithm for training. This strategy is referred to as self-play. Intuitively, by doing self-play all agents should incrementally improve their behaviour, forming a smarter opponent or teammate, which aids in further improving its learned policy until it converges. This approach requires the least edits from a single-agent environment and requires no additional knowledge or data, which is great for general applicability. However, self-play can have many unwanted side effects and there are many potential issues with this approach. Section 2.3 highlights some of these issues. Many techniques to incorporate with self-play have been proposed to combat the potential issues of the approach (DiGiovanni & Zell, 2021; Liu et al., 2021; Silver et al., 2018).

Another strategy that is often used in MARL, is to train agents against a random policy. This strategy is often combined with highly exploratory parameters for the agent so that the learning agent learns a policy from a broad range of samples. Whilst the resulting policy is often not that good, especially in settings where agents are meant to play against each other, it is often better than the initial policy of an agent which is almost always a random policy itself. This learned policy from playing against a random agent is often used in further training in more complex strategies such as self-play.

As an alternative for playing against a random policy, the agent can play against a fixed strategy agent to obtain a good base policy for further training through self-play or other techniques.

The fixed policy should not be too predictable so that the agent learns the general game rules rather than a good response to the behaviour of the other agents. This often means that this fixed policy requires a certain amount of domain knowledge which takes away from the attractive property of RL that it does not strictly require domain knowledge or starting data. However, as MARL is often a significantly harder problem than single-agent alternatives, some form of domain knowledge is often required, as was also the case for AlphaGo by Silver et al. (2016) discussed in section 1.3. One bonus of using a fixed policy based on domain knowledge is the possibility of creating varying difficulty levels of the policy, especially in duelling environments. This can allow for an approach called league-based training, where an agent is trained against increasing difficulty opponents. It should be noted that a random or fixed policy agent does not learn from the environment, this means that this strategy can be seen as single-agent RL according to the definition of MARL given in section 2.1.

A final approach that is worth mentioning for this paper, is an extension on self-play where the RL agents use differing RL algorithms. This allows for comparing the performance of different RL algorithms in a certain environment, where agents in a duelling environment using a specific algorithm might have a higher reward rate and thus can be seen as the better performing algorithm. This approach can also aid in reducing the risk that a learned policy is overfitting on the strategies of the other agents in the environment rather than learning a good general policy for the environment. Buşoniu et al. (2010) gives a more in-depth overview of different approaches to MARL.

2.3 Challenges with multi-agent reinforcement learning

Single-agent RL can already be highly challenging when used in complex environments. Adding multiple agents to the learning process with MARL makes the learning process harder and causes a set of new challenges to arise. One of the most challenging problems is the problem of non-stationarity. The problem an individual agent should learn to solve changes as the policy of the simultaneously learned agents also changes. If such changes are too rapid, the target for an individual agent might be moving too fast so that the agent can never learn a good policy. Papoudakis et al. (2019) discuss this issue in greater detail and give some potential methods to reduce this issue.

Another very challenging issue is the fact that MARL is less stable in general and requires more samples for learning. This makes MARL computationally even harder then the already computationally hard single-agent RL problems. However, as was the case for single-agent RL, literature in MARL has proposed methods that greatly improve the sample efficiency of MARL and aim to stabilise the training process. Liu et al. (2021) discuss some of the techniques that can be used for giving better stability and sample-efficiency in self-play.

Feriani and Hossain (2020) and Buşoniu et al. (2010) discuss other common challenges with MARL such as partial observability and scalability issues, but these fall outside the scope of this paper.

2.4 Using reinforcement-learning as a computer opponent in games

As a potential use case for RL in real life (IRL), this paper aims to study the feasibility of using MARL techniques to create an agent that can be used as a computer opponent in simple Python games. In particular, this paper aims to modify an existing connect four game implemented with Pygame so that it is compatible with common RL libraries. Afterwards, those common RL libraries are used to train common RL algorithms on the game. Multiple variants of the learned policy should ultimately be stored to have the option of an increasing difficulty computer opponent that is more human-like than rule-based policies. If feasible, this could enable small developers to provide computer opponents in their game without having to code a rule-based agent for it, which can often be hard or even impossible. The computational power of doing a prediction from an already trained RL agent is also very manageable on most hardware and often takes no more than a few milliseconds. This is far more desirable than approaches that rely on exploring game trees which can take a lot of computational power and time.

Since many of the commonly used tasks for RL discussed in section 1.5 revolve around playing video games such as Atari 2600 games, much literature exists on creating excellent performing bots for certain video games (Hessel et al., 2017; Mnih et al., 2015; Shao et al., 2019; Silver et al., 2018; Ye et al., 2021). However, the evaluation of these algorithms is often based on win rates against other RL algorithms or compared to the median human-normalized score. Whilst these are great objective measures, they don't tell much about the actual learned policy and how *human-like* it is. An algorithm with excellent performance according to these metrics might follow a policy that is completely different from classical human behaviour. The world champion Go beating algorithm, AplhaGo by Silver et al. (2016) made some surprising moves that many professional Go players labelled as very strange. The 37th move in the second game was so odd that some of the expert commentators of the live match thought it was a mistake by the RL system. Whilst creative moves like these can give great new insight into potential policies to be used in these games which humans may not have thought of, such surprising behaviour is not the desired behaviour of most computer controlled video game opponents. For this reason, this paper focuses on a more subjective evaluation of the emerged policies.

Since academic literature often focuses on beating state-of-the-art or human-level performance in games, literature on using RL as opponents in video games is rather limited. Mendonca et al. (2015) focused on creating a human-like RL agent for an arcade style fighting game. Their findings showed promising potential but were not satisfactory for general use. Mendonca et al. (2015) showed how training an RL agent requires a more thought through reward strategy and that objective measures don't reflect the subjectively found performance of the agents. Non scientific resources explore this idea a lot, with some examples for connect four specific RL agents existing online (Alderton et al., 2019; Dawson, 2020). However, these approaches are rather vague on the learned policy of the agent and actual usability of the learned policy as a fun connect four computer opponent.

Learning to play connect four with RL

The previous chapters introduced the main concepts of reinforcement learning (RL) and multi-agent reinforcement learning (MARL). It was discussed that real-life use cases for RL are mostly limited to institutions with a high amount of resources at the moment. As a possible use case for indie developers, which are small game companies or even individuals, the feasibility of using RL as a computer opponent in the simple Python games was studied. In particular, an open-source Pygame implementation of connect four was converted to a Gym-like environment and common RL algorithms were trained in an online manner on it. This chapter highlights the design decisions of that systems. All of the documented source code together with annotated Jupyter Notebooks are available on the GitHub repository of this project (Bontinck, 2022).

3.1 Connect four and its complexity

Connect four is a trademarked game first sold in February 1974, although many similar connection board games have been made throughout history. In connect four, two players compete against each other by dropping player-specific coins into an open slot of a shared board. This board is seven columns wide and six rows high. A player drops a coin from the top of a column and it drops to the lowest available row of that column which still has a free spot. If there is no free spot in a specific column, the coin can not be inserted in that column. The game is won by the player that can connect four of their pieces in a straight horizontal, vertical or diagonal line. When to board is full and none of the players has won, the game is a tie. A sample game between a rainbow policy and human player is shown in Figure 3.1, a video of which is also available on YouTube [1]. It is important to note that connect four is a solved game. This means that given the first player follows the optimal policy, this player is always guaranteed to win.

