University of Padova

Department of Mathematics "Tullio Levi-Civita"

Master Thesis in Data Science

You might have a chance to beat your grandparents at Briscola thanks to Deep Reinforcement Learning

Master degree thesis

Supervisor
Professor Gian Antonio Susto

Master Candidate
Alberto Sinigaglia
ID: 2044712

Academic year 2022-2023

Abstract

Recently, Reinforcement Learning, in conjuction with Deep Learning, is proving its ability to tackle very hard problems, thought to be decades away to be solvable.

The first real demonstration of their power it's been shown with the work from DeepMind with AlphaGo, the first Artificial Intelligence able to not only surpass the average player ability, but to defeat the current world champion.

Even though AlphaGo, and its following version AlphaZero, have solved problems though to be impossible, such problems are know in the literature as fully observable, and completely characterizable through a known model by assuming the opponent is playing optimally, thus allowing to use search techniques such as Monte Carlo Tree Search to further improve the final agent.
Later, Noam Brown et al. developed an Artificial Intelligence able to play Poker at master level called Libratus, which instead is a partial information game, as an agent doesn't have access to all the information that it needs to decide which action to take.

With this work, we aim as solving the game of Briscola, which does not only fall under the umbrella of partial information games, but it's also a stochastic game, where the game cannot be characterized by independent repetition, but instead composed by sequence of stages, thus combining Reinforcement Learning techniques for the optimization, Deep Learning models for the agent model, and Game Theory technique for the learning procedure.

"The man who asks a question is a fool for a minute,
the man who does not ask is a fool for life."

— Confucius

Acknowledgements

First and foremost, I am deeply grateful to my supervisor, prof. Gian Antonio Susto, for their guidance, expertise, and continuous encouragement throughout the entire research process.

My deepest appreciation goes to my family for their understanding and encouragement, which provided me with the strength to pursue this academic journey.

I am profoundly grateful to my exceptional colleagues and dear friends, Giulia, Chiara, Beatrice, Marco M., Marco B., Riccardo, and Sofia. Their steadfast support, insightful suggestions, and unwavering belief in my abilities were instrumental in surmounting challenges and maintaining my motivation throughout this entire odyssey.

Padova, September 2023 Alberto Sinigaglia

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Reinforcement Learning is a learning approach based on trial and errors. It's mainly composed by an agent, which interacts with an environment, taking decision through a policy, which is trained to maximize a specific reward function.
It's used in a variety of fields, such as:

∗ Computer vision: Reinforcement learning is used in computer vision in tasks like object recognition and attention modules.

∗ Natural Language Processing: lately, with the introduction of ChatGPT[1], Reinforcement learning is heavily used to align the model output with the human preference

∗ Games: by their nature, games are a perfect examples of decision making scenarios, where imitation can only bring an agent to be as good as its training data, thus never surpassing the human abilities

Recent advances in Deep Learning, with the introduction of neural networks, Reinforcement Learning is starting to be applied to always more complex scenarios, giving birth to the most impactful milestones in the AI history, such as AlphaGo[2] in 2018.
The aim of the thesis is to solve the game of Briscola combining Reinforcement Learning for the learning paradigm, Deep Learning for the agent formalization, and Game Theory for the training procedure.

1.1 Thesis outline

The remaining part of the thesis will be structured as follows:

∗ **Chapter 2**: brief introduction to the game of Briscola

∗ **Chapter 3**: brief introduction to Reinforcement Learning

∗ **Chapter 4**: brief introduction to Game Theory

∗ **Chapter 5**: introduction to Deep Reinforcement Learning

∗ **Chapter 6**: formalization of the problem

∗ **Chapter 7**: definition of the agent architecture

∗ **Chapter 8**: definition of the evaluation metrics

∗ **Chapter 9**: training of the agent and result

∗ **Chapter 10**: conclusions and future directions

Chapter 2

The game of Briscola

Briscola is a card game very popular in Italy. It can be played either in 1v1 or in 2v2 settings. The deck is composed by 40 cards, each of which has a specific value depending on its numerical value and its suit. Every player place a card on the table, one at a time, and the whole game is based on the idea of beating the highest already-placed card.

At the end of a turn, the player with the highest value card wins, and the next turn will start from him. Once the full deck has been used, each player will count the points he got, and the highest one wins the game.

In the case of 2v2, the points between the 2 player in each team are shared.

2.1 The cards

The deck of cards used to play Briscola is composed by 40 cards, split in 4 different suits called coins, swords, batons, cups. Each suit is composed by 10 cards, with figures from 1 to 7, jack, knight, king.

Italian name	English name	Value
Asso	Ace	11
Due	Two	0
Tre	Three	10
Quattro	Four	0
Cinque	Five	0
Sei	Six	0
Sette	Seven	0
Fante	Jack	2
Cavallo	Knight	3
Ree	King	4

Table 2.1: Cards name and respective points.

2.2 The rules

After and initial phase where the deck is shuffled to ensure randomness, at each player are handed 3 cards. Then, another card is drawn from the deck, which will be considered the Briscola for the entirety of the game, and will be placed face up underneath the deck.
The game can now start, from the rightmost player from the dealer, which will place a card on the table: the suit of that card will become the lead suit for the turn, and the same holds for all the following turns.
From there, anti-clockwise, all players will place a card on the table. Once all of them have played a card, if no briscola have been used, then the winner player is the one with the card with the highest score of the leading suit. If instead at least a briscola has been played, then the winner is the player which placed the highest briscola.
In the case which there is a tie, the card with the highest numerical value wins.
The winner collects all the placed cards, draws one card from the deck, and then all the others, and will have to place the first card on the table at the next turn.
Once all the cards have been placed and the deck is empty, each player will count the points he gained during the game, and the one with the highest final score wins.

FORTUNA CONTRA
CHI HA
SE TI PERDI DAN
NON TI FIDAR DI ME SE IL CUOR TI MANCA
PER UN PUNTO MARTIN PERSE LA CAPA

Figure 2.1: Example of Briscola cards

Chapter 3

Reinforcement Learning

Reinforcement learning is a learning approach inspired from the way humans learn. It is based on the idea of trial and error and it aims at surpassing human knowledge, thus imitation is not enough.
By its nature, Reinforcement Learning inherits aspects from Optimization, Mathematics, Neuroscience and Psychology.

The main hypothesis behind it is the following:
That all of what we mean by goals and purposes can be well thought of as maximization of the expected value of the cumulative sum of a received scalar signal (reward).

All problems in Machine learning can be rephrased in such a way to fit this definition, such as all problems that can be formulated as *Empirical risk minimization*, thus making it a generalization of them.
Its main focus however is about solving sequential decision making problems, as that is, for the founders of the field, the best formulation of human intelligence that we have so far, making RL a plausible path to Artificial General Intelligence.

So far, RL has been applied in very different fields, such as:

∗ Autonomous driving

∗ Games

∗ Natural Language Processing

∗ Computer vision

∗ Robotics

∗ Control System

∗ Recommender Systems

In all of these fields, it has allowed to achieve super human performances, of which the most known are:

∗ AlphaGo[2]: first AI to achieve superhuman performances in the game of Go

∗ ChatGPT[1]: most successful SaS in the history of mankind

∗ Tesla self-driving algorithm

∗ Boston Dynamics robots

3.1 Framework definition

Every reinforcement learning problem is defined by multiple entities that characterize the learning goal and settings: a *policy*, a *reward function*, a *value function*, and a *environment*.

The main idea behind those component is the following:
An agent, characterized by a policy π observes at time t the state S_t from the environment from which decides to take action A_t thanks to which ends up at state S_{t+1} with a reward R_{t+1}.
The policy can be either deterministic, thus $A_t = \pi(S_t)$, or stochastic, thus $A_t \sim \pi(S_t)$, and in simple settings can be composed by a simple lookup table, otherwise it will be approximated by some function approximators.

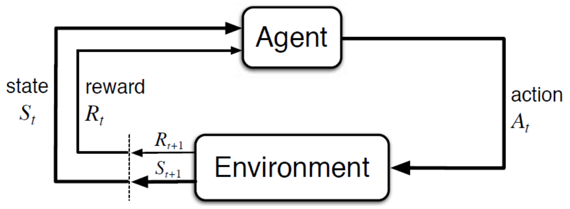

Figure 3.1: Interaction between main components of an RL problem

Given that R_t is a scalar reward that tells the agent how good was its last action, and given that past actions influences the future, the agent cannot only rely on maximizing the immediate reward greedily, but has to consider also how that decision will impact it's future. This consideration is well known in the field of dynamic programming (DP), field from which RL inherits a lot of properties and characteristics. However, in contrast to DP, in RL is not feasible to traverse the full problem tree and backtrack, as it would be too computationally expensive.
Thus, RL formulates the problem as an optimization problem, but still preserving its goal: in fact, the agent is asked to maximize the cumulative reward, called return, defined as:

$$G_t = R_{t+1} + R_{t+2} + ... = \sum_{i=t+1}^{\infty} R_i$$

However, this has a problem as it most likely will introduce infinite rewards, thus a γ parameter is introduced as a way to allow the infinite sum to converge, but still having a notion of future reward:

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

If $\gamma = 0$ then the agent will become greedy, else as $\gamma \to 1$ the agent will care more and more about its future.

3.2 Markov Decision Process

Markov Decision Process (MDP) are the mathematical formalization of an RL problem. It is an extension of Markov Reward Process (MRP) which are also an extension of Markov Process (MP).

A Markov Process is characterized by a set of states $\mathcal{S}$ and a transition matrix $\mathcal{P}$, which describes the probability to end in state s' starting from state s, thus defined as:

$$\mathcal{P}_{ss'} = P(S_{t+1} = s' | S_t = s)$$

MRP introduces the notion of discount $\gamma \in [0,1]$ and reward $\mathcal{R}$, which describes how inline with the end goal that transition was, defined as:

$$\mathcal{R}_s = \mathbb{E}[R_{t+1}|S_t]$$

With MRP we can already define the concept of returns, as the sum of discounted rewards from time t of a trajectory under the transition matrix $\mathcal{P}$, defined as:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + ... = \sum_{i=0}^{\infty} \gamma^i R_{t+i+1}$$

Since G_t describes the return of a single trajectory, we can instead define the function V that describes the expected return:

$$V(s) = \mathbb{E}[G_t|S_t = s]$$

By its structure, the return G_t can be recursively defined as:

$$\begin{aligned} G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... \\ &= R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + ...) \\ &= R_{t+1} + \gamma G_{t+1} \end{aligned}$$

With this decomposition, we can also rephrase the definition of the value function V:

$$\begin{aligned} V(s) &= \mathbb{E}[G_t|S_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma G_{t+1}|S_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma V(S_{t+1})|S_t = s] \end{aligned}$$

Finally, MDPs introduces the notion of action $\mathcal{A}$, thus redefining the transition probability as:

$$\mathcal{P}^a_{ss'} = P(S_{t+1} = s'|S_t = s, A_t = a)$$

and the reward function as:

$$\mathcal{R}^a_s = \mathbb{E}[R_{t+1}|S_t, A_t = a]$$

Thanks to the introduction of actions, we can now have an agent that takes decisions at each time, thus needing the definition of the policy π, which will specify the intention of the agent on a given state:

$$\pi(a|s) = P(A_t = a|S_t = s)$$

A policy fully defines the agent behavior, and will define how the agent learns over time. Since the value function V depends only on the state, there is no need to redefine it; however, a new function Q is introduced to take advantage of the ability of the agent to take actions, defined as follows:

$$Q(s,a) = \mathbb{E}[G_{t+1}|S_t = s, A_t = a]$$

By taking advantage of the definition of the return G_t, as previously done for the value function V, we can redefine the Q function recursively:

$$\begin{aligned} Q(s,a) &= \mathbb{E}[G_t|S_t = s, A_t = a] \\ &= \mathbb{E}[R_{t+1} + \gamma G_{t+1}|S_t = s, A_t = a] \\ &= \mathbb{E}[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})|S_t = s, A_t = a] \end{aligned}$$

The relation between the function V and Q can be seen by introducing each other in their definitions as follows:

$$V(s) = \mathbb{E}_{a\sim\pi(a|s)}[Q(s,a)]$$
$$Q(s,a) = \mathbb{E}[R_{t+1} + \gamma \mathbb{E}_{s'\sim\mathcal{P}^a_{ss'}}[V(s')]|S_t = s, A_t = a]$$

However, learning those functions is the part of the problem that makes it complex. For instance, assuming there exists a way of learning them, then we can derive the optimal action by just picking $a_t = argmax_a \pi(a, s_t)$, which corresponds to $a_t = argmax_a Q(a, s_t)$.

Learning them is a complex task, that gave birth to multiple algorithms, all of which have to deal with the *exploration/exploitation dilemma*: during the training, the agent wants to explore, to discover new ways of achieving its end goal, yet it needs to exploit the knowledge that it has already gained, two opposing sides.

3.3 Learning procedures

The first way of estimating these V and Q functions, is to just keep an average of the reward for each pair of s, a, and taking some actions using the current estimate. This last part "taking some actions" is what is going to define the trade-off between exploration and exploitation, and it's induced by the chosen policy type.

There exist many type of policies, and following are the most known and most used ones:

* ϵ-greedy: with probability ϵ take a suboptimal action, with probability $1-\epsilon$ the optimal action

* Upper-Confidence-Bound (UCB): we give a bonus to the actions that are less frequently been taken, which are the ones where we are less confident about the current estimation, and we pick the optimal one by summing to its Q value, the exploration bonus

* Gradient policy: given the current estimate $Q(a, s)$, we pick an action proportional to that estimate $\pi(a|s) \propto e^{Q(s,a)}$

Once picked a policy type, it's required to pick a learning procedure. At this point, many algorithms have been proposed, all of which falls in 2 groups: the ones for simple/tabular problems, and the real world ones.
The first one, targets problem which action and state space are discrete and relatively small, with small defined as the possibility to enumerate them in a matrix of size $S \times A$ in available hardware.
The second one, aims to tackle real world problems, where the state space is very large, if not continuous.
The first proposed technique uses Monte Carlo simulations to estimate the Q function of a specified policy π:

Algorithm 1: Monte Carlo estimation of Q function

```
Initialize for all $s \in S, a \in A(s)$ :
  $Q(s, a) \leftarrow$ arbitrary
  $\pi(s) \leftarrow$ arbitrary
  $Returns(s, a) \leftarrow$ empty list
while forever do
    Choose $S_0 \in S$ and $A_0 \in A(S_0)$, all pairs have probability $> 0$
    Generate an episode starting at $S_0, A_0$ following $\pi$ foreach pair $s, a$ appearing in
     the episode do
        $G \leftarrow$ return following the first occurrence of $s, a$
        Append $G$ to $Returns(s, a))$
        $Q(s, a) \leftarrow average(Returns(s, a))$
    end
    foreach $s$ in the episode do
        $\pi(s) \leftarrow argmax_a Q(s, a)$
    end
end
```

This algorithm has been extended in multiple ways, the main one gave girth to SARSA, which changes the update from the average, to a gradient update (assuming $Q(s,a)$ being Gaussian distributed):

$$Q'(a_t, s_t) = Q(s_t, a_t) - \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$

The main drawback of this learning procedure is that the update is performed only at the end of the procedure, and in some cases the problem does not have a terminal state.

To overcome this, Temporal difference learning has been proposed, and SARSA's equation previously cited exploits it. In fact, assuming that the estimate of the next state Q value is accurate, that is a valid way for updating the Q function, and it's guaranteed to converge to the optimal one.

The most used algorithm that follows this procedure is Q-learning:

Algorithm 2: Q-learning

```
Initialize for all $s \in S, a \in A(s)$ :
  $Q(s,a) \leftarrow$ arbitrary
  $\pi(s) \leftarrow$ arbitrary
  $Returns(s,a) \leftarrow$ empty list
while forever do
    initialize $S$ while $S$ is not terminal do
        Choose $A$ from $S$ using the policy $\pi$
        Take action $A$ and observe $R$ and $S'$
        $Q'(a_t, s_t) \leftarrow Q(s_t, a_t) - [r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$
        $S \leftarrow S'$
    end
end
```

Both Monte Carlo and Temporal Difference learning have their pros and cons: if on one hand MC is unbiased but computationally heavy, TD is biased but can learn while acting, so much computationally faster.

