UNIVERSITÀ
DEGLI STUDI
DI PADOVA

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA MAGISTRALE IN
Control Systems Engineering

"Deep Reinforcement Learning Approaches for the Game of Briscola"

Relatore: Prof. Gian Antonio Susto

Laureando: Amanpreet Singh

ANNO ACCADEMICO 2022 – 2023

Data di laurea 13/04/2023

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

MASTER THESIS IN CONTROL SYSTEMS ENGINEERING

Deep Reinforcement Learning Approaches for the Game of Briscola

MASTER CANDIDATE

Amanpreet Singh

Student ID 2021438

SUPERVISOR

Prof. Gian Antonio Susto

University of Padova

ACADEMIC YEAR
2022/2023

To my family
and friends

Abstract

Reinforcement learning is increasingly becoming one of the most interesting areas of research in recent years. It is a machine learning approach that aims to design autonomous agents capable of learning from interaction with the environment, similar to how a human does. This peculiarity makes it particularly suitable for sequential decision making problems such as games. Indeed games are a perfect testing ground for reinforcement learning agents, due to a controlled environment, challenging tasks and a clear objective. Recent advances in deep learning allowed reinforcement learning algorithms to exceed human level performance in multiple games, the most notorious example being AlphaGo. In this thesis work we will apply deep reinforcement learning methods to Briscola, one of the most popular card games in Italy. After formalizing the two-player Briscola as a RL problem, we will apply two algorithms: Deep Q-learning and Proximal Policy Optimization. The agents will be trained against a random agent and an agent with predefined moves. The win rate will be used as a performance measure to compare the final results.

Sommario

Il reinforcement learning sta diventando sempre di più una delle aree di ricerca più interessanti negli ultimi anni. Si tratta di un approccio di machine learning che mira alla progettazione di agenti autonomi in grado di apprendere dall'interazione con l'ambiente, in modo simile ad un essere umano. Questa peculiarità lo rende particolarmente adatto a problemi che richiedono decisioni sequenziali, come i giochi. Infatti i giochi sono un terreno di prova perfetto per algoritmi di reinforcement learning, grazie all'ambiente controllato, compiti impegnativi e un obiettivo chiaro. I recenti progressi nel deep learning hanno permesso agli algoritmi di reinforcement learning di superare il livello di prestazione umano in diversi giochi, l'esempio più noto è AlphaGo. In questo lavoro di tesi applicheremo metodi di reinforcement learning alla Briscola, uno dei giochi di carte più popolari in Italia. Dopo aver formalizzato la Briscola a due giocatori come un problema di RL, applicheremo due algoritmi: il Deep Q-learning e il Proximal Policy Optimization. Gli agenti saranno addestrati contro un agente casuale e un agente con mosse predefinite. La percentuale di vittorie sarà usata come misura delle prestazioni per confrontare i risultati finali.

Contents

List of Figures

List of Tables

List of Algorithms

List of Acronyms

RL Reinforcement Learning

MP Markov Process

MRP Markov Reward Process

MDP Markov Decision Process

DP Dynamic Programming

MC Monte Carlo

TD Temporal Difference

DQN Deep Q-Network

DRQN Deep Recurrent Q-Network

PPO Proximal Policy Optimization

1 Introduction

Reinforcement learning (RL) is a machine learning approach to learning from interactions. It has a variety of applications in problems that require sequential decision making. Games are often the ideal domain to test different RL algorithms since they provide a controlled environment with challenging tasks and a clear objective. Recent advances in deep learning allowed reinforcement learning algorithms to exceed human level performance in multiple games. The most famous example of this is the game of Go, considered to be one of the most challenging games for AI. In fact for years most AI approaches were only able to compete against amateur human players. Then the team at DeepMind developed AlphaGo which combined tree search, deep learning and reinforcement leaning. Their algorithm in 2016 was able to win against Lee Sedol (winner of 18 world titles). This was a major breakthrough in the field of artificial intelligence and showed the huge potential of deep reinforcement learning.

1.1 Thesis Outline

The remaining part of this thesis is structured as follows:

- Chapter 2 briefly presents the game of Briscola and its rules.
- Chapter 3 introduces the key concept in reinforcement learning.
- Chapter 4 describes the Deep Q-learning algorithm.
- Chapter 5 covers the Proximal Policy Optimization algorithm.

- Chapter 6 describes the state representation and the reward function used.
- Chapter 7 presents the training setup and the results.
- Chapter 8 concludes the thesis with some considerations about future work.

2

The Game of Briscola

Briscola is one of the most popular card games in Italy. It can have two to six players and can be played either individually or in teams. It is a trick-taking game, meaning that a player leads by throwing down a card and the other players take turns and try to beat it with a higher card in that suit or a card from the same suit of the "briscola". Whoever wins the trick leads the next one. To win the game a player, or a team, have to accumulate more points than the other players, or teams.