The rules of connect four are very simple and with a board of only six by seven, the game seems simple enough. However, connect four has 4531985219092 possible boards, which is in the order of $\approx 10^{13}$. Whilst this is still far less than the game complexity of chess ($\approx 10^{123}$) or Go ($\approx 10^{360}$), it is still significantly larger than the simple rules would suggest. Since there are so many states, each with up to seven possible actions (free columns), tabular approaches as discussed in section 1.4 are not feasible. Not only would such a table require significant memory, but it would also take far too long to reach all states at least once. This further shows how limited tabular methods are.

Whilst a manual feature representation could potentially be determined that lowers the state dimension to a feasible amount, it would require a domain specific approach and would give

[1] https://youtube.com/shorts/DZesQ4GI0hE

no value to a general feasibility study. Because of this, a deep reinforcement learning (DRL) approach is taken to learn the connect four game. This will make it possible to learn directly from the matrix representation of the board shown in the annotations of Figure 1.1. Two deep Q-network (DQN) approaches and one Rainbow approach are implemented and trained in a variety of ways, which is discussed in what follows.

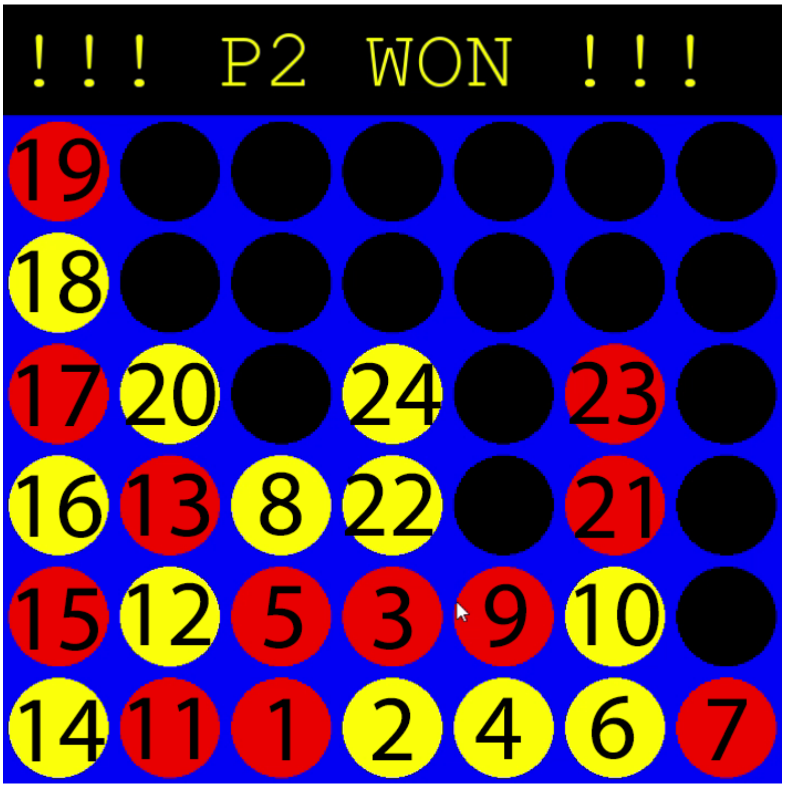

Figure 3.1: Sample connect four game annotated with the time steps at which the coins are placed. Player one is red and is the best-found rainbow policy from the paper notebook 9, player two is yellow and is a human. Player two won with the diagonal coins of 14, 12, 8 and 24. A video of this game can be found on YouTube (https://youtube.com/shorts/DZesQ4GI0hE).

3.2 Gym implementation of connect four game environment

Since this paper aims to study the feasibility of implementing RL opponents in simple Python based games, it was chosen to build further upon an existing connect four implementation. The chosen base implementation is one by Nayak (2019) and makes use of the popular Pygame library by Lindstrom et al. (2022). The author's permission of reusing his code was granted after brief communication on LinkedIn. Some edits were made to this base implementation to make the code more readable and the game UI more informative. Since this base game is just a human-play variant of the connect four game, it lacks many of the components required to use it in a

RL setting such as a reward strategy and more.

As a first step of converting this base Pygame to an environment usable for MARL, it was converted to a custom Gym environment. A custom Gym environment should inherit from the Gym class and provide a few specific attributes and functions, a skeleton for this is given in Algorithm 16. A Gym environment is a class from the popular Gym library by Brockman et al. (2016) introduced in section 1.5. Converting the base Pygame to a Gym environment was straightforward given the many tutorials and documentation available discussing how to create a custom environment. The conversion process took about four hours, including testing. This version of the environment is called `ConnectFourPygameEnvV1` and is available under `gym_connect4_pygame` folder in the Github repository of this project (Bontinck, 2022).

The init method supports two render modes, `terminal` and `human`. It also allows for specifying the column and row count of the connect four board, although the default grid of 6x7 is used throughout this paper. The observation space is specified as a Gym directory consisting of one key called `board`. The board in the observation space is a two-dimensional Gym box (≈ 2D array) with integer values between zero and two, including boundaries. Zero corresponds to an empty space, 1 to the first player's coin and 2 to the second player's coin. This corresponds to the representation annotated in Figure 1.1. The action space is a discrete space specifying an integer that corresponds to the column index a coin should be dropped into.

The step method contains the main logic of a Gym environment as it performs a specified action in the environment and updates the state of the environment. It also returns a reward, a boolean to specify if the game has ended and an info object. In this implementation, there is no mechanism in place to reject a player to insert a coin in a full column. However, attempting to do so keeps the board unaltered and gives the turn to the same player again. After this, the step function checks for a winning board or tie board to end the game. If the game is not over, the step function alternates the current player. The step function makes use of many abstracted functions from the base game, which are not shown in the minimal Gym skeleton of Algorithm 16.

The render method supports a terminal printed render of the board which prints the 2D matrix representation of the board after each step. It also has support for a human mode which renders the game in a Pygame instance, just like playing the game regularly would do. The reset and close functions are trivial and are not discussed in detail. The experimental notebook 2 gives details on how to work with this V1 of the custom connect four Gym environment (Bontinck, 2022). Since the Gym environment doesn't provide any standards on specifying a multi-agent environment, the implementation is custom and does not comply with any specific standard. In essence, the info object, which is one of the returned values from the step and reset function, gives all the required info for playing a loop in the game. Since this custom Gym environment is multi-agent but doesn't follow any conventions, it is not supported directly by any RL algorithm library and is therefore not used.

3.3 Petting Zoo implementation of connect four

Having implemented the base game as a custom Gym environment in the previous section, it was noted that the Gym library has no direct support or conventions for multi-agent environments. Because of this, the environment is unlike the other Gym environments, which are all single-agent environments. This means that it doesn't neatly integrate with existing RL algo-

Algorithm 1: Minimal structure of a Gym environment

```
// Init method should accept a render mode.
function __init__(self, render_mode, ...)
    if loss > $\theta_{\text{exp}}$ then
        ...
        // Observation space for the environment: structure of an
           observation
        self.observation_space = gym.spaces.Dict(...)
        ...
        // Action space for the environment: structure for an action
        self.action_space = gym.spaces.Discrete(...)
        ...
// Reset function to return the environment to its start state.
function reset(self, return_info=False)
    ... return (observation, info) if return_info else observation
// Step function to perform an action in the environment.
function step(self, action: int)
    ...
    // Return an observation of the new state, the received reward, a
       boolean specifying if the environment is finished and an info object
       containing additional information.
    return observation, reward, done, info
// Render function to visualise the environment.
function render(self, mode='terminal')
    ...
// Close function to clean up an environment.
function close(self)
    ...
```

rithm libraries such as Tianshou (Weng et al., 2021) or Ray RLlib (Liang et al., 2017). As this paper is a feasibility study, which includes ease of implementation as a criterion, it was opted to adopt this V1 of the environment to a Petting Zoo environment which is supported by these libraries. As discussed in section 1.5, Petting Zoo is a commonly used library for MARL as it provides multi-agent environments in a Gym like fashion. Converting the custom Gym environment to a Petting Zoo based environment doesn't involve a lot of code changes but did require significant troubleshooting. This is mostly due to the Petting Zoo documentation not providing instructions on how to implement a custom environment, which the Gym library documentation did do. However, through studying the source code of simple environments provided by Petting Zoo, such as the Tic-Tac-Toe environment, adopting the custom Gym environment to a Petting Zoo compatible environment took about 15 hours of work. Most of this time was spent troubleshooting compatibility issues with the Tianshou RL algorithm library.