To overcome this limitation, some edits have been proposed. The first one, tackles the problem by using a trade-off between the two, called N-step temporal difference learning. If Temporal Difference uses 1 step return to estimate the update, N-step waits N time-steps before the update, exploiting the following equation:

$$Q(s_t, a_t) = R_{t+1} + \gamma R_{t+2} + ... + \gamma^N max_a Q(s_{t+N+1}, a)$$

This reasoning is then being further exploited by the algorithm known in the literature as $TD(\lambda)$, which takes a weighted average of all the N-step returns:

$$G_t^\lambda = (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_{t:t+n}$$

Even though this seems more computationally expensive, it's been shown to be equivalent to a second formulation that exploits the idea of keeping in memory the responsibility of each state for the current reward, and back propagating it also to those, mimicking a responsibility mechanism. This technique is known as eligibility trace [3] and follows the following steps:

Algorithm 3: Eligibility Traces

```
Initialize $E(s,a) = 0, \forall s \in \mathcal{S}, \forall a \in \mathcal{A}$
foreach step do
    $E_t(s,a) = \gamma\lambda E_{t-1}(s,a), \forall s \in \mathcal{S} \neq S_t \bigvee \forall a \in \mathcal{A} \neq A_t$
    $E_t(S_t, A_t) = \gamma\lambda E_{t-1}(S_t, A_t) + 1$
    $\delta_t = R_{t+1} + max_a Q(S_{t+1}, a) - Q(S_t, A_t)$
    $Q(s,a) = Q(s,a) + \alpha\delta_t E_t(s,a) \ \forall s \in \mathcal{S}, a \in \mathcal{A}$
end
```

3.4 Function value approximation

Up until now, all the presented methods were based on the idea that the state-action space could fit in memory, thus being small and discrete. However, real world problems have a very vast state space, and thus a very large state-action space, if not continuous.
For this reason, tabular methods are not suited to solve them. To overcome this problem, since the whole procedure is based on the idea of learning the functions V and Q, we can resort to use function approximators, thus learning:

$$V_\theta(s) \approx V(s)$$
$$Q_\theta(s,a) \approx Q(s,a)$$

The type of approximators depends on the task that the agent is asked to learn. Behind most of the latest breakthrough there are neural networks, that are widely used because of their generalization capabilities.
However, combining NN with RL is not straightforward: one of the fundamental assumptions of neural networks is the assumption of *i.i.d.* about the training dataset, which is violated by the RL since the samples are time correlated, and also non stationary, as the sample distribution depends on π, which has to be learnt during training.
Furthermore, the learning is usually done on-line and using temporal difference learning, to overcome problems such as infinite horizon environments.
On the other hand, all previously known techniques known for the tabular case, can be extended to the function approximation case just by changing the update rule using any of the gradient optimizers known in the literature, such as Adam.

$$\nabla_\theta Q_\theta(s,a) = \nabla_\theta(Q(s_t,a_t) - [r_{t+1} + \gamma \max_a Q(s_{t+1},a) - Q(s_t,a_t)])^2$$

3.5 Introduction to Deep RL algorithms

Deep reinforcement learning algorithms combine the power of deep learning and reinforcement learning to enable agents to learn complex and intelligent behaviors in a wide range of environments. These algorithms have achieved remarkable success in tasks such as game playing, robotics, and natural language processing, surpassing human-level performance in several domains.

On one had we have Reinforcement learning (RL), which is a branch of machine learning that focuses on training agents to make sequential decisions based on feedback from their interactions with an environment.

Deep learning, on the other hand, is a subfield of artificial intelligence that employs neural networks to model and learn complex patterns and representations from observations. Deep neural networks excel at automatically extracting useful features from high-dimensional spaces and are capable of capturing intricate relationships within the data, property fundamental for large scale reinforcement learning problems.

Deep reinforcement learning algorithms combine these two fields by utilizing deep neural networks as function approximators to estimate action values or policy functions. Instead of relying on manually engineered features, deep RL algorithms directly learn representations of states or state-action pairs from raw sensory input, enabling agents to handle complex and unstructured input spaces.

The integration of deep learning with reinforcement learning has led to several breakthroughs in the field. One of the most notable achievements is the success of deep Q-networks (DQN[4]), which introduced the concept of using deep neural networks to approximate the action-value function in RL.

Deep reinforcement learning algorithms have also made significant contributions to robotics. Agents equipped with deep RL algorithms have successfully learned to manipulate objects, navigate complex environments, and perform dexterous tasks. These advancements have paved the way for the development of autonomous robots capable of interacting and adapting to real-world scenarios.

However, deep reinforcement learning algorithms come with their own challenges. The high-dimensional and continuous action spaces present in many real-world problems can lead to difficulties in exploration and convergence. Additionally, the training of deep neural networks can be computationally expensive and requires careful tuning of hyper-parameters.

To address these challenges, researchers have proposed various improvements to deep RL algorithms, such as prioritized experience replay, trust region policies, and distributional RL. These advancements aim to enhance exploration, stabilize training, and improve the sample efficiency of deep reinforcement learning algorithms.

3.6 Deep Q-learning

Q-learning is the previously presented method, and it's been the first one being extended in real world scenarios using neural networks as function approximators, and the presented method is in fact called Deep Q-Learning [4].
The algorithm shares almost everything with the tabular case, with the addition of some tricks that allowed convergence:

∗ Batch Normalization: there were previous attempts to integrate NN with RL, with little to no success; in the paper the authors heavily underlines that Batch Normalization was the key for the training of the neural network, showing that the friction between RL and NN is not ignorable

∗ Experience Replay: to overcome the high correlation in the training data induced by the time-dependency of the samples from the emulators, they introduced an experience replay, over which the algorithm can sample old data, de-correlating the batch

∗ Target networks: Q-learning has a overestimation problem induced by the *max* operator in its update formula; to overcome this, the authors introduced a notion of target network, used only for the estimation of the target of Q-learning, which consists in a old freezed version of the current neural network.

The implementation takes advantage of the neural network ability to share parameters, and instead of estimating $Q(s, a)$, directly estimates $Q(s)$, outputting a vector of values corresponding to all the Q values, so that in a single forward pass they can have all the Q values.

A final, but not as relevant detail they added, is the use of Polyak averaging for the target network.
Even though in non convex losses taking a convex combinations of two points in space has no guarantee, the authors wanted to have the target to be a "slow" version of the actual network, so that the overestimation problem was as tone down as possible.
In following sections this problem will be cover much more and will be necessary mathematically defined and solved to guarantee the convergence of another class of algorithm.

Algorithm 4: Deep Q-learning

```
Initialize $Q_\theta$, $\hat{Q} \leftarrow Q_\theta$, $N$ batch size, $M$ reset iterations for target, experience replay
  memory $D$ and policy $\pi$, $\lambda \in [0, 1]$
Initialize $S_0$ **while** *forever* **do**
    Choose action $a$ from $\pi$, observe $r$, $s'$ and $done$
    Store transition $(s, s', a, r, done)$ in $D$
    Sample a minibatch of $N$ transitions from $D$
    Calculate the loss:
        $$L(\theta) = \sum_{i=0}^{N}(Q(s_i, a_i) - (r_i + \gamma max_a Q(s', a) \cdot (1 - done_i)))^2$$
     Update Q using $\nabla L(\theta)$ with any gradient optimizer.
    **if** *iteration* $\%M == 0$ **then**
        $\hat{Q} \leftarrow \lambda Q_\theta + (1 - \lambda)\hat{Q}$
    **end**
**end**
```

Deep Q-learning, while a powerful algorithm for reinforcement learning, also has some drawbacks that can impact its performance and practical applicability:

∗ Overestimation of Q-values: Deep Q-learning can suffer from overestimation of Q-values, especially in scenarios with high variance or sparse rewards. The use of a max operator during action selection can lead to an overestimation bias, resulting in suboptimal policies and slow convergence.

∗ Lack of Continuous Action Support: Deep Q-learning is primarily designed for discrete action spaces, as it relies on a discrete action-value function to estimate the Q-values. Adapting deep Q-learning to continuous action spaces requires additional techniques, such as discretization or function approximation, which can introduce additional challenges and limitations.

∗ Correlation of Samples: Deep Q-learning samples experiences from a replay buffer, which can lead to a correlation between subsequent samples. This correlation violates the independent and identically distributed (i.i.d.) assumption typically required by standard deep learning algorithms. The correlation of samples can result in unstable training and slow convergence.

∗ Limited Exploration: Deep Q-learning relies on an exploration strategy, such as epsilon-greedy, to balance exploration and exploitation. However, it can struggle to explore the state-action space effectively, particularly in large and complex environments. The algorithm might get stuck in suboptimal policies due to inadequate exploration.

∗ Non-Stationary Target Network: To stabilize training, deep Q-learning employs a separate target network to compute the target Q-values during the update process. However, the target network introduces a delay between the updates of the Q-network and the target values, making the learning process non-stationary. This delay can result in unstable training and slow convergence.

∗ Sensitivity to Hyperparameters: Deep Q-learning involves several hyperparameters, such as learning rate, exploration rate, and network architecture, which can significantly impact its performance. Finding appropriate values for these hyperparameters can be a challenging and time-consuming task. Inappropriate choices may lead to unstable training, slow convergence, or even divergence.

3.7 Policy Gradient

Policy gradient[5] methods are a class of reinforcement learning algorithms that enable intelligent agents to learn optimal policies in complex and dynamic environments. Unlike value-based methods that focus on estimating the value of different actions or state-action pairs, policy gradient methods directly optimize the policy itself, which determines the agent's actions based on the observed states.

In many real-world scenarios, such as playing games or controlling robots, it is challenging to define a precise value function that accurately captures the desirability of different actions or states. Policy gradient methods address this limitation by directly optimizing the policy's parameters, allowing agents to learn from trial and error experiences.

At the heart of policy gradient methods is the concept of a policy gradient, which is a technique for updating the policy parameters in a way that maximizes the expected cumulative reward. The policy gradient is computed by estimating the gradient of a performance objective, such as the expected return, with respect to the policy parameters. This gradient information guides the agent to adjust its policy in a way that increases the probability of selecting actions leading to higher rewards.

Policy gradient methods have gained significant attention and popularity in recent years due to their ability to tackle high-dimensional and continuous action spaces, making them suitable for tasks involving robotics, natural language processing, and other complex domains. They have shown impressive results in various applications, including playing video games, controlling autonomous vehicles, and optimizing recommendation systems.

One of the notable advantages of policy gradient methods is their ability to handle stochastic policies, allowing agents to explore a wide range of actions and potentially discover more effective strategies. Moreover, policy gradient methods can naturally handle both discrete and continuous action spaces, making them versatile and adaptable to different problem settings.

While policy gradient methods have shown great promise, they also come with their challenges. The most significant obstacle is the high variance in gradient estimates, which can lead to slow convergence or even instability during training. Researchers have proposed various techniques to address this issue, such as baseline subtraction, trust region policies, and entropy regularization.

In conclusion, policy gradient methods provide a powerful framework for training intelligent agents to learn optimal policies in reinforcement learning settings. By directly optimizing the policy parameters, these methods can handle complex and high-dimensional action spaces, making them applicable to a wide range of real-world problems. With ongoing research and advancements, policy gradient methods continue to push the boundaries of reinforcement learning and hold great potential for future applications.

In order to achieve all of this, there is the need to estimate the gradient, so that, having a policy π_θ, we can perform the following update:

$$\theta' = \theta + \alpha \nabla_\theta J$$

where J is a reward function to maximize.

A suited choice of the loss function J is the value function V, as maximizing it, we maximize the expected return from state S_t.

The *policy gradient theorem* tells us that, considering V as loss function to maximize:

$$\nabla J \propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla_\theta \pi(a|s,\theta)$$

However this does not give directly the answer to how to optimize the policy, since it requires the estimation of the stationary distribution *mu*, however from here we can derive other algorithms that will be explained in the following sections

3.8 REINFORCE

REINFORCE, also known as the Monte Carlo Policy Gradient, is a fundamental policy gradient algorithm in the field of reinforcement learning (RL). It offers a simple yet powerful approach for training agents to learn optimal policies in a wide range of environments.

At its core, REINFORCE utilizes Monte Carlo sampling to estimate the gradients of the policy parameters. The policy defines the agent's behavior, mapping states to actions. The goal of REINFORCE is to maximize the expected cumulative reward by iteratively updating the policy based on the observed rewards during interactions with the environment.

To train the policy, REINFORCE employs a sampling-based approach. The agent interacts with the environment, taking actions based on the current policy, and collects trajectories that consist of states, actions, and rewards. These trajectories are then used to estimate the expected return, which is the sum of the rewards obtained from a given state onward.

The key idea in REINFORCE is to compute the gradients of the policy parameters by using these trajectories and the associated returns. The gradients indicate how the policy parameters should be adjusted to increase or decrease the probability of selecting actions that lead to higher expected returns. The update is performed via stochastic gradient ascent, where the policy parameters are adjusted in the direction of the estimated gradients.

In order to achieve this, we need to further elaborate the policy gradient formula:

$$\begin{aligned}
\nabla J &\propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla_\theta \pi(a|s,\boldsymbol{\theta}) \\
&= \mathbb{E}_\pi \left[\sum_a q_\pi(S_t,a) \nabla \pi(a|S_t,\boldsymbol{\theta}) \right] \\
&= \mathbb{E}_\pi \left[\sum_a \pi(a \mid S_t,\boldsymbol{\theta}) \, q_\pi(S_t,a) \frac{\nabla_{\boldsymbol{\theta}} \pi(a \mid S_t,\boldsymbol{\theta})}{\pi(a \mid S_t,\boldsymbol{\theta})} \right] \\
&= \mathbb{E}_\pi \left[q_\pi(S_t,A_t) \frac{\nabla_{\boldsymbol{\theta}} \pi(A_t \mid S_t,\boldsymbol{\theta})}{\pi(A_t \mid S_t,\boldsymbol{\theta})} \right] \\
&= \mathbb{E}_\pi \left[G_t \frac{\nabla_{\boldsymbol{\theta}} \pi(A_t \mid S_t,\boldsymbol{\theta})}{\pi(A_t \mid S_t,\boldsymbol{\theta})} \right] \\
&= \mathbb{E}_\pi \left[G_t \ln \pi(A_t|S_t,\boldsymbol{\theta}) \right]
\end{aligned}$$

This last step gives us a way of updating our policy:

$$\theta' = \theta + \alpha G_t \nabla \ln(\pi(A_t|S_t,\theta))$$

Following is the entire algorithm:

Algorithm 5: REINFORCE Algorithm (On-policy)

```
Initialize policy $\pi$ with parameters $\theta$;
Initialize empty trajectory buffer $D$;
while not converged do
    Initialize empty trajectory $\tau$;
    Initialize the starting state $s_0$;
    while $s_t$ is not a terminal state do
        Choose action $a_t$ according to the policy $\pi(\cdot|s_t, \theta)$;
        Execute action $a_t$ in the environment and observe the next state $s_{t+1}$ and
          reward $r_{t+1}$;
        Append $(s_t, a_t, r_{t+1})$ to trajectory $\tau$;
        $s_t \leftarrow s_{t+1}$;
    end
    Compute policy gradient estimate:
        $\nabla_\theta J(\theta) \approx \sum_{t=0}^{|\tau|} \nabla_\theta \log \pi(a_t|s_t, \theta) \cdot \left( \sum_{t'=t}^{T-1} r_{t'+1} \right)$
    Update policy parameters:
        $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$
end
```

While REINFORCE is a popular and effective algorithm for policy gradient reinforcement learning, it does have some drawbacks that can impact its performance and efficiency:

∗ High Variance: The REINFORCE algorithm suffers from high variance in gradient estimates. The gradients are estimated using Monte Carlo sampling, which introduces inherent noise into the estimation process. This high variance can result in slow convergence and make the learning process unstable.

∗ Lack of Exploration-Exploitation Trade-off: REINFORCE does not explicitly address the exploration-exploitation trade-off. It relies on the policy's exploration behavior, which is often controlled by hyperparameters or other exploration strategies. The algorithm might struggle to effectively explore the state-action space and get stuck in suboptimal policies.

∗ Absence of Baseline: Without a baseline value, REINFORCE suffers from high variance in gradient estimates. The lack of a baseline makes it challenging to estimate the advantages of different actions accurately. This can lead to inefficient learning and slower convergence.

∗ No Value Function Approximation: REINFORCE directly optimizes the policy without utilizing a value function approximation. This means that value estimates are not used to guide or accelerate the learning process. Value function approximation can provide additional information and aid in faster and more stable convergence.

∗ Sensitivity to Learning Rate: REINFORCE can be sensitive to the choice of the learning rate. An overly high learning rate can lead to unstable training, while an excessively low learning rate can result in slow convergence. Finding an appropriate learning rate can be a challenging and time-consuming process.