2.1 The Cards

The deck used to play the game is a forty card Italian style deck. The cards are divided into four suits. These suits and their corresponding Italian names are shown in Table 2.1 while point value is shown in Table 2.2 where the cards are ranked from low to high. The total number of points in the deck is equal to 120.

Italian name	Suit
Denari	Coins
Spade	Swords
Coppe	Cups
Bastoni	Batons

Table 2.1: Four suits

In Figure 2.1 we show some examples of cards: Three of Coins, King of Swords, Jack of Cups and Ace of Batons. These cards are from the "trevigiane" or "trevisane" deck, and several other regional decks exist.

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

Table 2.2: Point values of the cards

2.2 Rules

Here we describes more in detail how the game is played. At the beginning the deck is shuffled and each player is given three cards. Then another card is taken out from the deck and put face up underneath it. This card establishes which is the trump suit (the highest suit) for the current game. A card from this suit is called *briscola*. The first hand starts with the player on the right of the dealer who plays a card by putting it face up on the table. The suit of this card becomes the lead suit for the current hand. Then, following an anti-clockwise order, all the other players take turns and play their chosen card. Players are not required to play cards in the lead suit. If no briscola was played then the player who played the highest card in the lead suit wins the trick, otherwise the player who played the highest briscola wins. The winner collects all the played cards and puts them in its own pile or the team's pile. Then every player draws one card from the deck starting from the winner and going anti-clockwise. Before the last hand, teammates can look at each other cards once. The game ends when all the cards have been played. After the end the players or the teams calculates their points by summing all the points from the cards in their pile.

Whichever player or team accumulates more points than the others wins.

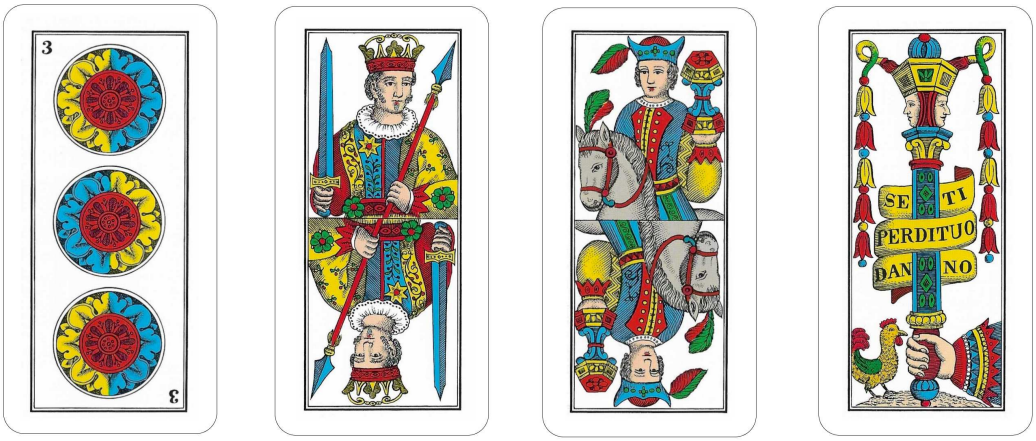

Figure 2.1: Example of cards of the four different suits

3

Introduction to Reinforcement Learning

Reinforcement learning is a computational approach to learning from interactions. It is an interdisciplinary field, situated at an intersection between Artificial Intelligence and Control Theory. It encompasses a variety of concepts drawn from other fields such as Optimization, Neuroscience and Mathematics. It differs from the traditional supervised and unsupervised learning frameworks because data is not available in advance but it is collected by the learner, the agent, by interacting with the environment. This aspect of reinforcement learning makes it particularly suitable for problems that require sequential decision making, such as games. By gathering experience from the environment the agent optimizes its actions over time to solve the specified task and achieve the desired goal. The fundamental concept on which reinforcement learning is based on is the **reward hypothesis**: *"That all of what we mean by goals and purposes can be well thought of as the maximization of the expected value of the cumulative sum of a received scalar signal (called reward)"* [11].
Some of the applications of reinforcement learning are in:

- Games
- Autonomous agents, such as self-driving cars, robots, drones...
- Heat, Ventilating and Air Conditioning (HVAC) energy optimization.
- Recommendation systems and online advertising.