The Petting Zoo variant of the connect four game is provided as `ConnectFourPygameEnvV2` under the `gym_connect4_pygame` folder in the Github repository of this project (Bontinck, 2022). Most notable changes include inheriting from the Petting Zoo `AECEnv` class rather than Gym's `Env` class and wrapping this custom Petting Zoo environment in some common Petting Zoo environment wrappers. The observation space is also extended to include an action mask besides the board representation taken straight from the V1 variant of the environment. This action mask specifies which of the actions from the action space are valid for the current state. If specified during initialisation, the environment can thus strictly enforce an agent to only use an allowed action for a given state. Whilst this incorporates some domain knowledge, it can drastically improve the learning speed of the agents. However, it is still possible to let the agents learn this behaviour by specifying to not use the mask and giving a negative reward for placing a coin in a full column. Besides this, the observation space and action space should be lists containing a variant of the space for both agents, although these spaces are equal for both agents in the connect four game environment. Next to this, the `step` and `reset` methods no longer return game information such as the state, a done boolean, a reward and an info object. Rather, Petting Zoo and the libraries that are compatible with it rely on attributes such as `rewards`, `dones` and `infos` to keep track of the game state and an `observe` function to observe the current state of the environment. Exact implementation details are not of importance and thus not given in this paper, although the source code is documented and available on GitHub (Bontinck, 2022). The final petting zoo environment is over 650 lines of code, compared to the 350 lines of code for the original base game implementation and 450 lines of code for the custom Gym environment (environment V1).

3.4 Reward strategy

As was discussed in section 2.2, the target policy of a MARL problem depends on the reward function, the used discount factor (γ) and the other agents in the environment. Whilst the discount factor and agents in the environment can be easily interchanged, the reward strategy has to be implemented in the custom environment. For the custom Petting Zoo environment, six types of rewards were provided. The value of each reward can be set during the initialisation of the environment. The most trivial rewards are those for winning, losing and having a tie game. To allow for learning not to place a coin in a full column, a (negative) reward for making an invalid move can also be specified. However, it is noted that the environment can be initialised to use action masks so that the agent can't choose an invalid move, meaning an invalid move would never occur and thus this reward would never be used.

As a means to incentivise longer games, it is also possible to specify a reward for each move made. Likewise, to incentivise a defensive behaviour by the agent, a reward for what is called a blocking move can also be given. A move is considered a blocking move if the chosen action by an agent would result in a win for the other agent if the other agent had done that action. The latter incorporates domain knowledge which guides the agent to learn a good policy. Whilst this diverges from the purest form of RL, it can drastically improve the training time and usability of the policy in terms of human play. The rewards for making a move and making a blocking move were added incrementally to the environment after it was found that the learned policy using only the other four rewards was very naive. Chapter 4 discusses the evaluation of the emerged policies and clarifies how different values for these rewards influence the found policies by the agents.

3.5 Using MLP based deep Q-networks to learn connect four

One of the first very successful RL algorithms in learning to play Atari 2600 games at a human level, was a generalisation of the Q-learning algorithm using a deep Q-network (DQN) by Mnih et al. (2015). The approach by Mnih et al. (2015) can be seen as a deep reinforcement learning (DRL) variant of the popular tabular Q-learning algorithm by Watkins (1989). As discussed in section 1.4, the difference between tabular methods and DRL methods is that the first maintains a table of all possible state-action pairs and an expected reward whilst the latter uses deep learning (DL) models to map the environment state to an action. Tabular methods can quickly blow up in dimensionality whilst DRL can reduce this dimensionality making it possible to solve problems that were not feasible to solve with tabular methods. As discussed in section 3.1, connect four has to many possible states for tabular methods such as Q-learning to be feasible, hence why the DRL variant DQN is used as a first attempt at learning connect four. More details on how tabular Q-learning works and a convergence proof of the algorithm is given by Watkins and Dayan (1992).

As the name suggests, DQN also uses the concept of Q-values just like the Q-learning algorithm but aims to generalise it. Q-values can be seen as an estimated reward for taking an action (a) in a given state (s). These Q-values are the metrics stored in tabular methods for determining and updating the optimal policy. However, DQN doesn't keep track of individual Q-values in a Q-table but rather aims to approximate a Q-function so that the Q-values can be approximated by the Q-function $Q(s, a)$. In essence, it replaces the Q-table with approximated Q-values in Q-learning by an approximated Q-function. However, learning that Q-function isn't an easy task, hence why DL and more specifically artificial neural networks (ANNs) come into play. When using a DQN, a deep neural network (DNN) is used to to approximate the optimal Q-function.

Figure 3.2 shows one of the DQN models that was used for the deep Q-learning of connect four. This first deep Q-learner is based on a multilayer perceptron (MLP). It consists of three hidden fully connected layers with 128 nodes, all using a ReLu activation function. The input is the board represented as a 6x7 2D matrix, which corresponds to the state of the environment. The output is the expected reward for each action if it were taken in that state. To update this network, a typical loss optimisation is used, in particular, the Adam optimizer proposed by Kingma and Ba (2014) is used. The loss function in question is based on the difference in the output of the network with the right-hand side of the Bellman equation given in Equation 3.1.

Since this target policy makes use of the maximum action a' in the new state s' upon taking an action a in the current state s, it differs from the behaviour policy which might use something like epsilon-greedy as well. Since these two policies are not equal, deep Q-learning is an off-policy approach, just like regular Q-learning, as was discussed in section 1.4.

$$q_*(s,a) = E[R_{t+1} + \gamma \max_{a'} q_*(s',a')] \tag{3.1}$$

More technical details on the working of deep Q-learning by using a DQN can be found in the paper by Mnih et al. (2015) which first proposed this algorithm. One of the attractive properties of using the custom Petting Zoo environment is that it directly integrates with the Tianshou library by Weng et al. (2021). This Tianshou library provides an off-policy learner and a DQN policy, which allows for using DQN just by specifying which network architecture to use. This makes the implementation of this rather complex algorithm for our custom connect four Petting Zoo environment extremely simple. All that has to be done is providing the Pytorch model of the MLP based DQN shown in Figure 3.2. More implementation details and the chosen parameters for some of Tianshou functions can be found in the paper notebooks 3, 4 and 5 available on the GitHub repository of this project (Bontinck, 2022).