3.9 REINFORCE with Baseline

The policy gradient theorem can be generalized to include a comparison of the action value to an arbitrary baseline $b(s)$, thus changing the update rule to:

$$\theta' = \theta + \alpha(G_t - b(S_t))\nabla \ln(\pi(A_t|S_t, \theta))$$

Algorithm 6: REINFORCE Algorithm with baseline

```
Initialize policy $\pi$ with parameters $\theta$;
Initialize value function $V$ with parameters $w$;
Initialize empty trajectory buffer $D$;
while not converged do
    Initialize empty trajectory $\tau$;
    Initialize the starting state $s_0$;
    while $s_t$ is not a terminal state do
        Choose action $a_t$ according to the policy $\pi(\cdot|s_t, \theta)$;
        Execute action $a_t$ in the environment and observe the next state $s_{t+1}$ and
          reward $r_{t+1}$;
        Append $(s_t, a_t, r_{t+1})$ to trajectory $\tau$;
        $s_t \leftarrow s_{t+1}$;
    end
    Compute policy gradient estimate:
        $$\nabla_\theta J(\theta) \approx \sum_{t=0}^{|\tau|} \nabla_\theta \log \pi(a_t|s_t, \theta) \cdot (G_t - V_w(S_t))$$
    Compute value gradient estimate:
        $$\nabla_w J(w) = \nabla_w \left( \sum_{t=0}^{|\tau|} (G_t - V_w(S_t))^2 \right)$$
    Update parameters:
        $$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$$
        $$w \leftarrow w - \alpha \nabla_w J(w)$$
end
```

This is valid until the baseline is either a constant or a function that does not depend on the action.
One natural choice for such baseline, is the value function V_π; in fact, since $V_\pi(s)$ is the expected return from state s under the policy π, $G_t - V_\pi(S_t)$ is a function that rewards actions with return above average, and discourage action below average.

3.10 Actor Critic

Actor-Critic[6] methods are a class of reinforcement learning algorithms that combine the advantages of both policy-based and value-based approaches. In Actor-Critic[6] methods, an agent consists of two components: an actor and a critic.

The actor component is responsible for learning a policy that maps states to actions. It directly interacts with the environment, selects actions based on the learned policy, and aims to maximize the expected cumulative reward. The actor typically uses a parameterized policy, such as a neural network, and updates its parameters to improve the policy's performance over time.

The critic component, on the other hand, evaluates the quality of the actor's actions and provides feedback. It estimates the expected cumulative reward or value function of a state or state-action pair. The critic uses a value function approximation method, such as a neural network, to estimate the values. The critic's estimation helps guide the actor towards better actions by providing feedback on the expected rewards.

The actor and critic components work in tandem. The actor takes actions based on its policy, and the critic evaluates the actor's actions and provides feedback in the form of value estimates. This feedback is then used to update the actor's policy parameters to improve its performance. The critic's value function estimation is updated based on temporal difference learning, using the difference between estimated and actual returns.

Algorithm 7: TD(0) Algorithm with baseline

```
Initialize policy $\pi$ with parameters $\theta$;
Initialize value function $V$ with parameters $w$;
Initialize empty trajectory buffer $D$;
while not converged do
    Initialize empty trajectory $\tau$;
    Initialize the starting state $s_0$;
    while $s_t$ is not a terminal state do
        Choose action $a_t$ according to the policy $\pi(\cdot|s_t, \theta)$;
        Execute action $a_t$ in the environment and observe the next state $s_{t+1}$ and
         reward $r_{t+1}$;
        Append $(s_t, a_t, r_{t+1})$ to trajectory $\tau$;
        $s_t \leftarrow s_{t+1}$;
        Compute policy gradient estimate:
            $\nabla_\theta J(\theta) \approx \nabla_\theta \log \pi(a_t|s_t, \theta) \cdot (R_t + \gamma V_w(S_{t+1}) - V_w(S_t))$
        Compute value gradient estimate:
            $\nabla_w J(w) = \nabla_w \left( (R_t + \gamma V_w(S_{t+1}) - V_w(S_t))^2 \right)$
        Update parameters:
            $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$
            $w \leftarrow w - \alpha \nabla_w J(w)$
    end
end
```

In order to accomplish all of this, it exploits the TD update [7], by redefining the concept of G_t in the REINFORCE with baseline algorithm with $R_t + \gamma V_\pi(S_{t+1})$.

While Actor-Critic methods offer several advantages, they also have some limitations and drawbacks, including:

* Variance in Actor Updates: Actor-Critic methods can suffer from high variance in the updates of the actor's policy parameters. The actor's policy is updated based on the critic's value estimates, which can introduce significant fluctuations and result in unstable learning.

* Bias in Critic Estimates: The critic's value function approximation introduces bias in the estimation of state or state-action values. Depending on the chosen function approximation method and network architecture, the critic's estimates may not accurately represent the true values, leading to suboptimal performance.

* Exploration-Exploitation Trade-off: Actor-Critic methods still face challenges in ef-

fectively balancing exploration and exploitation. While the actor's policy can exploit learned knowledge, exploration of new states and actions is crucial for discovering optimal policies. Designing effective exploration strategies remains an active area of research.

∗ Sensitivity to Hyper-parameters: The performance of Actor-Critic methods can be highly sensitive to the choice of hyper-parameters, including learning rates, discount factors, and exploration parameters. Selecting appropriate values for these hyper-parameters can be challenging and time-consuming, and suboptimal choices may hinder convergence or lead to poor performance.

∗ Non-stationary in the Learning Process: The interaction between the actor and the critic introduces a non-stationary in the learning process. As the actor updates its policy, the distribution of state-action pairs changes, which can make learning more challenging and result in slower convergence.

∗ Sample Efficiency: Actor-Critic methods may require a large number of samples to learn effective policies. Especially in complex environments, gathering sufficient data to train the actor and critic networks can be time-consuming and computationally expensive.

To reduce the variance of the gradient, the algorithm can be extended to do a batch update, by having multiple simulators running in parallel.

3.11 Trust Region Policy Optimization (TRPO)

The Trust Region Policy Optimization (TRPO)[8] algorithm is motivated by the need to address two key challenges in policy optimization: sample efficiency and policy improvement guarantees.

One of the main motivations behind TRPO[8] is to improve the sample efficiency of policy optimization methods. Traditional policy gradient algorithms often suffer from high sample complexity, requiring a large number of interactions with the environment to achieve good performance. TRPO[8] aims to mitigate this issue by providing a more efficient and principled approach to policy updates.

Another motivation of TRPO is to provide theoretical guarantees on policy improvement. Policy optimization algorithms aim to iteratively improve the policy by updating its parameters. However, in practice, updating the policy too aggressively can lead to instability or even performance degradation. TRPO addresses this by introducing a trust region constraint, which ensures that policy updates are limited to a region where the performance improvement is guaranteed.

By defining a constraint on the maximum policy divergence, TRPO prevents large policy updates that may cause instability or a significant drop in performance. Instead, it focuses on making incremental policy updates within a trust region that ensures a certain level of improvement. This allows for more stable and controlled policy optimization.

To achieve this, TRPO reuses the samples many times, until the current policy is in the neighborhood of the one used to sample the data. However, if for DQN Polyak averaging has been used to ensure this, it's not the optimal way: in fact, the relation between parameter space and policy space is not the same, thus small or big changes might have opposite consequences in the policy space.
Thus, TRPO uses the following update definition:

$$\theta_{\text{new}} = \theta + \alpha \cdot \arg\max_{\delta\theta} L_{\text{PG}}(\theta + \delta\theta)$$

subject to the constraint:

$$\mathrm{KL}\left[\pi_{\text{old}}(\cdot|s) \parallel \pi(\cdot|s;\theta)\right] \leq \delta$$

where:

$$L_{\text{PG}}(\theta) = \hat{\mathbb{E}}_t \left[\frac{\pi(\cdot|s_t;\theta)}{\pi_{\text{old}}(\cdot|s_t)} \cdot A^{\pi_{\text{old}}}(s_t, a_t) \right]$$

is the surrogate objective function, π_{old} is the policy with old parameters, $\pi(\cdot|s;\theta)$ is the policy with updated parameters, KL denotes the Kullback-Leibler divergence, δ is the trust region radius, α is the step size for the update, and $A^{\pi_{old}}$ is the advantage function of Actor Critic, defined as $r_t + \gamma V(s') - V(s)$.

One of the drawbacks of the Trust Region Policy Optimization (TRPO) algorithm is the challenging and computationally expensive estimation of the Kullback-Leibler (KL) divergence between the old and updated policies.
Estimating the KL divergence between two probability distributions, such as the old and updated policies, typically requires either sampling or numerical integration methods. Both approaches can be computationally expensive and may not scale well to complex environments with large state and action spaces.
Estimating the KL divergence accurately is crucial because an incorrect estimation can lead to overly conservative or overly aggressive policy updates. If the KL divergence is underestimated, the policy update may not explore new and potentially better policies adequately. On the other hand, if the KL divergence is overestimated, the policy updates may be unnecessarily conservative, hindering the learning process.

3.12 Proximal Policy Optimization (PPO)

Proximal Policy Optimization (PPO)[9] is an improvement over Trust Region Policy Optimization (TRPO)[8] that addresses some of its limitations and offers several advantages. Here are a few key improvements of PPO over TRPO:

* Simplified Trust Region: TRPO uses a strict KL divergence constraint to limit policy updates. However, estimating and enforcing the KL constraint accurately can be challenging and computationally expensive. PPO simplifies this by introducing a clipped surrogate objective function that directly constrains the policy update within a pre-defined threshold. This simplification makes PPO easier to implement and more computationally efficient.

* Better Sample Efficiency: PPO tends to achieve better sample efficiency compared to TRPO. It utilizes the collected data more effectively by reusing samples across multiple iterations and updating the policy multiple times per iteration. This enables PPO to learn from the data more efficiently and achieve good performance with fewer interactions with the environment.

* Improved Stability: PPO addresses the issue of high variance in policy updates that can arise in TRPO. The clipped surrogate objective function in PPO bounds the policy update, preventing large policy changes and reducing the risk of destabilizing the learning process. This increased stability makes PPO easier to tune and less sensitive to hyper-parameters.

* Flexibility in Step Sizes: TRPO relies on a line search to determine the step size for policy updates, which can be computationally expensive. PPO, on the other hand, allows for simpler and more flexible choices of step sizes. It uses a fixed step size or adaptive methods such as adaptive KL penalty coefficients, which simplifies the implementation and reduces computational overhead.

Simplicity and Ease of Use: PPO's algorithmic simplicity and ease of use are additional advantages over TRPO. PPO has fewer hyper-parameters to tune and requires fewer implementation details, making it more accessible for practitioners and researchers. This simplicity facilitates faster experimentation and adaptation of the algorithm to various reinforcement learning problems.

In order to achieve this, it offers 2 reformulations of the problem. The first one being a relaxation of the TRPO update, with an adaptive parameter:

$$L^{PPO}(\theta) = \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} \cdot A_t - \lambda_t \mathbb{E}_{a\sim\pi_\theta(\cdot|s_t)} \left[\text{KL}\left(\pi_\theta(\cdot|s_t)||\pi_{\theta_{\text{old}}}(\cdot|s_t)\right)\right]\right]$$

However, the second proposed reformulation is the one that is more vastly used, thanks to its simplicity and it's effectiveness:

$$L^{PPO}(\theta) = \hat{\mathbb{E}}_t \left[\min\left(r_t(\theta) \cdot A_t, \text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right) \cdot A_t\right)\right]$$

where:

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$$

is the probability ratio between the current policy π_θ and the old policy $\pi_{\theta_{\text{old}}}$, A_t is the advantage function at time step t, and ϵ is a hyper-parameter that determines the range of the clipping.

After PPO, further methods are being proposed in the literature, however never received as much attention as PPO, because they would usually introduce many hyper-parameters, with no prior good choice, as instead PPO does. Thus, the newest proposed methods usually tend to overcome other issues introduced by the large scale of the projects, thus dealing with asynchronous update or federated learning.
PPO it's still probably the most widely used even nowadays and the go-to in most of the scenarios, often used as benchmarks and in real world problems, most notably, it's been used to fine-tune GPT-3 giving birth to ChatGPT.

For completeness, the following is the PPO algorithm:

Algorithm 8: Proximal Policy Optimization (PPO) with Clipping

```
Input: Initial policy parameters $\theta_0$, initial value function parameters $\phi_0$
Initialize: Step sizes $\alpha_{\text{policy}}$, $\alpha_{\text{value}}$, clipping threshold $\epsilon$
for iteration = $1, 2, \ldots$ do
    Collect a set of trajectories using the current policy $\pi_\theta$
    Compute advantages $A_t$ for each time step $t$
    Optimize the value function by minimizing the mean squared error:
        $$\phi_{\text{new}} = \arg\min_{\phi} \frac{1}{N} \sum_{t=1}^{N} \left(V_\phi(s_t) - R_t\right)^2$$
    for mini-batch = $1, 2, \ldots, num\_mini\_batches$ do
        Compute probability ratio $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$ for each time step $t$
        Compute the clipped surrogate objective:
            $$L^{PPO}(\theta) = \hat{\mathbb{E}}_t \left[\min\left(r_t(\theta) \cdot A_t, \text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right) \cdot A_t\right)\right]$$
        Update the policy parameters using gradient ascent:
            $$\theta_{\text{new}} = \theta_{\text{old}} + \alpha_{\text{policy}} \cdot \nabla_\theta L^{PPO}(\theta)$$
        Update the value function parameters using gradient descent:
            $$\phi_{\text{new}} = \phi_{\text{old}} - \alpha_{\text{value}} \cdot \nabla_\phi \left(\frac{1}{N} \sum_{t=1}^{N} \left(V_\phi(s_t) - R_t\right)^2\right)$$
    end
    Update the old policy parameters: $\theta_{\text{old}} \leftarrow \theta_{\text{new}}$
end
```

Chapter 4

Game Theory

Game Theory is a rigorous mathematical framework used to analyze strategic interactions and decision-making in various contexts. It provides a formal structure to understand how rational agents make choices when their outcomes are influenced by the actions of others.

It's highly relevant for this project as it captures the multi-agent nature of the game, and moreover, the setting called 2 player 0 sum (2p0s) games have been extensively studied.

4.1 Main components

Game Theory is based on few building block, on which everything else is built on top.

Utility function

The utility function is a function which is used to describe the desirability of a certain outcome $u(q)$. If q is a countable good, it's assumed that it's directly proportional but sub-linearly increasing with respect to q ($u'(q) \geq 0$ and $u''(q) \leq 0$), the absolute value attributed to u usually does not count, as it's used to describe a preference ($u(q) \geq u(q')$).

Player

Players in game theory are whom potentially might act in such game. The only assumption, which is the fundamental assumption that holds any other theorem built in this branch, is that players are rational.

By rational, Game Theorist mean that a player will act for his own good, thus will pick q over q' if and only if $u(q) \geq u(q')$.

In case multiple subsequent choices have to be made, a discount factor $\gamma \in [0, 1]$ can be introduce to express the idea that present has more importance over the future.

4.2 Static game of complete information

In this form of the game, each player i simultaneously and independently chooses an action from its own set of available actions, and the joint action $(a_1, ..., a_n)$ determines the outcome of the game.

In addition, as assumption to this, it's assumed that the payoff of each player, the available actions and rationality of the agents are common knowledge, and also it's assumed that everybody knows that everybody knows. This implies that everybody knows that everybody is trying to maximize their own payoff.

In case of 2-player games, a bi-matrix is used to represent the payoffs.

		Player Y	
		A	B
Player X	L	$u_1(A, L), u_2(L, A)$	$u_1(B, L), u_2(L, B)$
	R	$u_1(A, R), u_2(R, A)$	$u_1(B, R), u_2(R, B)$

A joint strategy is considered Pareto dominated by a second joint strategy if the latter if equally good for all the players, but strictly better for at least one player with respect to the former. If a joint strategy has no other strategy that dominates it, it's said to be Pareto efficient.

Instead, a Best Response (BR), is a strategy that best respond to its opponent strategy:

$$u_i(s_i, s_{-i}) \geq u_i(s'_i, s_{-i}),\ \forall s'_i \in S_i$$

Given such definition, it can be said that a rational player who believes that the opponents are playing some $s_{-i} \in S_{-i}$, will always choose a best response to s_{-i}

4.3 Nash Equilibrium

Game Theory aims to predict what a rational player will play. A prediction is correct if the players are willing to play their predicted strategy. Given the previous definition of BR, a strategy that a player wants to play has to be a BR to the predicted strategy of others:

$$u(s^*_i, s^*_{-i}) \geq u(s_i, s^*_{-i}),\ \forall s_i \in S_i$$

If such joint action exists, it's a Nash Equilibrium (NE), which means that no player has an incentive to deviate.

4.4 Mixed Strategies

Until now only pure strategy have been considered, which means that a player deterministically chooses an action. However, this restrictions doesn't allow to guarantee to have a Nash Equilibrium.

A mixed strategy, differently from a pure strategy, it's a belief over the actions, which can be formalized as a distribution over the actions.

$$p : A \to [0, 1],\ \sum_{a \in A} p(a) = 1$$

Given such strategy, the payoff/utility is redefined as follows:

$$u_i(m_1, ..., m_n) = \sum_{s \in S} \prod_j m_j(s_j) \cdot u_i(s)$$

Instead the definition of Nash Equilibrium can be extended naturally:

$$u(m^*_i, m^*_{-i}) \geq u(m_i, m^*_{-i}),\ \forall m_i \in \Delta S_i$$

However, differently from earlier, the Nash Theorem proves that every game with finite S_i's has at least one Nash Equilibrium, possibly involving mixed strategies.

4.5 Dynamic Games

Static games of complete information are very limiting, as they model a one-shot game where actors play together. Dynamic Games are introduced to relax this playing together constraint, by allowing a player to move first, and a second player to move later, knowing what the first have chosen.

To represent this structure, instead of a bi-matrix, it's used a decision tree for the graphical representation.

4.6 Minimax & Maximin and Zero Sum Games

Minimax and Maximin are two approaches used to solve perfect information 2-player games, introduced specifically for 0-sum games, which are characterized by the fact that $u_i(s) = u_{-i}(s)$.

Maximin picks the action that maximizes the minimum reward (worst payoff), instead minimax picks the action that minimized the maximum possible loss.

$$\text{Minimax} = \min_{s_1}\left(\max_{s_2} U_1(s_1, s_2)\right)$$
$$\text{Maximin} = \max_{s_2}\left(\min_{s_1} U_2(s_1, s_2)\right)$$

More intuitively, in case of zero sum games, can be though of taking the safest option, which is the one that minimizes the maximum possible loss: such point, if exists, it's a saddle point.

4.7 Markov Games & Stochastic Games

Stochastic Games, also known are Markov Games, are the common territory between the RL formulation of the problem we are trying to solve, and the Game Theory formulation of it. A Stochastic Game is defined by:

* S, the set of all possible states of the RL formulation (dual of game G for GT formulation)
* a, the set of all possible actions of the RL formulation (dual of the action s for GT)
* π, the set of all possible policy of the RL formulation (dual of strategy m for the GT formulation)
* R, the set of rewards of the RL formulation (dual of utility u for the GT formulation)
* $P(s'|s, a_1, a_2)$, the transition probability between states/games
* γ a discount factor

For such class of problems, the best response of a player i to a second player $-i$, is defined as follows:

$$\pi^{i*} \in Br(\pi^{-i}) := \left\{ \underset{\hat{\pi} \in \Delta\mathcal{A}}{argmax}\ \mathbb{E}_{\hat{\pi}^i, \pi^{-i}} \left[R^i(a^i, a^{-i}) \right] \right\}$$

For a Stochastic Game, a stronger equilibrium is usually used instead of the NE, called the Markov perfect NE:

$$V^{\pi^{i,*}, \pi^{-i,*}}(s) \geq V^{\pi^i, \pi^{-i,*}}(s), \quad \forall s \in \mathbb{S}, \forall \pi^i \in \Pi^i, \forall i \in \{1, \ldots, N\}$$

In order to learn such policy, in the 2p0s perfect information games, the minimax theorem states that that NE exists, and can be achieved minimizing the maximum loss, thus RL is guaranteed to converge.

Chapter 5

Problem Formalization

In this chapter we will formalize the two-player Briscola game as a Reinforcement Learning problem. In order to do so, we will formalize the state, action and reward function, since they heavily impact the end result, thus making their definition very important.

5.1 State

The state of a Reinforcement Learning problem defines the set of information that the agent have access to. If the state gives every information the agent needs to decide it's action, we say that the problem is fully observable (MDP), satisfying the Markov property and thus allowing the problem formalization to be sound: an example of this is the board of a chess game, which is a perfect information game, thus the current state fully determines the information set needed by the agent to learn the optimal action.
Instead, if the state does not give all the information that the agent needs to evaluate the action, then the problem is called partially observable (POMDP), making it more difficult, as usually the history of how an agent ended up in that state, is also relevant for the decision making part of the problem. This is the case of Briscola if faced in the naive way, as the set of information available to an agent at a specific time, it's not enough, and the history of the previous game-steps should be also considered, making it a partial information game.
However, we can assume that the order of which the previous steps have been made, is not relevant, thus transforming the history from a time-dependent series, to just a set.
Exploiting this fact we can approximately transform the POMDP to a MDP, thus making the training much more efficient and effective.

The state will then be composed by multiple aggregated information which can be listed as follows:

1. its own cards: 40×2 flattened tensor where the first element of each entry represent if the player has that card in his hand, and the the second entry if that card is a briscola
2. the cards on the table: 40×2 flattened tensor where the first element of each entry represent if the card is on the table, and the the second entry if that card is a briscola
3. the briscola of the game: 4 dimensional one-hot vector representing all possible suits
4. the player score: a single entry with the score
5. the turn counter: a single entry with the counter
6. history of player cards: 40×2 flattened tensor where each entry represent if that card has already been played, and if that card is a briscola or not

In total, the state for both actor and critic is composed by 243 values.

However, for the actual implementation, an ablation study of what was relevant for the state, and which state formalization led to the best performance, has been performed, however no statistical significance has been observed between them, given enough training time.

5.2 Actions

The action space of the actor is discrete, thus can be represented by a multinomial distribution, describing each entry the probability of playing that card.
Two different options have been explored:

1. actions in $\{1, 2, 3\}$, where the first entry describes the probability to play the first card encoded in the state space in the player hand, and so on

2. actions in $\{0, ..., 39\}$, where each entry specifies a specific card

The second option gives much better results, thus it's the one that has been chosen. However, not all cards are available at each time-step to the player, thus a masking-and-re-normalization approach has been taken:

$$m = \{(0 \vee 1)^{40} : 0 \text{ if player doesn't have card at time } t, 1 \text{ otherwise}\}$$
$$\tilde{\pi} = \phi(s)$$
$$\pi_i = \frac{\tilde{\pi}_i \cdot m_i}{\sum \tilde{\pi}_j \cdot m_j}$$

5.3 Reward function

Depending on the reward function formalization, the agent will learn to prefer some moves over others, in order to maximize such formalization.

The naive approach would be to give the agent a reward of $+1$ for winning and -1 for losing. However, such sparse reward would lead to a very slow learning, as the agent has to resort to trial and error, and learning by exclusion, as it has no clue which moves allowed him to win.

To densify the rewards, we could resort to a binary reward for each turn, however the agent would learn to maximize the number of turn won, which is not optimal if we want to maximize the win-rate.

In order to solve those tow problems, by exploiting the properties of the game of Briscola, we can give the agent a reward proportional to the number of points gained in that turn, and ask the agent to maximize such reward function.

In addition to this, another reward function that gives an additional bonus for winning has been tested, however no statistical significance has been observed.

As presented the optimization goal of a RL problem is the following:

$$\max \mathbb{E}\left[\sum_{r} \gamma^t r_t\right]$$

Such formalization introduces an hyper-parameter, γ, which is mainly needed in the case of infinite horizon MDPs, in order not to deal with infinite rewards. In our case, such hyper-parameter would make the agent more greedy, thus no values of gamma have been tested outside of 1 since it would make little sense.

5.4 Model Architecture

RL naively would use a discrete set of state and a discrete set of actions in it's formalization. Thus, a RL problem can be solved holding in memory a table of size $|S| \times |A|$, where each entry is the Q estimate of that s, a pair. While learning such table, one can either pick a policy, or learn one in the same way. Indeed, RL is just used to estimate a gradient:

$$Q(s^t, a^t)_{new} = Q(s^t, a^t) - \alpha\nabla(Q(s^t, a^t) - G^t)^2$$
$$\pi(a^t|s^t)_{new} = \pi(a^t|s^t) + G^t\nabla \ln \pi(a^t|s^t)$$

Thus, it doesn't need anything more than a matrix in the discrete case. In the continuous case, it's needed to resort to approximations, assuming an underlying distribution.

However, such formalization is not scalable. Indeed, we need multiple samples to learn each entry of such table. This can be feasible, and even done, with low dimensional problems, as it's theoretically sound, stable and fast. This is not true anymore with large scale problem, where the state space of the underlying MDP can be huge, if not continuous, and relying on approximators as mixture of Gaussians, is no more an option.

In order to overcome this, RL relies on ML, mostly for function approximators. In the early days of RL, such function approximators would receive some handcrafted features or heuristics, in order to learn a mapping from state to Q values, for example. Some examples on how those features might be built are:

* Coarse Coding: overlapping circles are distributed over some state space representation, and if the current state is inside a circle, the corresponding feature is set to 1, otherwise 0
* Tile coding: the state space is discretized and multiple shifted grids are laid on top, and then the same reasoning of Coarse Coding is used
* Radial basis functions: an "inverse"-distance measure is used to estimate the closeness to some hotspot (centers of the Gaussians, for example)

Once such formalization/encoding $x(s, a)$ is defined, methods such as linear regression are used to learn Q values:

$$Q_w(s, a) = w^T x(s, a)$$
$$\Delta w = (G_t - Q_w(s_t, a_t))x(s_t, a_t)$$

However, even such methods fail in very high dimensional settings, either because they relies on assumptions, for example if the relations between features and Q values is non-linear then linear regression fails, or because the generalization is not good enough.

To overcome this problem, Deep Learning techniques are applied. Indeed, in recent history, most of the superhuman RL agents under the hood use some neural network to get better generalization.
However, combining RL and DL is not as straightforward as it might seem: even if RL is agnostic to the type of function approximators used, it has a big challenge by the nature of its problem setting, which is the non stationarity.
The non stationarity in RL is a big problem, and it refers from the fact that the data distribution that the network is trained on is not coming from a stationary distribution. In fact, as the agents learns, it changes the state distribution of the MDPs, since the transition matrix is also defined by the policy of the agent.

The main problem with it, it's the fact that neural network need the assumptions of stationarity to converge to a good solution, and moreover, it's been shown that they are actually bad at handling non stationarities, due to a phenomenon called catastrophic forgetting, problem that gave birth to a whole new research area called Continual Learning, which indeed tries to mitigate this non stationarity problem.

Even if the first trials of combining DL and RL needed special tweaks to make it work, as for example DQN required the target policy not to change too fast, nowadays this friction has been alleviated.

The two main reasons nowadays interactions between DL and RL work much easier are the following:

∗ Reinforcement learning advancements: lately many new learning approaches have been proposed, that tend to reduce the variance of the gradient estimate, as a trade-off with bias, which have been shown to improve significantly the training

∗ Optimizers: indeed, RL is an optimization process, and gradient based optimizers plays an important role in the process. Adaptive gradient and momentum play a significant role in reducing the variance of the gradient, and dealing with ill-conditioned losses, leading to much better results in RL

∗ Deep Learning Advances: the most important one is for sure the introduction of Batch Normalization [10], but also any other normalization techniques proposed such as LayerNorm[11], InstanceNorm[12] etc. Batch Norm reduces the internal covariate shift problem, which also alleviates the non stationarity of the problem thanks to the on-the-fly-estimate of its μ and σ parameters. Depending on the problem also newly proposed pre-training techniques can be highly effective, leading to the rise of the firsts RL foundational models.

For the Briscola project, as shown in the previous chapter, we will use a one hot encoded high dimensional vector, thus there is little inductive-bias that we can encode in the network. For this reason the architecture used for the project it's just a FFNN.

In case of Q-networks, the following architecture has been used

∗ Affine transformation with output size 128

∗ Batch Norm

∗ $tanh$ non linearity

∗ Affine transformation with output size 128

∗ Batch Norm

∗ $tanh$ non linearity

∗ Affine transformation with output size 40, initialized with weights sampled from $N(0, 0.01)$

In case of V-networks, the following architecture has been used

∗ Affine transformation with output size 128

∗ Batch Norm

∗ $tanh$ non linearity

∗ Affine transformation with output size 128

∗ Batch Norm

∗ $tanh$ non linearity

∗ Affine transformation with output size 1, initialized with weights sampled from $N(0, 0.01)$

In case of π-networks, the following architecture has been used

* Affine transformation with output size 128
* Batch Norm
* *tanh* non linearity
* Affine transformation with output size 128
* Batch Norm
* *tanh* non linearity
* Affine transformation with output size 40, initialized with weights sampled from $N(0, 0.005)$
* $softmax$ non linearity

In particular, the initialization have been shown to be very important for an effective training of the RL agents, mainly in π networks, as they can lead to biases or the need to much more iterations due to bad initializations, thus in case of policies, a very low σ has been used to guarantee that at the beginning of training the policy distribution is approximately uniform.
In addition to this *tanh* has been used as activation function as it has been shown [13] that for relatively small network, is much more effective in RL, mainly thanks to the smoothness that it offers over the ReLU activation function family in the loss landscape, as ReLU has to build the function piece-wise with hyperplanes, where *tanh* uses smoother curves.
Finally, Batch Norm has been highly used thanks to its ability to alleviate convergence and to work greatly with the latest optimizers.

For the optimization, Adam has been used for all three types of networks, as it has been empirically shown[13] to be the most effective one both in general DL but also in RL settings.

Regarding the loss functions to minimize, for the policy it's going to be used the one of the respective method, however regarding the value functions V and Q, it has been shown[13] that even though Huber loss should be better for it's ability to deal with outliers and high variance, it's not true for RL, thus Mean Squared Error it's going to be the regression loss for such functions.

Even though regularizations play an important role in DL, no evidence of such importance has been seen in RL [13], thus no regularization techniques has been applied for the various models. In addition to the usual regularization, in RL there are some additional ones called Entropy regularization, that are use to regularize the policy more than the weights, however even though they are theoretically sound, and the intuition behind them seems correct, no evidence of their efficacy has been seen in large scale scenarios[13].

5.5 Evaluation

As for any other ML task, an evaluation or benchmark is needed to assess the quality of the product of the training.
However, Reinforcement Learning is notorious for being far from the other ML tasks. Usually in ML there would be some type of data considered to be the dataset, which would then be divided 3 part, training validation and testing, in order to assess the generalization of the model via some metric, it being accuracy, recall, MSE, F1, IoU, Perplexity and others, depending on the field we are working on.

RL instead have no notion of Dataset, apart few exceptions like successors features tasks and off policy learning, thus the evaluation it's always been a non-trivial task.

In cases like AlphaGo[2], humans and other AI have been compared using the Elo rating system, which establishes a relative order inside a pool of players.
Given R_A the score of player A and R_B the score of player B, the expected score of the game is the following:

$$E_i = \frac{1}{1 + 10^{(R_i - R_{-i})/400}}$$

At the end of the game (given S_i the score for player i), a player score is updated with the following formula:

$$R'_i = R_i + K \cdot (S_i - E_i)$$

Such rating system has some drawbacks in the real word, as players can stop playing to keep their score, as there is no dependence on time in any formula, and also a player can cherry-pick the opponent in order to maximize his score. However, in case of AIs, such drawbacks cannot be exploited, thus it's a valid metric.

On the other hand, this system requires in the first place a pool of players, and the resulting scores are as good as such players are. Indeed, if a pool of "average" players is used for the evaluation of an agent, the resulting AI will have a easy time getting a high score, where instead with a pool of experts, it's not the case. Nonetheless, the main problem is that such system requires other players.
In our case, as Briscola is well known in few states in the word, we found no open-source agents to compare our one with, thus Elo cannot be employed to evaluate performances.
Instead, we used 2 baselines:

* Random Agent: random agents are unbiased agents, and being able to beat 100% of the times a random agents, ensures optimality, since such agent has coverage over all possible other policies, thus in expectation will play every possible policy. The problem with this baseline, is the noise in the signal that it gives. If on one hand a random player will in expectation play every possible trajectory, most of them will be highly suboptimal, thus an AI should pretty quickly learn to defeat such agent.

* Rule based agent: we managed to find an open source rule based agent on Github, which uses handcrafted heuristics to evaluate the board and pick the best action. However, as for all handcrafted agents, it's hard to make a perfect agent, since the space of possible games it's too large, but also it will be at most as good as the person that engineered it.

By using those 2 agents, we will have a decent baseline to compare agents trained with different hyperparameters or algorithms. However, it has to be kept in consideration that it's not strictly correct to assume that an agent that has a higher win-rate against those agents is better than another agent with a lower win-rate.

As explained in the Game Theory section, the concept of Nash Equilibria implies that the resulting policy will not loose in expectation in 2p0s games. Thus, a perfect agent might perfectly have a 50% win-rate against both of them.