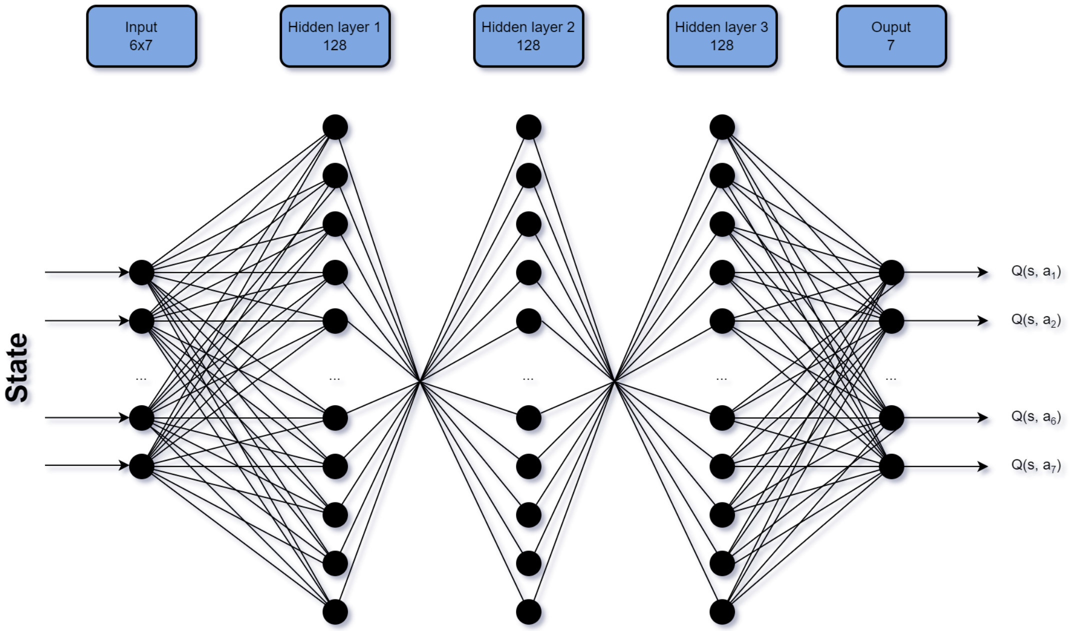

Figure 3.2: MLP based DQN architecture used for Deep Q-Learning.

3.6 Using CNN based deep Q-networks to learn connect four

The MLP based DQN proposed in section 3.5 is sophisticated enough that it likely won't be a factor in making learning of the connect four game impossible. However, there is a more logical architecture to be used. This more logical architecture revolves around using a convolution layer, forming a type of convolution neural network (CNN). Convolution layers are often used for image processing and an in depth explanation of this layer is given by Alzubaidi et al. (2021).

Intuitively, the 2-dimensional convolutional layer used for the DQN goes over the input 2D board representation with a 4x4 kernel. This kernel consists of a 4x4 grid of weights that have to be learned. These weights are used to perform a convolution of the 4x4 grid to a singular value. This kernel moves over the 2D board matrix in a left to right, top to bottom manner, moving by one element each time. This reduces the initial board input of a 6x7 2d matrix to a 2d matrix of 3x4. The kernel uses the same weights for the whole board in a single filter. This process is visualised in Figure 3.3.

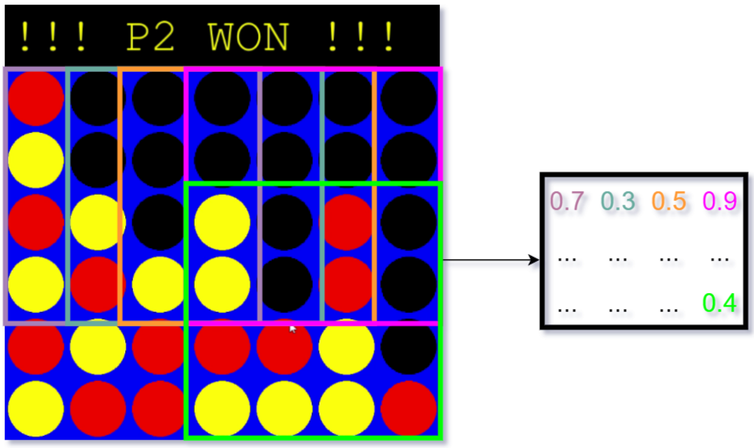

Figure 3.3: Single convolutional filter for a 4x4 kernel with stride one.

Whilst only one such filter would greatly reduce the input dimension, a lot of information would be lost. At least both diagonal lines, four horizontal lines and four vertical lines should be recognisable since these form a win or a loss condition. Whilst it would be possible to learn these conditions with fewer filters, it is reasoned at least 10 or even 20 filters should be available to recognize all win and loss conditions. Since the computational overhead of providing more filters isn't drastic and it can give better performance by allowing to learn more than just the win conditions, it was opted to create 64 filters. This results in a final output dimension of 3x4x64 for the convolutional layer. After this convolutional layer, the output is flattened and two fully connected layers of size 128 are provided before going to the output layer of the network. In essence, the hidden layer 1 shown in Figure 3.2 is replaced by 64 filters of the CNN layer visualised in figure 3.3. The implementation of this CNN based DQN is performed in paper notebook 6 which is available in the GitHub repository of this project (Bontinck, 2022).

3.7 Using CNN based Rainbow to learn connect four

Whilst DQN has proven to be very powerful at learning single-agent Atari 2600 games, there are some issues with DQN. Many extensions have been proposed to solve one or more of these issues, but they fail to solve all of them or even introduce new ones, as discussed by Hessel et al. (2017). To solve this, Hessel et al. (2017) proposed a RL algorithm that combines all of the strong points of six different DQN extensions into a single RL algorithm. This resulted in the best final performance based on the median human-normalized score for 57 Atari 2600 games and required fewer sample frames to do so than any of the individual extensions. The exact details on the Rainbow algorithm can be found in the Hessel et al. (2017) paper. One of the nice things about using the Tianshou library by Weng et al. (2021) is the fact that many of these state-of-the-art and complex algorithms are implemented already and can be set up with relatively few lines of code and only a high-level understanding of the algorithm.

The Rainbow algorithm is implemented to use the same convolutional layer of the CNN based DQN from section 3.6. This layer can be seen as a feature extraction layer. Subsequent layers are adapted from the original Rainbow paper by Hessel et al. (2017), where noisy linear layers are used together with an epsilon-decay. Normally, the algorithm only uses one of these two to make the agents exploratory, but to combat the low amount of training samples due to limited computational power and to incentivise keeping an exploratory behaviour, they are used together for this project. The rainbow algorithm also makes use of the duelling property described by Wang et al. (2015) as it should aid in learning a good policy for this MARL connect four environment. The implementation of the Rainbow algorithm is further described in the paper notebook 9 on the GitHub repository of this project (Bontinck, 2022).

3.8 Using mini-max as a rule-based opponent

Finally, a mini-max based algorithm is implemented to function as a connect four player. Minimax is a game tree searching algorithm that can make use of alpha-beta pruning to limit the number of trees to visit. Intuitively the mini-max algorithm alternates in maximising (current player's optimal strategy) and minimising (opponent's player optimal strategy) the chosen action based on a given score that is returned at the end of the recursive process. Since it is computationally too hard to do this for the full depth of the game tree, it is only done for a specified depth. The recursive loop stops once the desired depth is reached or the game is terminated. When the desired depth is reached, the score of that final state is determined by the number of consecutive coins present on the board for both agents. If the tree results in a win, the maximum score is returned and if it results in a loss, the minimum score is returned. A tie is given a score of 0. Normally, the deeper the search depth, the stronger the policy of the mini-max agent.