Chapter 6

Training

In this chapter we will present the details about the training procedures of the agent.
In particular, to fully comprehend the nature of the problem, the ability of such algorithms to cope with the problem, and the issues from those, we decided to face the training in multiple steps:

1. Training against rule-based agent
2. Training against itself
3. Training against a set of agents

6.1 Against rule-based

This scenario wanted to be a litmus test/sanity check, as there was no reason for the setting not to work, so it wanted to be a benchmark to check how hard the task actually is. In fact, by fixing the enemy, the learning becomes a "naive" partially observable MDP, with a stationary environment, formalization well known in the RL community, which convergence it's guaranteed under mild assumptions.

In order to accomplish this, we used an already-existing rule-based agent, which tries to estimate which was the best move via handcrafted rules/heuristics.
In addition to it, also a random agent has been used for the evaluation, in order to check the training history even while no progress against the rule-based was made.

The result of such training is know in Game Theory is known as a best-response, in the sense that the resulting agent would have learnt a specific policy which is the best response to that specific adversary. However, this would cause little to no generalization, so the resulting agent is far from being what we are aiming to, and for this reason this should only be considered as a step towards that end goal.

Indeed, it's well known in Game Theory, that any mixed strategy m has a best response that is a pure strategy s, since a mixed BR at most can be a linear combination of some pure strategies, thus yielding at most as the best of those pure strategy.
Thus, this approach is not even effective in the case where the adversary, in our case the rule based agent, is the actual optimal one. In fact, even if it was the optimal agent, our AI would learn a BR to that, which is not guaranteed to be a Nash Equilibria, so a BR of the optimal agent, might be 100% exploitable.

For example, in the game of Rock-Paper-Scissor, the NE is playing each action with a belief of 1/3. However, a BR for such agent, can be a pure strategy s which always plays Rock (or one of the other options), and the expected utility would be equivalent. However, clearly there is a BR to that BR which can beat it 100% of the times.

However, it can be said that in a 2-player-zero-sum game, any possible strategy can be defeated half of the times. If that condition is not met, thus the learned agent has a win-rate lower than 50% (ignoring the ties), it means that it has still room for improvements, but if it is not met, and the learned one surpasses significantly the 50% win-rate threshold, it means that the rule based agent is not optimal.

In all the cases, we have little to no guarantee about the generalization of the learned agent, trained against the rule based.

6.1.1 Deep Q Network

A neural network has been trained using DQN[4]. However, by itself the method does not provide a policy out-of-the-box, thus one has to handpicked. Some options are:

* ϵ-greedy policy: with probability $1 - epsilon$ we sample the greedy action (the one with highest expected returns), and with probability ϵ a suboptimal one.
* Boltzmann exploration [14], thus selecting an action proportionally to its expected return $p(a|s) \propto exp(Q(s,a)/T)$, where T is a temperature that is used to have more deterministic policies with time

For the training, the following hyper-parameters have been used:

Q-learning	
Batch-size	512
Stepsize	$1e-3$
Learning steps	1
Replay-mem size	100.000
Policy	ϵ-greedy
Training steps	10.000
Evaluate every	500
Evaluate with n. games	500

Table 6.1: Hyperparameters for DQN against Rule-based agent.

The agent has then been trained with the ϵ-greedy policy with a linear decay of ϵ for 10.000 steps, and the following are the win-rates of the agent against the rule-based agent and the random agent:

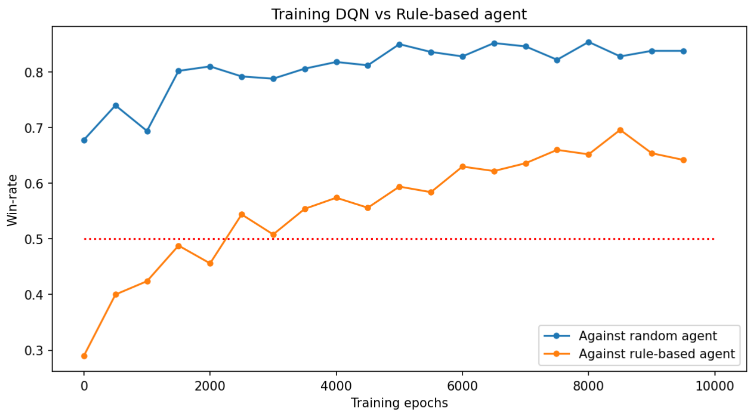

Figure 6.1: Training of DQN against Rule-based agent

Clearly, since the rule-based agent has been created with good heuristics, even though the agent is trained to learn a best response of the other agent policy, it's leading to some generalization, as showed by the random agent. However, it's needed to be kept in mind that the random agent, even though eventually will play all possible trajectories, most of them are highly suboptimal.

6.1.2 PPO

A neural network has been trained using PPO [9]. Thanks to the underlying actor-critic structure, this approach aims to learn a policy in addition to a value function, thus there is no need to handpick a policy. However, the policy learning is much slower as it relies on reinforces, thus needs more steps.

For the training, the following hyper-parameters have been used:

Q-learning	
Batch-size	512
Stepsize	$1e-3$
Learning steps	1
Clipping ϵ	0.1
Training steps	10.000
Evaluate every	500
Evaluate with n. games	500

Table 6.2: Hyperparameters for PPO against Rule-based agent.

The actor-critic agent has then been trained with Proximal Policy Optimization for 10.000 steps, and the following are the win-rates of the agent against the rule-based agent and the random agent:

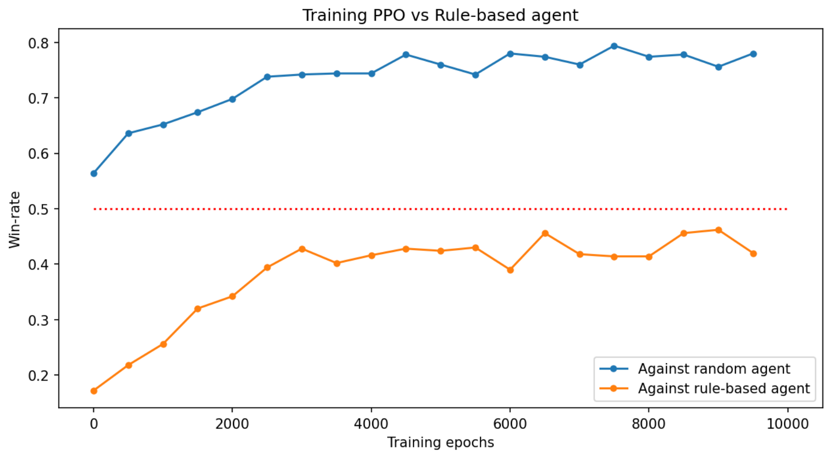

Figure 6.2: Training of PPO against Rule-based agent

As it's clear from the plots, the Q-learning agent learns much faster. However, such result was possible only thanks to the fact that the enemy was fixed throughout the whole training, however such agent cannot be used in the more advanced settings.
In addition to this, the results are not meant to be a comparison of the learning algorithms, as they are heavily impacted by their hyper-parameters, thus one could be shown to outperform

the other just by worsening the parameters of the latter.
Furthermore, as previously said, these are just the first 10.000 training steps, which is far from the training that the final agent will receive, thus should definitely be taken only as a sanity check for the RL formalization of the problem.

6.1.3 Other variants

Many other variants have been proposed in the literature, and have been coded and tested in this setting in order to obtain an overview of which are more interesting for this setting. In fact, many of them work better than other in low-variance rewards, other better in self-play environments, other are not much effected by bias induced by the temporal difference etc etc.

In particular, the following options have been tested:

* Generalized Advantage Function[15]: proposed many years ago, it aims at finding a trade-off between variance and bias in the temporal difference settings. It relies in a λ-exponentially-decaying weighted convolution of the future temporal difference errors to estimate the return at time t. If $\lambda \approx 1$ we get REINFORCE, if $\lambda \approx 0$ we get TD(0). However, this approach worked only in settings with $\lambda \approx 0$ showing that the game itself is effected by very high variance, and since this method has a non-ignorable computational cost, we discarded it and we used the naive TD(0)

$$\hat{A}_t^{GAE(\lambda,\gamma)} = \sum_{l=0}^{\infty}(\gamma\lambda)^l \delta_{l+t}^V$$
$$\delta_{l+t}^V = r_t + \gamma V(s_{t+1}) - V(s_t)$$

* Symmetric-Clipping: PPO has a pessimistic point of view when learning, in the sense that it learns much more with negative reward than with positive one, in order to avoid encountering again states where it has seen negative rewards, but will not overfit on positive rewards. This method showed some improvements on the learning, thus will be tested in future sections.

$$L^{\cancel{PPO}Clip}(\theta) = \hat{\mathbb{E}}_t\left[\cancel{\min(r_t(\theta)\cdot A_t,}\,\text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\cdot A_t\cancel{)}\right]$$
$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$$

* Recurrent PPO: naive ppo learns a policy $\pi(a|s)$, however as said before briscola is a partially observable environment, which can be solved using what's known in the literature as a "belief", that in our case would be a memory holding information about the past, for example an RNN. However, this method is not computationally cheap, as it is by itself more costly, but also needs much more data to learn, with little improvement over naive PPO, thus has been discarded.

$$\pi(a|s_t) \Rightarrow \pi(a|o_t, b) \Rightarrow \pi(a|o_t, \{o_{t-1}, o_{t-2}, ...\})$$
$$V(s_t) \Rightarrow V(o_t, b) \Rightarrow V(o_t, \{o_{t-1}, o_{t-2}, ...\})$$

* Action dependent baseline[16]: even though the policy gradient theorem admits a baseline as long as it only depends on the state, in the literature it has been proposed to use an action dependent baseline, the Q function, in order to reduce the variance. However, no significant improvement has been observed, and Q function training it's

trickier than V function training, thus this method has been dropped.

$$\begin{aligned}
\nabla_w L &\approx \mathbb{E}\left[\sum R_t \nabla_w \ln \pi_w(a_t|s_t)\right] \\
&\approx \mathbb{E}\left[\sum Q(s_t,a_t) \nabla_w \ln \pi_w(a_t|s_t)\right] \\
&\approx \mathbb{E}\left[\sum (Q(s_t,a_t) - V(s_t)) \nabla_w \ln \pi_w(a_t|s_t)\right] \\
&\approx \mathbb{E}\left[\sum (Q(s_t,a_t) - \mathbb{E}_a[Q(s_t,a)]) \nabla_w \ln \pi_w(a_t|s_t)\right] \\
&\approx \mathbb{E}\left[\sum (Q(s_t,a_t) - \sum_a \pi_w(a,s_t) Q(s_t,a)]) \nabla_w \ln \pi_w(a_t|s_t)\right]
\end{aligned}$$

6.2 Self-play

Self-play is a technique well known in the Game Theory in order to learn optimal policies in fully observable 2-players zero sum games. In consist on an agent that plays against himself, and relying on the minimax approach, it learns the optimal policy. The minimax algorithm is an algorithm for small extended games that has been shown to be optimal, and it relies on the idea of minimizing the maximum loss. Indeed, assuming that the enemy is playing optimally, than it's going to pick the action that maximize its utility, thus maximize your loss.

By playing against himself, the agent plays always more challenging trajectories and will asymptotically learn the best policy. This is theoretically sound also for repeated perfect information 2-player zero sum games, called Markov Games, and has been shown to be effective also when dealing with function approximators, for example in the case of AlphaGO[2] and AlphaZero[17].

The fact that DQN was able to win more than 50% of the games against the Rule-based agent, shows that that policy is far from optimal.

Self-play on the other hand forces the player to improve against itself, thus learning the best response against itself, which, even though won't learn to defeat all possible players, will learn an optimal policy, which by definition minimizes the maximum loss (potentially even always tying).

This definition of optimality, as said, does not guarantee to win, however, it guarantees not to lose in expectation, no matter which is the opponent we are facing.
However, human beings hardly are able to approximate a Nash-equilibrium in their policy, thus not playing optimally, which will indeed lead to possibly some wins for the self-play-agent.

Algorithm 9: Self-Play Algorithm

```
Input: Initial policy $\pi_{\text{init}}$, number of iterations $N$
Output: The final trained policy $\pi_{\text{current}}$
Initialize $\pi_{\text{current}} \leftarrow \pi_{\text{init}}$;
for $i = 1$ to $N$ do
    Collect data by playing games using $\pi_{\text{current}}$;
    Train a new policy $\pi_{\text{new}}$ using the collected data;
    Update $\pi_{\text{current}} \leftarrow \pi_{\text{new}}$;
end
```

Self-play can also be seen as a sort of Curriculum Learning for a RL agent. In fact, if we had a binary reward function, at the beginning the agent would have a hard time learning since all rewards will be losses, thus no signal has been given to the agent to learn. Instead, self-play will gradually learn with more challenging game-trajectories, always giving some, possibly noisy, signal to the agent.

However, all of this reasoning doesn't hold in the case of imperfect information games, such as Briscola. In fact, in this case the optimal strategy is not deterministic (there is no notion of optimal move), instead it is a distribution over all possible actions. For example for the game of rock-paper-scissors with uniform reward, the optimal strategy is the uniform policy over actions.

It's been shown that RL in this setting has a hard time converging to the Nash Equilibria, mainly when dealing with gradient based optimization. In fact, the Nash Equilibria here is a minimax problem, thus the optimal policy is a saddle points in the solution space, which minimizes the loss for one agent, and maximize the loss for the other one.
The same problem is faced with any other problem formalized as minimax. One of such problems that are very well known in the AI community, are GANs, where indeed the discriminator tries to maximize it's accuracy, and the generator to minimize its loss.

To overcome this problem, in the literature 3 main approaches have been proposed:

∗ Regret minimization: here Game Theory is involved to minimize regret, which is defined as the difference between the result gotten, and the optimal that could have been achieved, for example Counterfactual Regret Minimization[18]. However, such methods are very delicate, requires high knowledge of Game Theory, and they are very computationally expensive compared to naive RL

∗ Naive RL: ironically, naive RL even though not theoretically sound, has been shown to be highly effective, even more than Game Theoretical ones, for example in the case of AlphaHoldem

∗ Population based: instead of using a single agent, in this scenario a population of agents are used, which helps generalization and convergence (as for GANs, multiple discriminators usually are used to help convergence). In this category, multiple proposals can be found, ranging from the least expensive, such as fictitious self-play, which puts the agent against an agent using reservoir sampling against past versions of itself, to the most expensive one, which is Adversarial training, where best response of the current agent are trained, and then used to improve the main agent. A special one worth noting is used by AlphaStar[19] from DeepMind, which uses league play to improve generalization

In order to do so, we will initialize a random policy represented by a neural network, and its corresponding value function, and we will train the agent against itself, and once we gain enough data, perform a learning step using PPO.

This has 2 main advantages:

1. Double collected data: since the games are composed by two identical copies of the same agent, we can use both trajectories to learn, thus needing half the games to arrive to our desired batch-size, where instead in all the other cases, for example for the case against a rule-based-agent, half of the data had to be thrown away.

2. Computationally cheap: avoiding adversarial or population based learning approaches, we don't need to learn multiple agents. Even in the naive case where we would use N-sized population, we would have a $1/N$ training efficiency for the main agent.

6.2.1 Infrastructure

Most of the work it's been spent on developing the architecture to make this as efficient as possible. In order to do so, extensive performance tests have been done in order to locate the bottlenecks of the pipeline, and to code possible alternatives to overcome them.

During such tests, we looked for functions that were called many times, and they required a lot of time. Many "slow" functions were called only a handful of times, thus an improvement of them would make no observable difference. Instead 2 main bottlenecks have been found:

1. Agent cloning: SP requires to clone an agent in order to create an adversary. Such cloning, since the agents are stateful as they carry the current game state they are playing, were implemented as deep cloning, which happened to also clone the neural network of the agent, which lead to a huge overhead. In order to overcome this, lazy-initialization and object pointers have been used to delay the creation of the agent networks, thus allowing to share the same one using pointers.
   This simple trick decreased the running time of around 40%.
2. Data collection: while the agent plays a game, the environment would collect the data generated from them, in order to then pass it to the relative algorithms. Such algorithms would then take the data, process it in order to make it efficient to then learn on top of it, and store it. This processing was very expensive and was requiring a lot of time, thus it's been delayed from game to game, to only when there is enough data to learn, and only than, by exploiting multi-processing, it would have been processed. This fix allowed the project to reduce its running-time by around 14%.
3. Graph computation: Tensorflow is the deep learning framework used for the training and definition of the neural networks. It allows out of the box eager and graph execution: the former, executes the code as-is, where the second one records the operations, so that if that code is run again, it already knows the necessary computation, and can optimize them.
   This improved the performance by an additional 5%, however depends highly on the underlying hardware.

The following is the final pipeline:

Algorithm 10: Training pipeline

```
Initialize agent $A$ with respective $\pi$ and $V$ function.
Initialize game $G$ for 2 players.
Initialize algorithm $alg$ with respective data-buffer.
for $i = 1$ to $N$ do
    for $j = 1$ to $M$ do
        Enemy = $A$.clone()
        $S, S', A, R, D, M \leftarrow$ Play_Game($G$, [$A$, $E$])
        $alg$.store($S, S', A, R, D, M$)
    end
    if $alg.has\_enough\_data()$ then
        $alg$.preprocess() # exploiting multi-processing
        $alg$.learn($A$)
    end
    if $i$ % $EVAL$ == $0$ then
        evaluate($G$, [$A$, RandomAgent()])
        evaluate($G$, [$A$, RuleBasedAgent()])
    end
end
```

6.2.2 A2C-SP-Agent

In this section we will present the training of the Advantage Actor Critic agent, trained using self-play against himself. The agent is not pre-trained in any way, and instead initialized randomly as reported earlier. It Temporal difference learning to reduce the variance for both actor, $\pi(\cdot|s)$, and critic, $V(s)$.

The following are the hyper-parameters used for the training:

A2C	
Batch-size	512
Stepsize	$1e-4A, 3e-4C$
Learning steps	1
Training steps	100.000
Evaluate every	1000
Evaluate with n. games	1000

Table 6.3: Hyperparameters for A2C-SP-Agent.

The following are the evaluations of the trained agent against a random agent, used just as baseline to analyze the training, and against the rule-based-agent, which we want to emphasize that is unseen for the A2C-SP-Agent.

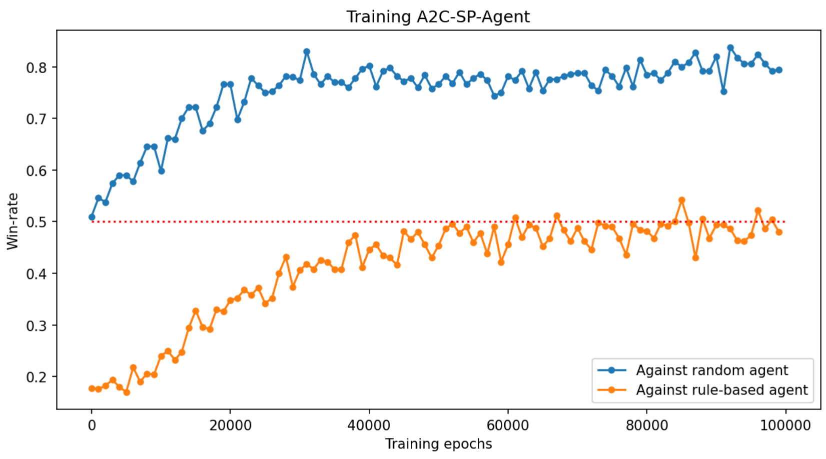

Figure 6.3: Training of A2C-SP-Agent

Clearly the agent is able to improve by learning against itself, and to generalize against unseen adversaries. As can be seen from this plot, the agent after 100.000 training steps is plateauing around 50% against the rule based agent. This is a good result, in fact, theory says that a Self-Play (SP) agent, should learn a Nash Equilibrium, which by definition ensures not to loose in expectation. Such definition means that potentially our agent might learn that the best it can do is to try to tie every game, and never win. If that's the case, such agent would have around 50% win-rate against every other "good player". However, we do not believe that this is the case, as instead, it's just a matter of time for the SP agent to surpass the rule-based, as done by the DQN in the adversarial training shown before.

However, A2C it's known to be very data-hungry, as its on-policy definitions requires new data after every gradient update.
Even though the code and the theory provided the ability of going off policy by correcting the gradient by the importance sampling factor, such thing is rarely being done as it might

introduce a lot variance:

$$\begin{aligned}
\nabla_\theta J(\theta) &\approx \mathbb{E}_\pi\left[\sum_t \nabla_\theta \ln \pi_\theta(a_t|s_t)A_t\right] \\
&\approx \mathbb{E}_\pi\left[\sum_t \frac{\nabla_\theta \pi_\theta(a_t|s_t)}{\pi_\theta(a_t|s_t)}A_t\right] \\
&\approx \mathbb{E}_b\left[\sum_t \frac{\pi_\theta(a_t|s_t)}{b(a_t|s_t)}\nabla_\theta \ln \pi_\theta(a_t|s_t)A_t\right] \\
&\approx \mathbb{E}_b\left[\sum_t \frac{\pi_\theta(a_t|s_t)}{b(a_t|s_t)}\frac{\nabla_\theta \pi_\theta(a_t|s_t)}{\pi_\theta(a_t|s_t)}A_t\right] \\
&\approx \mathbb{E}_b\left[\sum_t \frac{\nabla_\theta \pi_\theta(a_t|s_t)}{b(a_t|s_t)}A_t\right]
\end{aligned}$$

Indeed, if $\pi(s|a) \gg b(s|a)$, that ratio becomes larger and larger, and that might happen if $A_t \gg 0$ with just few steps of gradient descent.

This impossibility of reusing data multiple time is the reasons which Proximal Policy Optimization (PPO) and Trust Region Policy Optimization (TRPO), were born, and we believe that is what is limiting the agent to learn better policy in reasonable amount of time.

6.2.3 PPO-SP-Agent

In this section we will present the training of the PPO agent, trained using self-play against himself. We will use PPO as it solves the problem of not being able to reliably use multiple time the same data without risking to have exploding gradients. Also this agent is not pre-trained in any way, and instead initialized randomly as reported earlier. The training algorithm is set to use 10 different processes for performance speed-up and it's going to use Temporal difference learning to reduce the variance for both actor, $\pi(\cdot|s)$, and critic, $V(s)$, as done for the A2C-SP-Agent.

However, since it required multiple training steps, in order to have a fair evaluation, the training consisted in only 50.000 steps, which had a wall clock time equal to the A2C-SP-Agent with 100.000 steps.

The following are the hyper-parameters used for the training:

PPO	
Batch-size	512
Stepsize	$1e-4A, 3e-4C$
Learning steps	$10A, 3C$
Clipping ϵ	0.1
Training steps	50.000
Evaluate every	1000
Evaluate with n. games	1000

Table 6.4: Hyperparameters for PPO-SP-Agent.

The following are the evaluations of the trained agent against a random agent, used just as baseline to analyze the training, and against the rule-based-agent, which we want to emphasize that is unseen for the PPO-SP-Agent.

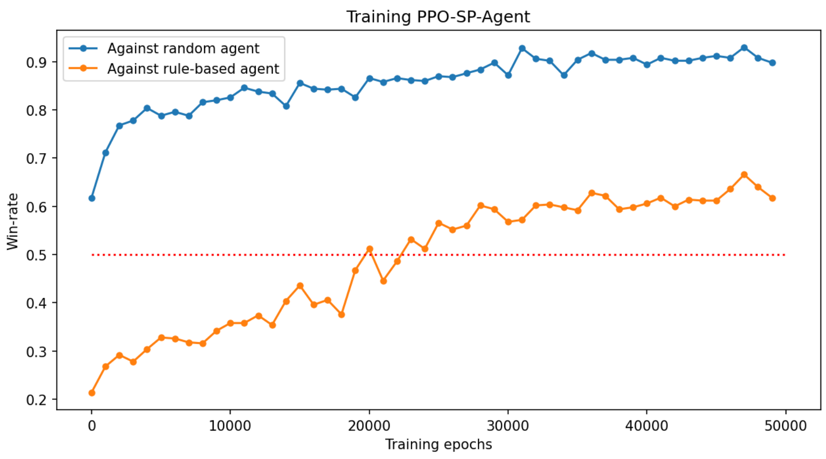

Figure 6.4: Training of PPO-SP-Agent

As for the A2C-SP-Agent, the PPO-SP-Agent is able to improve by learning against itself, and to generalize against unseen adversaries, such as the rule-based-agent. However, contrary to what has been observed in the A2C case, the PPO agent is not plateauing , which makes us believe that it has yet more room for improvement.

We would like to emphasize how the agent now is able to reach over 60% of win-rate against the Rule-based-agent, for the first time ever for an actor-critic method, as the only one that was able to do so was the DQN agent, by specifically learning how to beat such agent. This shows that self-play is not only improving, but possibly even more effective than the adversary learning for actor critic in this case, probably due to the curriculum learning that the agent is receiving, by going against always more challenging agents.

As a comparison, those are the A2C-SP-agent compared to the PPO-SP-agent on clock wall time:

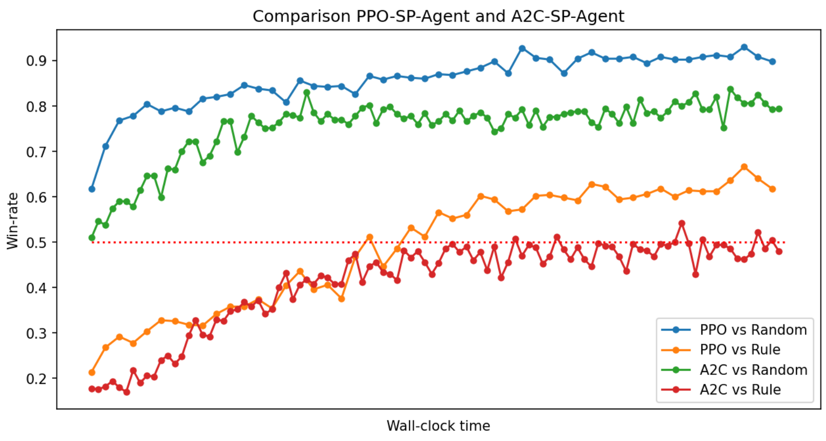

Figure 6.5: Comparison PPO-SP-Agent and A2C-SP-Agent

It can be seen how at the beginning of the training, PPO-SP-Agent is not benefitting from the multiple training, as the training data comes from very random games since the agent has yet to learn, thus the signal coming from it is very noisy. However, once the agent actually starts to learn, there are a lot of insights that the agent can draw from the training data, and it can be seen how it benefits from this from the fact that it outperforms the A2C agent.

The reported result is just a single run of the training, however, training usually do not differs from each other more than $\pm 3\%$, and the difference between A2C and PPO can easily be observed in all the training performed.

However, none of the two methods received any hyper-parameters tuning, and instead, we used the one that are suggested as default in the respective papers. RL is notorious to be highly hyper-parameters tuning dependent, as by changing even only one of them, some methods can be shown to outperforms others. On the other hand, we have no reason to show that PPO works better than A2C, and instead, we believe that the ability to pick a good default value for hyper-parameters is one of the key properties that a good algorithm should have. Once we explored all interesting proposals for the algorithms, and we have an idea of which one works better, we indeed preform a grid search over the hyper-parameter space of the best one.

6.2.4 Sym-Clip-SP-Agent

PPO has an intrinsic pessimistic view of the RL problem, an it can be seen as reward shaping procedure. In fact, in its original paper, they explicitly say that if we incur in a negative transition, the method will still learn from it. This can be seen from the clip-PPO update formula:

$$L^{PPO}(\theta) = \hat{\mathbb{E}}_t \left[\min\left(r_t(\theta)\cdot A_t, \text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\cdot A_t\right)\right]$$

In fact, if A_t is negative, thus we took a below-average action, the clipping would not occur, as we are going to take the minimum of the two terms, and thus the unclipped one.

This is reasonable to assume in normal environments, since there is no reason to assume that if an action was below-average now, all of a sudden becomes above-average, as there is always some greediness in RL problems usually.
However, we believe that this is not true in the case of Self-Play: indeed, an agent at the beginning of the training, learns a policy that base it's reasoning on it's ability to play future moves. However, as those future moves changes, the initial ones also have to change, because we might have learnt to exploit something new, so an initial action might change from positive to negative.

In order to tackle this face of the problem, we tried to use a symmetric clipping, thus no matter if the agent does something positive or negative, we are going to treat this equally, which changes the target to the following:

$$\begin{aligned} L^{\cancel{PPO}Clip}(\theta) &= \hat{\mathbb{E}}_t \left[\cancel{\min(r_t(\theta)\cdot A_t,} \, \text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\cdot A_t\cancel{)}\right] \\ &= \hat{\mathbb{E}}_t \left[\text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\cdot A_t\right] \\ r_t(\theta) &= \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} \end{aligned}$$

The following are the hyper-parameters used for the training:

Sym-Clip	
Batch-size	512
Stepsize	$1e-4A, 3e-4C$
Learning steps	$10A, 3C$
Clipping ϵ	0.1
Training steps	50.000
Evaluate every	1000
Evaluate with n. games	1000

Table 6.5: Hyperparameters for Sym-Clip-SP-Agent.

The following are the evaluations of the trained agent against a random agent, used just as baseline to analyze the training, and against the rule-based-agent, which we want to emphasize that is unseen for the A2C-SP-Agent and the PPO-SP-Agent.

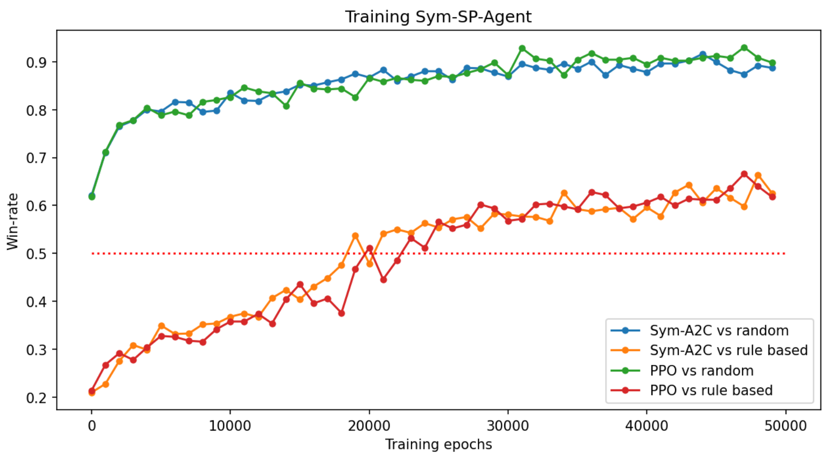

Figure 6.6: Training of Sym-Clip-SP-Agent

No significant difference has been observed with respect to the ordinary PPO agent with the same hyperparameters, probably due to the fact that the stepsize that has been used is pretty small, thus it's unlikely that in 10 GD steps the threshold severely surpassed, encoring in the case pointed out.

We suspect that the reasoning behind the min in PPO is not to have a pessimistic point of view of the training, but more a hidden variance reduction technique.
The advantage function reduced the variance of the gradient changing the estimation of the reward:

$$\begin{aligned}\nabla_\theta J(\theta) &\approx \mathbb{E}_\pi\left[G_t \frac{\nabla \pi(a_t|s_t)}{\pi(a_t|s_t)}\right]\\ &\approx \mathbb{E}_\pi\left[(R_t + \gamma V(s_{t+1}) - V(s))\frac{\nabla \pi(a_t|s_t)}{\pi(a_t|s_t)}\right]\\ &\approx \mathbb{E}_\pi\left[(R_t + \gamma V(s_{t+1}) - V(s))\frac{\nabla \pi(a_t|s_t)}{\pi(a_t|s_t)}\right]\\ &\approx \mathbb{E}_b\left[(R_t + \gamma V(s_{t+1}) - V(s))\frac{\nabla \pi(a_t|s_t)}{b(a_t|s_t)}\right]\end{aligned}$$

Instead PPO consider that the last ratio, $\frac{\nabla \pi_{\text{new}}(a_t|s_t)}{\pi_{\text{old}}(a_t|s_t)}$ might get very big after multiple steps if $\pi(a_t|s_t)_{\text{new}} \gg \pi_{\text{old}}(a_t|s_t)$, which happens only if $A_t > 0$, and thus the action is reinforced positively. If instead $A_t < 0$, then the action is reinforced negatively (decreased it's probability to be sampled), then $\pi(a_t|s_t)_{\text{new}} \leq \pi_{\text{old}}(a_t|s_t)$ which means that $\frac{\nabla \pi_{\text{new}}(a_t|s_t)}{\pi_{\text{old}}(a_t|s_t)} \leq 1$, not causing any high variance, therefore if $A_t > 0$, then we should clip the ratio, in order to avoid having exploding gradients, but if $A_t \leq 0$ than the clipping is not necessary, since no high variance should be encountered.

6.3 Fictitious Self Play

Fictitious Self Play (fSP) has been introduced to tackle the problem of generalization, mainly in partial information games. If with perfect information two player zero sum games Self-Play is guaranteed to converge to a Nash-Equilibria, thus not to lose in expectation, this is no more valid in partial information games, specifically in repeated information games.

fSP, contrary to SP, is based on the idea that going against a pool of agents should help generalization. In order to accomplish this, a buffer is initialized to store the history of the agent, and through reservoir sampling, an opponent is chosen among them.