However, a mini-max agent does not form a pleasant opponent, since even the simplest minimax agent with depth 1 will always block a winning move by design. Thus, to win against the smallest depth mini-max agent, a scenario has to be created where two possible winning moves have emerged. This ensures that even when the mini-max bot blocks one of them, the game can still be won by playing the other. This is a rather hard scenario to obtain, and as the depth of the mini-max agent increases, the mini-max agent can anticipate this as well, making it near impossible to beat as a human, even with a depth of only 3. An ideal bot will sometimes fail to see a potential win for the opponent, as humans would also do. The mini-max implementation used in this project is based on a mini-max connect four agent by Galli (2019). Videgain and Sanchez (2021) explain the mini-max algorithm in more depth.

Since mini-max uses a fixed policy and doesn't learn from the algorithm, playing against mini-max can be seen as single-agent RL as described in section 2.1. However, this mini-max agent has been made available as a Tianshou `BasePolicy` object so that it can be used in the same multi-agent training pipeline that is used for self-play. The idea behind playing against a mini-max agent is to provide a strong opponent to learn from for the RL agents. As it is assumed that increasing the depth of the mini-max agent will increase its policy's strength, changing this depth allows for creating a league-based training setting as described in 2.2. The best scoring RL agent for each of these iterations in the league based training loop would then be incrementally better over the previous iteration. This would ideally provide varying difficulties for the connect four RL bot. Paper notebook 11 describes this in more depth and is available on the GitHub repository of this project (Bontinck, 2022).

Evaluation of connect four bots

The previous chapter discussed the two DQN agents and the Rainbow agent that were implemented for this project. Section 2.4 discussed how the learned policy of the agents would be used as a computer-opponent in the connect four game and thus should be subjectively pleasing to play against rather than objectively good according to a specific metric. This chapter will aim to evaluate the policies obtained throughout various experiments with these RL agents.

4.1 Objective evaluation of the emerged policies

As discussed in section 2.4, objective evaluation metrics of the RL agents have small value in this feasibility study. An agent that wins all the time is objectively very good, but since connect four is a solved game and a perfect policy for the agent playing as player 1 exists, the objectively best agent would be one that is unbeatable, making it rather useless as a computer-controlled opponent in a video game. Likewise, it could be argued that an agent who wins in very few steps is objectively better than one that wins in many steps. However, when letting an RL agent train against a random agent, the RL agent simply learns to stack coins in a singular column. This same behaviour is also present when doing self-play and only providing rewards for a win, loss or a tie. This is the case for both DQN agents as well as for the rainbow agent, although some agents seem to learn to block some winning moves as well, yet stick to the stacking strategy. This behaviour is statistically sound when playing against a random agent, as the random agent would only block the vertical win 3 out of 7 times, resulting in a win in four moves 4 out of 7 times for the RL agent. However, in self-play, this behaviour is not statically sound and it is very likely that given more training iterations, these policies would diverge from the stacking behaviour into a better policy, as some signs of blocking behaviour emerged after half an hour of CUDA training. However, the metric based on wins and favouring shorter average game length would score this stacking policy highly, but it is almost worthless in human play. An example of this bad policy is shown in figure 4.1.

When optimising for a metric where longer games are favoured, with the idea that playing a long game requires the blocking of winning moves, a similar trend occurs where a good scoring policy is worthless in human play. Indeed, a policy which gets draws through self-play every time has maximal game length and maximal summed scores since both agents are positively rewarded for a draw. This policy can be achieved by rewarding the agents of either proposed algorithm for making moves and coming to ties. Since the maximum reward is obtained when getting a tie, the agents learn a policy that goes for ties. This involves deliberately ignoring win potentials and learning to play as a team with the other agent in self-play rather than playing against it. Thus, the resulting algorithm is again useless in human play.

One interesting objective evaluation that can be made is the performance of an agent playing as player 1 vs it playing as player 2. Alderton et al. (2019) concluded for their connect four agents

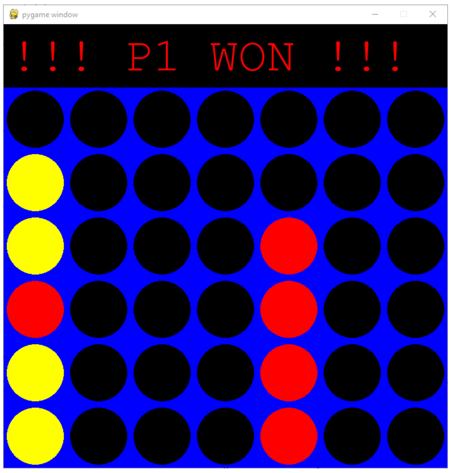

Figure 4.1: The MLP DQN agents having both learned stacking policies through self-play, with the agent playing as player 1 having some blocking behaviour as well. This policy would score good on some objective measures but is near worthless as a computer-opponent.

that an agent playing as player 1 in self-play is far more likely to win over an agent playing as player 2. They hinted that this could be due to the solved nature of the connect four game where an optimal policy exists for player 1 to ensure a win every time. When looking at the training results of a Rainbow agent against a varying depth mini-max agent given in Table 4.1 it becomes clear when playing as player 1, the agent does indeed learn faster and the best policy is indeed better. However, the found policy is far from the optimal policy that results in a win each time, as will become clear when looking at the win rate against even a simple random agent. Thus, we suspect the difference between the performance of playing as player 1 or player 2 originates from the fact that player 2 always has one coin less on the board when doing his move.

Another interesting objective metric is looking at the win rate of the trained RL agent against a random agent. This can give some insight into how good a given policy is since a random agent should be relatively simple to beat. Figure 4.2 shows the win rate against the random opponent for the best-found policy of all three RL agents. This further shows that the RL agent playing as agent 1 is indeed better than the best-found policy learned when playing as player 2. It also becomes clear that the Rainbow policy seems to outperform the DQN based policies. The difference between the MLP based DQN and CNN based DQN is negligible. This corresponds with the findings surrounding the human-play behaviour of these algorithms, although there the Rainbow policy is far better than the DQN policies.

4.2 Usability of deep Q-network policy as connect four bot

The two DQN approaches, one using a MLP-based model and one using a CNN-based model, showed similar trends throughout all experiments when used in human-play. When training the RL agents through self-play and only providing a reward for wins, losses and ties, the learned policy is the stacking behaviour discussed in section 4.1 and shown in Figure 4.1. This was the case for the best-found policy and final policy after training on 1 million steps in the connect

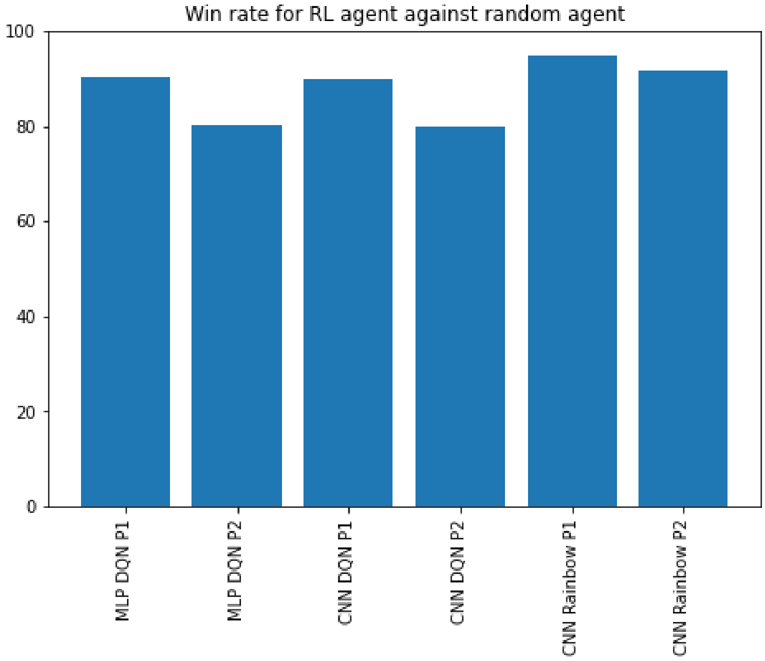

Figure 4.2: Win rate of the best found RL policy against a random agent. All three of the discussed policies are used once as player 1 and once as player 2. The results are averaged over 1000 games.

four game. This was done through 1000 epochs consisting of 1000 steps per epoch and updating the agent's policy every 64 steps. This behaviour is near worthless in human play. When also providing a reward for making moves, the agents learn to play together as was also already discussed in section 4.1. Since this agent assumes the other agent wouldn't go for a winning move as it also wants to prolong the game for as long as possible, the final obtained policy through a similar training procedure as before is again non-satisfactory in human play.