Reservoir sampling has been chosen as sampling technique since it given more probability to more recent agents, property derived trivially by its definition:

$$P(i\text{-th element at } k\text{-th iteration}) = \begin{cases} 1, & \text{if } i \leq k \text{ (uniformly chosen from the first } k \text{ elements)}, \\ \frac{k}{i}, & \text{if } i > k \text{ (chosen with probability } \frac{1}{i}\text{)}. \end{cases}$$

Thanks to such sampling, at each game, an agent opponent is chosen in the recent history of the trained agent, and after some games, the main agent is trained on such data:

Algorithm 11: Fictitious Self-Play Algorithm

```
Input: Number of iterations: num_iterations
Data: Number of games per iteration: num_games
Initialize: agent with a random policy
Initialize: fictitious opponents buffer $\mathcal{D}$
for iteration = 1 to num_iterations do
    for game = 1 to num_games do
        Generate fictitious opponents by sampling from $\mathcal{D}$
        Let agent play against fictitious opponents
        Update agent's policy using RL algorithm
        Store agent policy in $\mathcal{D}$
    end
    Update agent's policy to the new current policy
end
```

If the idea behind such method might be correct, it has the main drawback that half of the data is discarded because it comes from other agents, which might be far from the current agent policy, and also, the signal might be noisier, since the agent goes against weaker opponents, which are his past versions.

Considering such drawbacks, it is worth the additional computational cost only if the generalization of the agent is a crucial problem. It has been shown to be not only successful, but necessary for the convergence of some of the most complicated agents developed in the history of AI, such as AlphaStar.
DeepMind trying to develop an agent able to play the game of StarCraft II, has seen that there was little to no improvement in the agent if left training from scratch. Therefore, decided to collect some data, and thorough imitation learning and supervised learning, pre-trained a model that approximate the average player way of playing the game, in order to give some prior knowledge to the agent.
Once such pre-trained agent has finished training, then it was left training in SP: however, even though it was getting better, they observed that the advancement was little compared to the expectation, so not even a pre-trained model with SP was enough for such problem.
In order to overcome this, they used a variant of fSP that they called League Play[19], where not only the agent would play against old versions of itself, but they introduced additional players that would learn a best response to the main agents (exploiters), in order to avoid that the agent has trivial flaws, but also some "regularization" player, to avoid that none of the previous overfits on the others (league player).

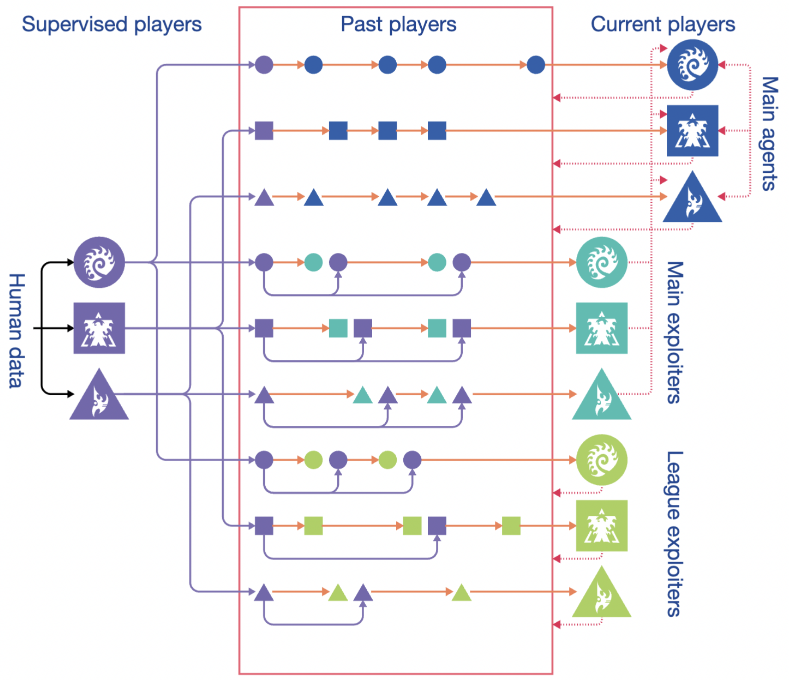

Figure 6.7: League play structure, from the AlphaStar paper

6.3.1 Sym-Clip-fSP-Agent

Given the reasoning behind fictitious Self-Play, we tested the symmetric clipping Actor Critic agent with such agent.

However, fSP introduces additional hyperparameters that have to be tuned, in particular the size of the buffer, which defines the probability of keeping an agent to $1/|D|$, and every how many epochs add a new agent.

In particular, the buffer size determines how close we want to stay to SP. Indeed, if the buffer size is 1, then the agent goes against a version of itself that is very recent, approximating a behavior similar to the one in the design of DQN.
Instead, the insertion rate, defines how diverse we want the agents: if it is large, than the agent has the risk to go against a very noisy agent, since it may be very old, but if it is too small, then the agent are not diverse enough to improve generalizations.

In order to have a fair comparison, we kept fixed the hyperparameters of the RL algorithm, even though, as for the SP version, such hyperparameters have not been optimized, but just set to default one:

Sym-Clip	
Batch-size	512
Stepsize	$1e-4A, 3e-4C$
Learning steps	$10A, 3C$
Clipping ϵ	0.1
Training steps	50.000
Evaluate every	1000
Evaluate with n. games	1000

For the fSP algorithm instead, we added a little tweak to ensure that the agent goes also against a good agent which is during sampling, we would sample with probability $1-p$ an agent in the buffer $\mathcal{D}$, and with probability p the current agent, and if the current agent is selected as enemy, then also the data produced by it is used for the training:

$$Enemy_t = \begin{cases} e \sim \mathcal{D} & \text{with probability } 1-p \\ \text{main agent} & \text{with probability } p \end{cases}$$

Following are the hyperparameters used for the fictitious self play algorithm:

Sym-Clip	
p	30%
$\lvert\mathcal{D}\rvert$	20
Insert every	20 epochs

Table 6.6: Hyperparameters used for fictitious self play

The following are the evaluations of the trained agent Sym-Clip-fSP-Agent against a random agent, used just as baseline to analyze the training, and against the rule-based-agent, compared to the its version trained with self play Sym-Clip-SP-Agent.

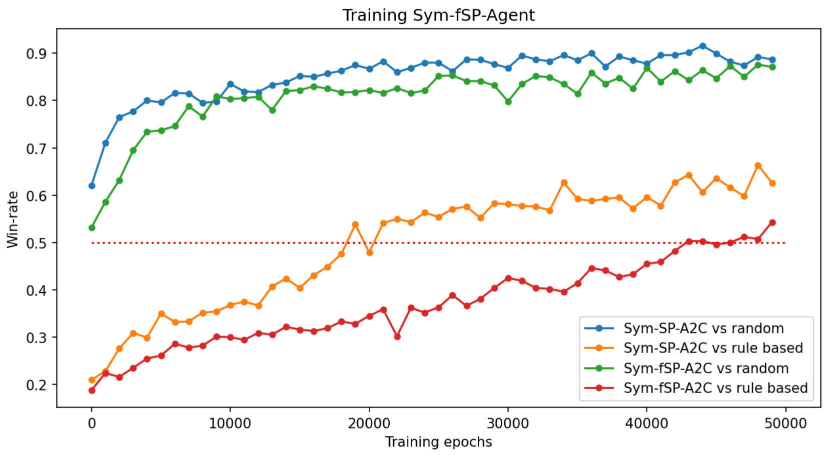

Figure 6.8: Training of Sym-Clip-fSP-Agent

Unfortunately, seems that fS is actually hurtful for the performance of the agent, and the main reasons that could cause this are:

* noisy data: indeed, playing against different opponents creates the problem that the agent has to generalize, and generalize to agents that could be very far. The idea behind self-play is to approximate the minimax algorithm, which leads to a Nash Equilibria, but that happens if and only if the enemy is itself the optimal player, but if we have a buffer of them, might happen that we bias our training in a direction not concordant to the optimal strategy, thus injecting noise

* fewer data: the data collected by the agent during fSP is likely its own perspective, where instead in SP it could also collect the data of the adversary, as it's a clone of the actual agent. This asymmetry means that fSP needs double the data, thus double the time to even the training steps. Thus, even though it might happened that in expectation fSP might behave better than SP, in a fix-budget, it needs more time.

6.4 Other tested algorithms

In addition to fPS and SP, other 2 alternatives have been tested, and are going to be presented here, however, no score will be shown as both of them appeared to be too slow to have a meaningful training in reasonable time.

6.4.1 Population based Self Play

In order to investigate which of the previous 2 points is causing the problem of worsening the performance, we implemented population based Self Play (PbSP).
In PbSP, contrary to what is done in fSP, there are P agents, all actively trying to learn. For this reason, the agents never meet enemies that are not the best possible, event that is quite likely in the fSP setting.

Algorithm 12: Population-Based Self-Play

```
Input: Initial population of agents
Data: Number of generations $N$, Population size $P$
for $i \leftarrow 1$ to $N$ do
    for $j \leftarrow 1$ to $P$ do
        Randomly select two agents $A$ and $B$ from the population;
        Let $A'$ and $B'$ be copies of $A$ and $B$;
        Play($A'$, $B'$);
        Record the result of the game between $A'$ and $B'$;
    end
    Improve $A$ and $B$ via RL
end
```

However, this algorithm by definition, requires that multiple agents are learning all at once, thus there is an overhead due ti that, but also there is no "best/current agent" thus might happen that the one taken in consideration, might be worse than the rest of the population.

For this reason, we did not proceed with the training, as it would have required substantial more computational resources than the previous ones.

6.4.2 Adversarial training

This is a special case of PbSP, where we were training the main agent against itself and a second agent, that was specifically and only trained against the main one.
Thanks to such simplification, it's computational price is not that significantly more than the earlier one, and might still lead to better performance, as the second agent specifically learns a best response to the first one.
Indeed, if the main agent converges to a Nash Equilibria, also the best response one does, however the trajectory taken by the two agents are very different, and are induced by the training history.

Algorithm 13: Adversarial Self-Play Algorithm

```
Input: Number of iterations: num_iterations
Initialize: agent $A$ with a random policy, and the exploieter enemy $E$
for $iteration = 1$ to $num\_iterations$ do
    Play game $A$ against $E$
    Play game $A$ against $A$
    Update $A$ and $E$ policy using RL algorithm with the collected data
end
```

As for the agent, we used the PPO-agent, since it led to one of the best results, both for the main agent and for the exploiter agent. However, a nice alternative that could be

tested is the version where the exploiter is a DQN agent, since it's faster to learn, and thus to adapt its best response to the new agent policy.
For the main agent, we used the same hyperparameters of the PPO-SP-agent, in order to have a fair comparison of the methods with the other agent.

PPO	
Batch-size	512
Stepsize	$1e-4A, 3e-4C$
Learning steps	$10A, 3C$
Clipping ϵ	0.1
Training steps	50.000
Evaluate every	1000
Evaluate with n. games	1000

Table 6.7: Hyperparameters for PPO-AdvSP-Agent.

In order to efficiently apply such algorithms, we decided to use a much more aggressive learning for the exploiter agent. Thanks to that, the exploiter can learn faster a best response to the newly updated main agent, thus being more effective.

PPO	
Batch-size	512
Stepsize	$1e-3A, 1e-3C$
Learning steps	$10A, 5C$
Clipping ϵ	0.1
Training steps	50.000
Evaluate every	1000
Evaluate with n. games	1000

Table 6.8: Hyperparameters for PPO-AdvSP-Exploiter-Agent.

The following is the comparison of PPO-SP-Agent with PPO-AdvSP-Agent, which even though required more learning, has a reasonably small additional computational cost.

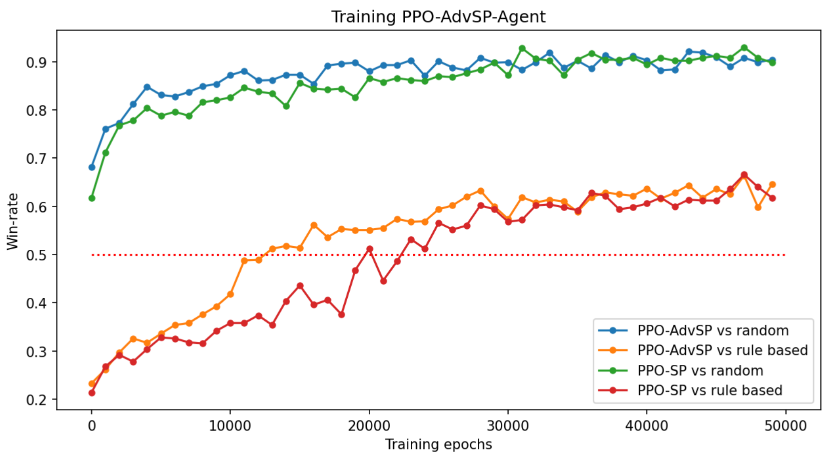

Figure 6.9: Training of Sym-Clip-fSP-Agent

Seems that Adversarial Self Play training outperforms naive-SP mainly at the beginning, which is reasonable considering that the agent not only has the signal of the "at the moment" best agent, itself, but the signal of its best response, thus is much less susceptible to best response attacks, and able to patch its flaws faster.

6.5 Other tested reward functions

Until now, the reward function that we optimized for exploited the formalization of the game, thus assuming that maximizing the gained points is a good proxy for maximizing the win-rate, which is the actual end goal of the project.

Thus, assuming that r_t^a describes the points at turn t for the agent a, the current reward function is defined as follows, with $\gamma = 1$:

$$G_t^a = \mathbb{E}\left[\sum_{t=0} \gamma^t r_t^a\right]$$

However, this might incentivize the agent to wait for a move/action that might get him with cumulative reward > 60 if that might help him to get more points later on.

In order to avoid this behavior, we also tested a different formulation of the reward function, which can be described as follows, with a being the agent and e the enemy:

$$G_t^a = \mathbb{E}\left[\sum_{t=0} \gamma^t \begin{cases} 1, & \text{if } \sum_{i=0}^{t-1} r_i^a < 60 \text{ and } \sum_{i=0}^{t} r_i^a > 60 \\ -1, & \text{if } \sum_{i=0}^{t-1} r_i^e < 60 \text{ and } \sum_{i=0}^{t} r_i^e > 60 \\ 0 & \text{otherwise} \end{cases}\right] \tag{6.1}$$

With this formulation, the agent receives a reward of 1 on the transition that allowed him to pass from < 60 cumulative reward to > 60 as cumulative reward, -1 on the transition that allowed the enemy to pass from < 60 to > 60, 0 in all other cases.

Thanks to that, the agent is incentivized only to get over 60. If $\gamma = 1$, the agent will have no reason to be greedy in doing such switch, which is also reasonable in expectation, since it will learn if waiting is worth it, where instead if $\gamma < 1$, then the agent is also incentivized to reach that switch as soon as possible.

To test this, we have taken the PPO agent, and switched the reward function with the new one, and used the following hyperparameters:

PPO	
Batch-size	512
Stepsize	$5e-4A, 5e-4C$
Learning steps	$10A, 5C$
Clipping ϵ	0.1
Training steps	100.000
Evaluate every	5000
Evaluate with n. games	1000

Table 6.9: Hyperparameters for PPO-Binary-SP-Agent.

The following is one round of training of the Binary-reward PPO agent and the Dense-reward PPO agent:

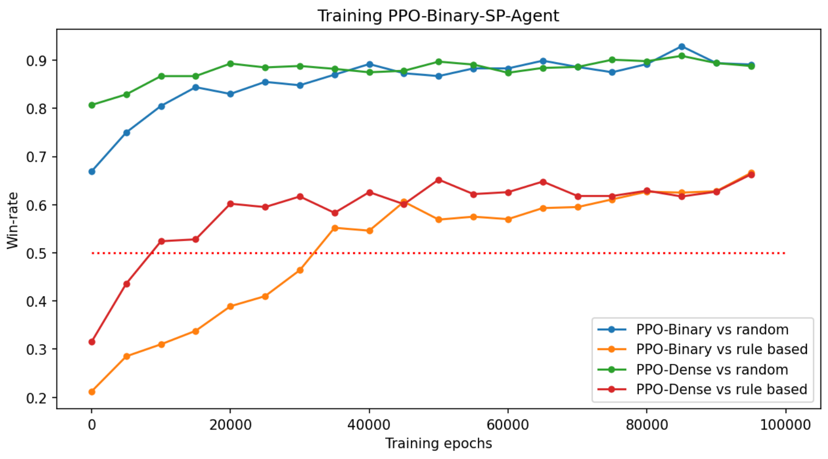

Figure 6.10: Training of PPO-Binary-SP-Agent

The sparsity of the reward function actually hindered the training performance of the agent, since it has to rely on trials and errors to figure out what led him to that $1/-1$ reward. Even though maybe with more resources this approach might be better, it's too slow to converge.