In an attempt to improve this behaviour, an experiment was set up inspired by a league-based approach. Alternating in varying frequencies, one agent was frozen so that it doesn't learn whilst the other was unfrozen so that it does learn. This was done for a decreasing amount of epochs per freezing alternation, but an increasing amount of alternations in 7 different experiments for both DQN policies. Each time the new experiment started both agents with the best-found policy of the previous experiment. According to the definition given in section 2.1 this makes the problem a single-agent RL problem and this reduces the non-stationary problem. This was done for a considerable amount of training time and exact details can be found in paper notebooks 7 and 8 which are available on the GitHub repository of this project (Bontinck, 2022). However, this seems to result in the agents overfitting to each other's policy rather than a more pleasing human play policy. This can be seen by the agent's being capable of winning a lot against the frozen opponent but losing just as much when frozen itself.

No experiments on the DQN agents were done using the reward for making a blocking move, as it was only introduced when working with Rainbow agents. This leaves the DQN agents with no real pleasing found policy for human play. It is noted that if an experiment were to be done using a reward for making blocking moves, it is likely it would cause a more promising policy to emerge.

The DQN agents likely suffer from something that is called catastrophic forgetting. This phenomenon occurs when a model is training on new unseen data and forgetting the things it has learned in the past. This could explain why the alternating freezing of the agents didn't yield a more pleasing human-play result since it could be possible the learned blocking techniques in previous iterations are being forgotten in new iterations. The fact that the batch size is 1 for these DQN agents also doesn't aid with this potential problem. However, known issues when combining the Tianshou library's batch mechanism and multi-agent policies forced the DQN implementations of this paper to work with a batch size of 1. Catastrophic forgetting is a common issue with DNNs as further explained by Pfülb et al. (2019). The performed experiments described above can all be found in paper notebooks 4 to 8 which are available on the GitHub repository of this project (Bontinck, 2022).

4.3 Usability of Rainbow policy as connect four bot

Since the Rainbow policy is known to be both more sample-efficient and better performing than DQN, as discussed in section 3.7, even more effort was put into making a pleasing Rainbow policy for human play. This policy doesn't suffer from the issue of having to use a batch size of 1 and uses a batch size of 64 throughout all experiments.

When using a reward for making a move as well as a reward for winning, losing and getting a tie, the learned policy is more interesting than was the case for the team playing agents with DQN. This is shown in Figure 4.3 where the yellow player performs blocking moves in self-play, even though no specific reward is given for playing a blocking move. However, when greeted with the chance to win, as was the case for getting a horizontal win in the second row from the bottom, the yellow agent doesn't seem to go for this winning move. This is likely since it wants to prolong the game and go for a tie rather than go for a win now, as the latter yields a lower total reward. Although not very frequent, this training procedure did result in some tie games. Since the second player has some move blocking behaviour, it forms a somewhat interesting computer-opponent in human play, although it mostly seems to block vertical wins and almost always fails to block horizontal or vertical wins.

Since the policy from rainbow showed some promising signs of intelligence, it was opted to help it become better by rewarding it for making a blocking move. This reward strategy was already discussed in section 3.4. By doing so, the agent became significantly more powerful. When playing against the best-found policy for player 1 of this training procedure, games like the one shown in Figure 3.1 are obtained through human play. The agent manages to block vertical and horizontal wins early on but fails to block the final diagonal winning move made by the human. When playing against this bot multiple times, a pleasant experience can be had, but some flaws do show up. The stacking coin strategy isn't blocked when done in the far left or right corner of the board, as is shown in Figure 4.4. It is expected that this follows from the fact that through the convectional layer, centre columns are present in multiple kernel passes whilst the edge columns are only present in one pass per row. It is assumed that a longer training procedure would likely improve this behaviour, although it was not feasible given the computational power at hand. Similarly, the blocking behaviour of horizontal rows works well in the early stages of the game, where the lower rows are played but when playing in the upper rows of the board, the blocking behaviour seems to disappear only to be present in very specific boards. This is likely

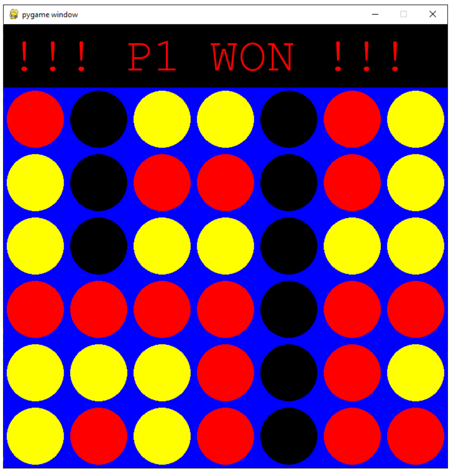

Figure 4.3: Rainbow policy self-play in an environment where a reward is given for making a move. The policy tends to go for getting as many moves as possible. For example, the agent playing as yellow does perform win blocking moves but doesn't do moves that would result in a win when given the opportunity.

again caused by the fact almost all training samples contain player coins in the lower rows of the board but coins in the higher rows of the board are rather rare. Training for a longer period could again possibly solve this issue.

When using the same freezing technique as was done for the DQN algorithms, the algorithm once again overfits to the opponent's policy and/or suffers from catastrophic forgetting. This results in a high win rate against the frozen agent but a worse human play policy. The performed experiments described above can all be found in paper notebooks 9 and 10 which are available on the GitHub repository of this project (Bontinck, 2022).

4.4 Viability of varying difficulty bot performance

In an attempt to further increase the performance of the Rainbow agents, a different approach to league-based training is done. Instead of using a frozen agent to play against, an increasing depth mini-max agent is played against. The ideology behind this is explained in section 3.8. Since the mini-max calculations take a considerable amount of time, experimenting once as player 1 and once as player 2, took over 72 hours. Each iteration of the mini-max opponent was trained for either 250 epochs or when an average test score of at least 10 was obtained. The latter test score would correspond with an average combination of winning (5) and making 5 blocking moves ($5 * 1$), getting a tie (3) and making 7 blocking moves ($7 * 1$) or losing (-5) but making 15 blocking moves ($15 * 1$). All of these behaviours would be pleasant human play behaviours. The training results are given in Table 4.1.

However, as was discussed in section 2.2, league based opponents should not only increase in difficulty, but they should also not be too predictable to combat potential overfitting on the opponent's policy rather than better learning to play the game. This behaviour was present when doing the frozen agent approach, and it was thought that the adaptation from the mini-

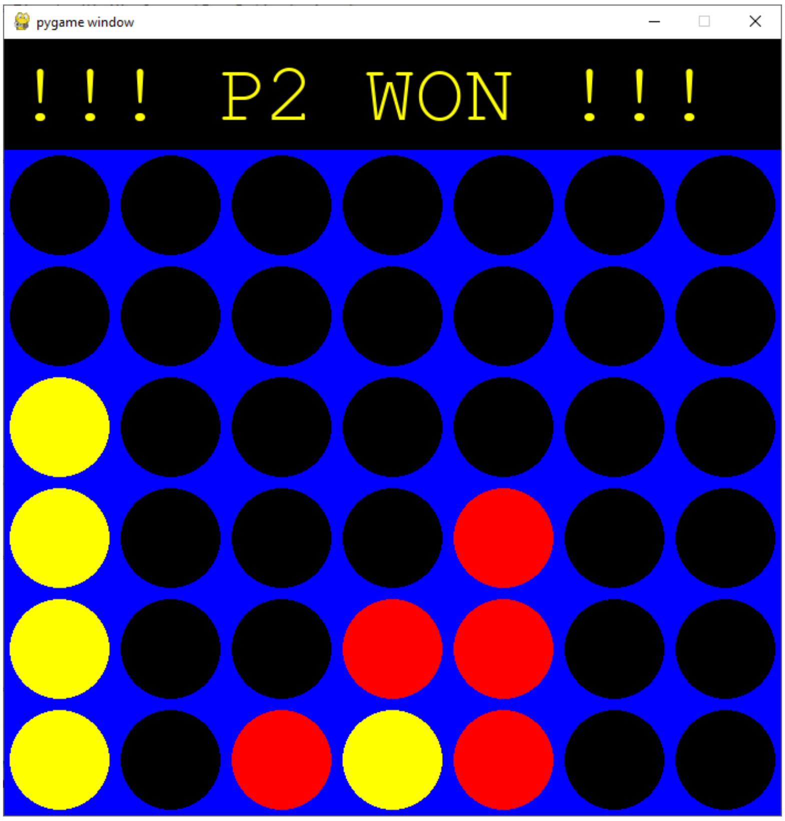

Figure 4.4: Sample connect four game with the best-found rainbow policy (player 1, red) from the paper notebook 9, and a human. The best-found policy struggles in blocking winning horizontal moves placed in the far left corner or far right corner.

max agent to a certain move would reduce this overfitting behaviour. However, the final policies obtained from this experiment were not that satisfactory in human play, hinting that the same issues as was the case when freezing the opponent occurred. Although this time it is more likely to be a type of catastrophic forgetting rather then overfitting and ideally the experiment should be redone using a decaying learning rate to try and mitigate this problem. However, due to the lengthy computations, this was not feasible for this project. If the experiment was redone using a decaying learning rate and potentially using other tricks to help and combat catastrophic forgetting, the author of this paper believes this approach could provide varying difficulty bots.

Mini-max depth	**Player 1 best test score**	**Player 2 best test score**
1	10.82 ± 0.24 in epoch 182	9.78 ± 1.56 in epoch 186
2	10.70 ± 0.00 in epoch 45	10.20 ± 0.00 in epoch 217
3	11.30 ± 0.00 in epoch 11	10.90 ± 0.00 in epoch 73
4	10.48 ± 3.66 in epoch 6	7.10 ± 0.00 in epoch 132
5	10.90 ± 0.00 in epoch 6	6.90 ± 0.00 in epoch 27

Table 4.1: Results of letting Rainbow agents play against a varying depth mini-max agent. Maximum amount of training epochs is 250 per mini-max depth and the stopping criteria is set to a test score of over 10.

Discussion

Having introduced both reinforcement learning (RL) and multi-agent reinforcement learning (MARL), the previous chapters discussed the implementation and evaluation of connect four RL agents. This chapter discusses all of these previous chapters and the discussed findings. It finishes off with closing remarks and potential future work.

5.1 Using general purpose libraries for custom tasks

Part of the reason that using RL agents as a way of providing a computer opponent in simple games could be feasible, is the fact that many libraries exist which provide very complex RL algorithms with relatively small amounts of code. However, these libraries are tailored to work with only the most popular environments and most of them focus on single-agent RL. Even when MARL environments are supported, they have to be of a specific type, such as a Petting Zoo environment. This paper discussed that this Petting Zoo environment, contrary to the Gym environment, does not provide clear instructions on how to make a custom environment and thus some troubleshooting is to be expected. However, converting relatively simple Pygame based games to a Petting Zoo library should be feasible and thus using RL algorithm providing libraries such as Tianshou should also be able to be used on these games. Whilst this is great, the support for Petting Zoo and MARL in general is something that should be improved. For example, the Ray RLlib library by Liang et al. (2017) claims to have support for Petting Zoo environments but the Ray RLlib provided example fails to work as it gives undocumented errors. This is shown in the experimental notebook 4 available on the GitHub repository of this project (Bontinck, 2022). Likewise, the Tianshou library has multiple known issues with its support for MARL, such as the problem of using a batch size different than 1 for DQN agents in a multi-policy manager.

5.2 Reinforcement learning for indie developers

With over 70 GitHub commits and many hours put into this project, providing a RL bot as a computer opponent for connect four was arguably too hard to be feasible. After all, the development of the game would take less time than the development of a pleasing bot. However, the best-found Rainbow policy is a pleasing amateur level computer opponent for connect four. This is far better than rule-based policies such as mini-max, where the behaviour of the bot feels inhuman and some faults humans make can't be made with them. The CNN based Rainbow opponent proposed in this paper is therefore far more human and more pleasant than the mini-max agent is in playing connect four. However, even the best-found policy has clear flaws as discussed in section 4.3 and the attempt at making a varying difficulty policy failed as discussed in section 4.4. However, both of these issues seem solvable, one through longer training and the other through some form of parameter tuning. This is promising but for it to become feasible that indie developers use RL as video game opponents, even more sample-efficient RL algorithms should be proposed in combination with better documented and supported libraries as mentioned in section 5.1.

5.3 Future work

Section 4.3 and 4.4 proposed potential ways to improve the found CNN based rainbow policy and to provide a varying difficulty bot. Doing these proposed experiments is something that could be of interest to further back the claim that RL will ultimately be an amazing tool for providing video game opponents that are more human-like. Besides this, this project only focused on online training. Since these games often also support regular multiplayer, the games played between humans could potentially be used as a dataset for offline training. Finally, testing different RL algorithms on the created custom Petting Zoo environment can also give potentially great results.

List of abbreviations and acronyms

A

AI artificial intelligence.

ANN artificial neural network.

C

CNN convolution neural network.

D

DL deep learning.

DNN deep neural network.

DP dynamic programming.

DQN deep Q-network.

DRL deep reinforcement learning.

I

IRL in real life.

M

MARL multi-agent reinforcement learning.

MDP Markov decision process.

ML machine learning.

MLP multilayer perceptron.

R

RL reinforcement learning.

References

Alderton, E., Wopat, E., & Koffman, J. (2019). Reinforcement learning for connect four. https://web.stanford.edu/class/aa228/reports/2019/final106.pdf

Alzubaidi, L., Zhang, J., Humaidi, A. J., Al-Dujaili, A., Duan, Y., Al-Shamma, O., Santamaria, J., Fadhel, M. A., Al-Amidie, M., & Farhan, L. (2021). Review of deep learning: Concepts, CNN architectures, challenges, applications, future directions. *Journal of Big Data*, *8*(1). https://doi.org/10.1186/s40537-021-00444-8

Anthony, S., Speckner, C., & Jentzsch, T. (2021). Stella: A multi-platform atari 2600 vcs emulator. https://stella-emu.github.io/index.html

Bellemare, M. G., Naddaf, Y., Veness, J., & Bowling, M. (2013). The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, *47*, 253–279.