6.6 Other tested state representation

Given the second option for the reward function it's very crucial for the agent to know what's its current cumulative points, in order to accurately decide what's the best action to perform, as the reward is highly dependent on the state and the current points of the agent.

In order to make the agent know as precise as possible its score, we redefined the state space. In particular, in the original formulation, the score of the agent was encoded as a single entry in the input tensor, constraining the integer value of the score.
However, we imagined that that was not the most precise way fo telling the agent its score, thus we dropped that single entry, in favour of a 121-one-hot encoded vector, where the 1 would be inserted in the position of the agent current score.

To test this, we have taken the PPO agent with binary reward, and switched the state representation with the new one, and used the following hyperparameters (identical for the compared agent):

PPO	
Batch-size	512
Stepsize	$5e-4A, 5e-4C$
Learning steps	$10A, 5C$
Clipping ϵ	0.1
Training steps	100.000
Evaluate every	5000
Evaluate with n. games	1000

Table 6.10: Hyperparameters for PPO-Binary-NewState-SP-Agent.

The following is one round of training with the new state formalization and the old state formalization:

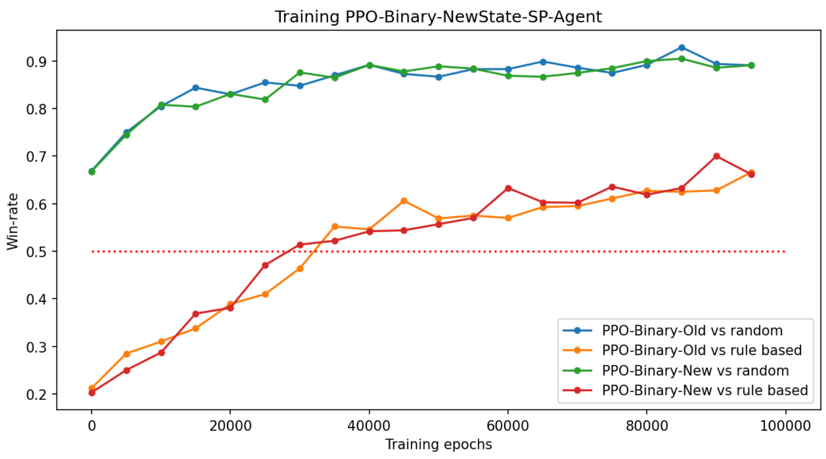

Figure 6.11: Training of PPO-Binary-NewState-SP-Agent

For a full comparison, we tried also the version with a time-discounted reward function, with the same hyperparameters, with the following result:

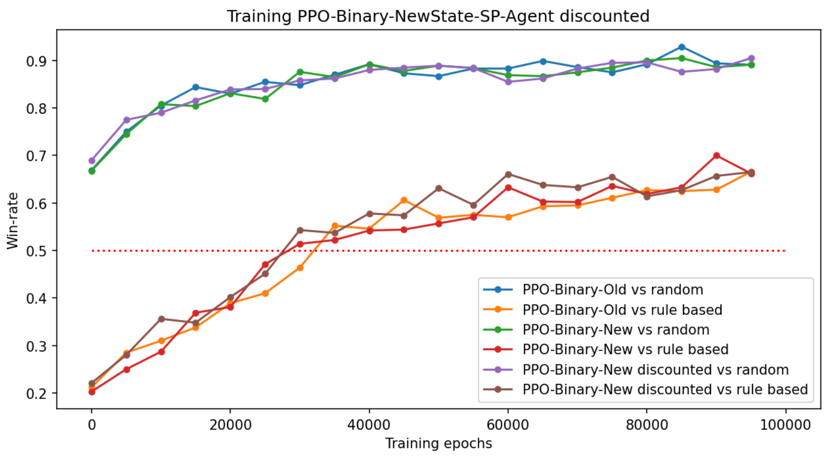

Figure 6.12: Training of PPO-Binary-NewState-SP-Agent with time discount

As can be seen from the plots, such re-formalization of the state brought no statistically significant improvement to the agent learning history, which considered together with the fact that such re-formalization increased significantly the network-size, we did no proceeded to use it.

6.7 Other tested architectures

Additionally, we also tested different architectures. Given a large enough network, no difference has been observed changing depth and width, if not an slight decrease in performance, which is well known in Reinforcement Learning [13].

However, we severely tested regularization techniques as they play a fundamental part in improving the performances of deep neural networks. However, we observed little to no improvement over the non-regularized version. This phenomenon is also well known in the DeepRL community, and it's been observed many times [13].

The following is the training of the best model with no regularization, with additional Dropout layers in between layers.

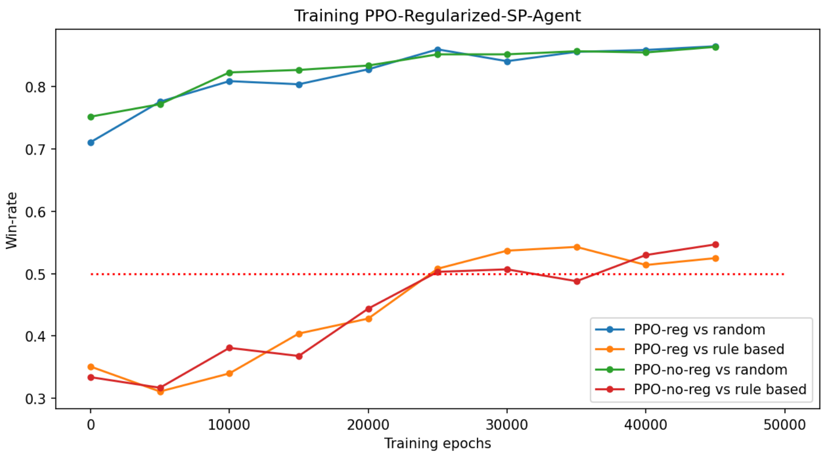

Figure 6.13: Training of PPO-Regularized-SP-Agent

6.8 Final Agent

Finally, once had an overview of all the methods, we decided that the PPO-SP-Agent was the best one, and thus proceeded to pick it as final one.
Given that, we proceeded to train it for longer, to see what performance improvement we could get with more training.
However, if at this point usually there should be the part with the hyper-parameter search either with a grid search or with a Bayesian search, it's not feasible to do so with agents like this. If usually performance of a model of this size can be roughly estimated in order to tens of minutes when dealing with supervised learning, here there is also the additional cost of the simulation, which are hard to parallelize due to the fact that multiprocessing would require serialization of the agents, including the neural network. For this reason, a training of 100.000 epochs, might require 10 hours of training in the case of PPO Self Play agents with 10 updates for the actor, and more in case of others algorithm or with more updates.

For this reason, we picked reasonable hyperparameters for the training, and hoped that they would be good enough to allow the agent to converge to a good solution:

Final PPO	
Algorithm	Self-Play
Discount	1
Batch-size	512
Epsilon	10^{-8}
Stepsize	$5e-4A, 5e-4C$
Learning steps	$15A, 5C$
Clipping ϵ	0.1
Training steps	200.000
Evaluate every	5000
Evaluate with n. games	1000

Table 6.11: Hyperparameters for final PPO Agent.

The training of such agent required $\sim$ 17 hours on a single GPU. The main bottleneck of the process is the data generation, which is alleviated with model-based RL if it's the environment the expensive part, however here it's the policy that consumes most of the time, since it has do a forward pass for every move, thus not exploiting any for of GPU-optimization, which might be a possible future improvement, in order to achieve more extensive training in the same time-window, thus possibly achieving better performances.

The following is the training history of the final model on the 200.000 training steps with the reported hyper-parameters lasted ~ 17:

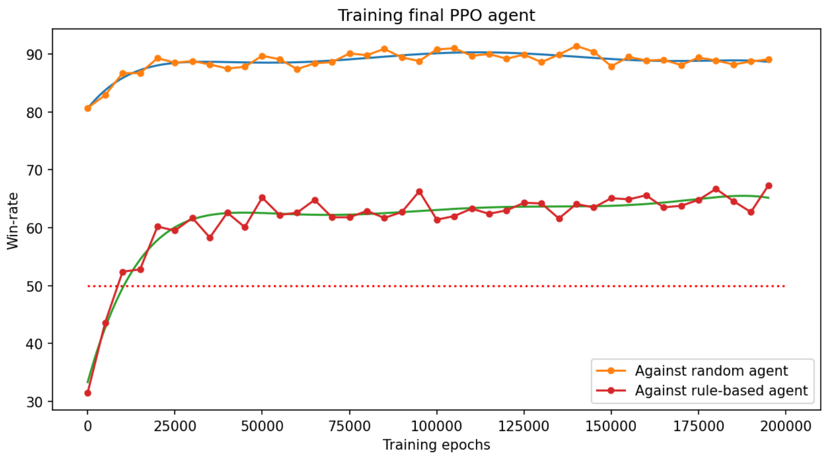

Figure 6.14: Training of final PPO agent

It's clear from this plot that the 50.000 threshold for number of episodes for testing purposes was fine, as most of the improvement happens there, which is usual for problems such as this one: for example, AlphaZero required 3 days of training to reach very good performances, but more than 70 days to surpass reach its full ability.
However, differently to what done by DeepMind, we used a rule-based agent and a random agent to evaluate the improvement of the RL-agent, where instead they used Elo rating, which however would have required much more computational resources.

6.8.1 Evaluation of final agent

Until now, we used the win-rate against a random agent and a handcrafted rule based agent as a proxy for goodness of an agent. However, since the beginning we aimed at reaching a Nash Equilibria, which by definition in 2 player 0 sum games has the property that the agent will not loose in expectation.
As previously said, this implies that potentially, a NE agent won't also even win, thus using win-rate as a proxy is not theoretically sound.
Instead, it's theoretically sound to say that given a player that plays a NE in a 2p0s game, its best response will tie with it in expectation.

In order to assess it, in simple normal form games, algorithms such as Fictitious Play or Counterfactual Regret Minimization are used to find a strategy that is a best response to a certain player.

However, it's been proven that if a mixed strategy m is a Best Response to an agent a, any of the pure strategy s involved in m are Best response of a. For this reason, if we fix the opponent strategy, we can use a Q-learning algorithm that becomes greedy with time to approximate the best response, as it's guaranteed to exists a deterministic policy that best exploits tha mixed agent strategy.
On the other hand, this approach is computationally expensive, since it requires to train a best response agent against the agent we want to evaluate, which can take hours for each single BR training. For this reason, only to test the final agent, we will fix as enemy strategy

our final agent, and we will train a DQN to exploit it.
For the training, the following hyperparameters have been used:

Q-learning	
Batch-size	512
Stepsize	$1e-3$
Learning steps	1
Replay-mem size	10.000
Policy	ϵ-greedy
Training steps	10.000
Evaluate every	1000
Evaluate with n. games	500

Table 6.12: Hyperparameters for DQN BR for final agent.

We run the code for 5 times with 5 different seeds, and tracked the performance of the DQN training, and the following is the result:

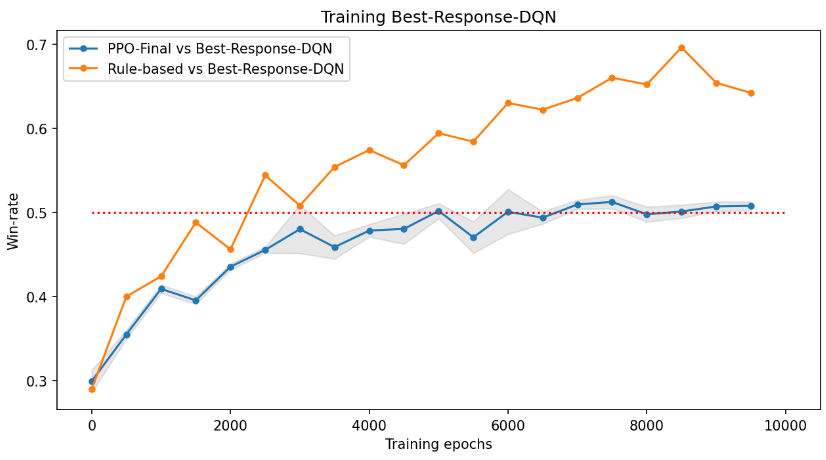

Figure 6.15: Training of BR for final agent

For reference, we trained one time also the DQN against the reference rule-based agent with the same hyperparameters to have a comparison.

Clearly, our final agent it's much harder to exploit, showing that even though Self-play is not guaranteed to converge to a Nash Equilibria, our agent it's pretty close, as it's exploitability is around 1%.

Chapter 7

Conclusions and future works

In this thesis we investigated the effectiveness of pure Reinforcement Learning approaches for the game of Briscola.
To do so, we formalized the game as a RL problem, with the addition of the Self-Play algorithm in order to make the environment able to evolve with time. Then, via algorithms such as DQN, PPO, A2C, we trained an agent to improve initially against a fixed hand-crafted rule based agent, and then against itself, via several Game Theory inspired techniques in order to achieve the best result possible.
Finally, a small hyperparameters tuning has been performed in the final agent, which was composed by a actor and critic neural networks, and was trained using PPO.
The final agent of such training outperformed any other AI agent we developed, and learned zero-shot to beat the handcrafted rule based agent we used as benchmark, showing that the learned policy was highly effective, thus hinting that was close to the Nash Equilibrium of the game.
Finally, we trained a best response agent against the fully trained final model to evaluate in a precise manner its exploitability. The results clearly show that the final agent is almost optimal, as a heavily trained best response manages to get around 51% win-rate against it, thus showing that even with thousands of games against such agent, it's very hard to find any weakness of it.

As future work we aim to expand this work in the team setting, with the 2v2 scenario, in order to investigate the effectiveness of RL algorithms in settings where both cooperation and coordination are required. In addition to that, we aim to fully explore the ability of this agent, by performing a much more comprehensive hyperparameter tuning, in order to assess which features are better in the state representation, which RL algorithm is more effective, and which Game Theoretical setting leads to the best possible result in the least amount of time, and then transferring this knowledge to the 2v2 setting, which will inevitably present many more challenges.

Bibliography

[1] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, *et al.*, "Training language models to follow instructions with human feedback," *Advances in Neural Information Processing Systems*, vol. 35, pp. 27730–27744, 2022.

[2] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[3] T. Degris, M. White, and R. S. Sutton, "Off-policy actor-critic," *arXiv preprint arXiv:1205.4839*, 2012.

[4] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[5] R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," *Advances in neural information processing systems*, vol. 12, 1999.

[6] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *International conference on machine learning*, pp. 1928–1937, PMLR, 2016.

[7] R. S. Sutton, "Learning to predict by the methods of temporal differences," *Machine learning*, vol. 3, pp. 9–44, 1988.

[8] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization," in *International conference on machine learning*, pp. 1889–1897, PMLR, 2015.

[9] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[10] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *International conference on machine learning*, pp. 448–456, pmlr, 2015.

[11] J. L. Ba, J. R. Kiros, and G. E. Hinton, "Layer normalization," *arXiv preprint arXiv:1607.06450*, 2016.

[12] D. Ulyanov, A. Vedaldi, and V. Lempitsky, "Instance normalization: The missing ingredient for fast stylization," *arXiv preprint arXiv:1607.08022*, 2016.

[13] M. Andrychowicz, A. Raichuk, P. Stańczyk, M. Orsini, S. Girgin, R. Marinier, L. Hussenot, M. Geist, O. Pietquin, M. Michalski, *et al.*, "What matters in on-policy reinforcement learning? a large-scale empirical study," *arXiv preprint arXiv:2006.05990*, 2020.

[14] N. Cesa-Bianchi, C. Gentile, G. Lugosi, and G. Neu, “Boltzmann exploration done right,” *Advances in neural information processing systems*, vol. 30, 2017.

[15] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel, “High-dimensional continuous control using generalized advantage estimation,” *arXiv preprint arXiv:1506.02438*, 2015.

[16] S. Gu, T. Lillicrap, Z. Ghahramani, R. E. Turner, and S. Levine, “Q-prop: Sample-efficient policy gradient with an off-policy critic,” *arXiv preprint arXiv:1611.02247*, 2016.

[17] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, *et al.*, “Mastering chess and shogi by self-play with a general reinforcement learning algorithm,” *arXiv preprint arXiv:1712.01815*, 2017.

[18] M. Zinkevich, M. Johanson, M. Bowling, and C. Piccione, “Regret minimization in games with incomplete information,” *Advances in neural information processing systems*, vol. 20, 2007.

[19] O. Vinyals, I. Babuschkin, W. M. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. H. Choi, R. Powell, T. Ewalds, P. Georgiev, *et al.*, “Grandmaster level in starcraft ii using multi-agent reinforcement learning,” *Nature*, vol. 575, no. 7782, pp. 350–354, 2019.

[20] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *arXiv preprint arXiv:1412.6980*, 2014.