Bellemare, M. G., Dabney, W., & Munos, R. (2017). A distributional perspective on reinforcement learning. https://doi.org/10.48550/ARXIV.1707.06887

Bontinck, L. (2022). *Reinforcement learning at vub 2021 - 2022* [GitHub commit: TODO]. Retrieved June 6, 2022, from https://github.com/pikawika/VUB-RL

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., & Zaremba, W. (2016). Openai gym.

Buşoniu, L., Babuška, R., & Schutter, B. D. (2010). Multi-agent reinforcement learning: An overview. *Innovations in multi-agent systems and applications - 1* (pp. 183–221). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-14435-6_7

Campbell, M., Hoane, A., & Hsu, F.-h. (2002). Deep blue. *Artificial Intelligence*, *134*(1-2), 57–83. https://doi.org/10.1016/s0004-3702(01)00129-1

Dawson, R. (2020). Learning to play connect 4 with deep reinforcement learning. https://codebox.net/pages/connect4

De Smet, R. (2020). *Vub latex huisstijl* [GitHub commit: d91f55...]. Retrieved November 2, 2020, from https://gitlab.com/rubdos/texlive-vub

DiGiovanni, A., & Zell, E. C. (2021). Survey of self-play in reinforcement learning. https://doi.org/10.48550/ARXIV.2107.02850

Feriani, A., & Hossain, E. (2020). Single and multi-agent deep reinforcement learning for ai-enabled wireless networks: A tutorial. https://doi.org/10.48550/ARXIV.2011.03615

Galli, K. (2019). *Connect4 with ai* [GitHub commit: 8db5827...]. https://github.com/KeithGalli/Connect4-Python/blob/master/connect4_with_ai.py

Hessel, M., Modayil, J., van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M., & Silver, D. (2017). Rainbow: Combining improvements in deep reinforcement learning. https://doi.org/10.48550/ARXIV.1710.02298

Jaderberg, M., Czarnecki, W. M., Dunning, I., Marris, L., Lever, G., Castaneda, A. G., Beattie, C., Rabinowitz, N. C., Morcos, A. S., Ruderman, A., Sonnerat, N., Green, T., Deason, L., Leibo, J. Z., Silver, D., Hassabis, D., Kavukcuoglu, K., & Graepel, T. (2018). Human-level performance in first-person multiplayer games with population-based deep reinforcement learning. https://doi.org/10.48550/ARXIV.1807.01281

Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. https://doi.org/10.48550/ARXIV.1412.6980

Lazic, N., Boutilier, C., Lu, T., Wong, E., Roy, B., Ryu, M., & Imwalle, G. (2018). Data center cooling using model-predictive control. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, & R. Garnett (Eds.), *Advances in neural information processing systems*. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2018/file/059fdcd96baeb75112f09fa1dcc740cc-Paper.pdf

Liang, E., Liaw, R., Moritz, P., Nishihara, R., Fox, R., Goldberg, K., Gonzalez, J. E., Jordan, M. I., & Stoica, I. (2017). Rllib: Abstractions for distributed reinforcement learning. https://doi.org/10.48550/ARXIV.1712.09381

Lindstrom, L., Dudfield, R., Shinners, P., Dudfield, N., & Kluyver, T. (2022). Pygame. https://www.pygame.org/

Liu, S., Cao, J., Wang, Y., Chen, W., & Liu, Y. (2021). Self-play reinforcement learning with comprehensive critic in computer games. *Neurocomputing*, *449*, 207–213. https://doi.org/10.1016/j.neucom.2021.04.006

Mendonca, M. R. F., Bernardino, H. S., & Neto, R. F. (2015). Simulating human behavior in fighting games using reinforcement learning and artificial neural networks. *2015 14th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames)*. https://doi.org/10.1109/sbgames.2015.25

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., & Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, *518*(7540), 529–533. https://doi.org/10.1038/nature14236

Nayak, N. (2019). *Connect four game using pygame and numpy* [GitHub commit: 9105a9b...]. https://github.com/Nihar99/pygame

Papoudakis, G., Christianos, F., Rahman, A., & Albrecht, S. V. (2019). Dealing with nonstationarity in multi-agent deep reinforcement learning. https://doi.org/10.48550/ARXIV.1906.04737

Pfülb, B., Gepperth, A., Abdullah, S., & Kilian, A. (2019). Catastrophic forgetting: Still a problem for dnns. https://doi.org/10.48550/ARXIV.1905.08077

Schrittwieser, J., Antonoglou, I., Hubert, T., Simonyan, K., Sifre, L., Schmitt, S., Guez, A., Lockhart, E., Hassabis, D., Graepel, T., Lillicrap, T., & Silver, D. (2019). Mastering atari, go, chess and shogi by planning with a learned model. https://doi.org/10.48550/ARXIV.1911.08265

Shao, K., Tang, Z., Zhu, Y., Li, N., & Zhao, D. (2019). A survey of deep reinforcement learning in video games. https://doi.org/10.48550/ARXIV.1912.10944

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., 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. *Nature*, *529*(7587), 484–489. https://doi.org/10.1038/nature16961

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K., & Hassabis, D. (2017). Mastering chess and shogi by self-play with a general reinforcement learning algorithm. https://doi.org/10.48550/ARXIV.1712.01815

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K., & Hassabis, D. (2018). A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, *362*(6419), 1140–1144. https://doi.org/10.1126/science.aar6404

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., Sifre, L., van den Driessche, G., Graepel, T., & Hassabis, D. (2017). Mastering the game of go without human knowledge. *Nature*, *550*(7676), 354–359. https://doi.org/10.1038/nature24270

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning* (2nd ed.). Bradford Books.

Szajowski, K. (2019). Stochastic games and learning. *Encyclopedia of systems and control* (pp. 1–7). Springer London. https://doi.org/10.1007/978-1-4471-5102-9_33-3

Terry, J. K., Black, B., Grammel, N., Jayakumar, M., Hari, A., Sulivan, R., Santos, L., Perez, R., Horsch, C., Dieffendahl, C., Williams, N. L., Lokesh, Y., Sullivan, R., & Ravi, P. (2020). Pettingzoo: Gym for multi-agent reinforcement learning. *arXiv preprint arXiv:2009.14471*.

Videgain, S., & Sanchez, P. G. (2021). Performance study of minimax and reinforcement learning agents playing the turn-based game iwoki. *Applied Artificial Intelligence*, *35*(10), 717–744. https://doi.org/10.1080/08839514.2021.1934265

Wang, Z., Schaul, T., Hessel, M., van Hasselt, H., Lanctot, M., & de Freitas, N. (2015). Dueling network architectures for deep reinforcement learning. https://doi.org/10.48550/ARXIV.1511.06581

Watkins, C. J. C. H., & Dayan, P. (1992). Q-learning. *Machine Learning*, *8*(3-4), 279–292. https://doi.org/10.1007/bf00992698

Watkins, C. J. C. H. (1989). Learning from delayed rewards.

Weng, J., Chen, H., Yan, D., You, K., Duburcq, A., Zhang, M., Su, H., & Zhu, J. (2021). Tianshou: A highly modularized deep reinforcement learning library. *arXiv preprint arXiv:2107.14171*.

Ye, W., Liu, S., Kurutach, T., Abbeel, P., & Gao, Y. (2021). Mastering atari games with limited data. https://doi.org/10.48550/ARXIV.2111.00210