3.1 The Reinforcement Learning Framework

The two entities that interact with each other in a reinforcement learning problem are the **agent** and the **environment**. At each time step t the agent receives a **reward signal** R_t from the environment and its **state** is updated to S_t. The state S_t is an exhaustive description of the system (the agent and the environment) at time step t. Based on the state the agent chooses an **action** A_t according to its policy π. The policy defines the behavior of the agent. It is a mapping from states to actions. The mapping could either be deterministic, in that case it is a function $a = \pi(s)$, or stochastic, i.e. $a \sim \pi(a \mid s)$. The state S_t and the action A_t chosen by agent determine the reward and state at the next time step R_{t+1}, S_{t+1} respectively. This interaction is depicted in Figure 3.1

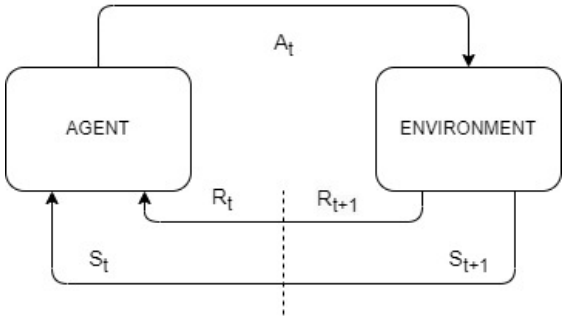

Figure 3.1: Agent-environment interaction

Since past actions influence future rewards, a good policy should not only choose actions that maximize the immediate reward R_{t+1} but also all the future rewards $R_{t+2}, R_{t+3}, R_{t+4}, \ldots$ What we are looking for is to maximize the cumulative reward. For some tasks, called **episodic tasks**, we assume that the interaction between the agent and the environment is divided in episodes, where each episode is independent from another. We define the return G_t as the cumulative reward starting at time step t as follows

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + R_{t+4} + \ldots + R_T$$

Where T is the final time step of that episode, after that the agent is in a terminal state and another independent episode can start. However this is not always the case because some other tasks cannot easily be divided in episodes, these are called **continuing tasks**. A discount factor $\gamma \in [0, 1]$ is introduced in the return to make it mathematically more tractable and also because depending on

the problem in consideration it could be better for the agent to focus more on immediate or long term rewards.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{3.1}$$

If $\gamma = 0$ then the agent only gives value to the present reward ("myopic").
If $\gamma = 1$ then the agent gives the same importance to all rewards, even if they are far in the future ("far-sighted"). If the task is an episodic task, meaning that it terminates after a finite number of steps T, which could be different for each episode, then the sum is truncated. Since the reward at each step is stochastic, the return is also a stochastic quantity. It is possible to decompose the return as the immediate reward plus the discounted return of the successor state as

$$\begin{aligned} G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + ... \\ &= R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + ...) \\ &= R_{t+1} + \gamma G_{t+1} \end{aligned} \tag{3.2}$$

3.2 Finite Markov Decision Processes

Markov decision processes (MDPs) are the mathematical formalism that is used in reinforcement learning. These are an extension of the Markov processes (MPs) and Markov reward processes (MRPs). An MDP is an idealized abstraction of the sequential decision making problem. A Markov process is a memoryless random process, i.e. a sequence of random states $S_1, S_2, S_3...$ with the Markov property. A state S_t is Markov if and only if

$$\mathbb{P}[S_{t+1} \mid S_t] = \mathbb{P}[S_{t+1} \mid S_1, ..., S_t]$$

Formally a Markov process is a tuple $\langle \mathcal{S}, \mathcal{P} \rangle$ where:

- $\mathcal{S}$ is a finite set of states.
- $\mathcal{P}$ is transition probabilities matrix with entries

$$\mathcal{P}_{ss'} = \mathbb{P}[S_{t+1} = s' \mid S_t = s]$$

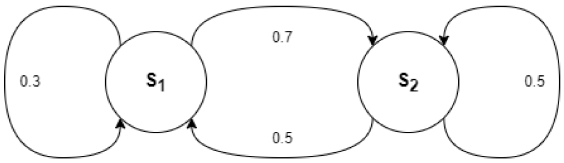

Figure 3.2: Graphical representation of a MP with two states, the arrows are labeled with the transition probabilities.

In a Markov process the agent has no agency since actions and rewards are not present. This is not a complete reinforcement learning problem yet. It is only possible to draw random samples (sequences of states). Adding rewards to a Markov process leads to a Markov reward process.
A Markov reward process is a tuple $\langle \mathcal{S}, \mathcal{P}, \mathcal{R}, \gamma \rangle$ where:

- $\mathcal{S}$ is a finite set of states.
- $\mathcal{P}$ is transition probabilities matrix with entries
$$\mathcal{P}_{ss'} = \mathbb{P}\left[S_{t+1} = s' \mid S_t = s\right]$$
- $\mathcal{R}$ is a reward function.
$$\mathcal{R}_s = \mathbb{E}\left[R_{t+1} \mid S_t = s\right]$$
- $\gamma \in [0, 1]$ is a discount factor.

Now it is possible to define the return of a state s starting at time step t as in Equation 3.1. The return G_t is a stochastic quantity, we define its expectation as the state value function $v(s)$

$$v(s) = \mathbb{E}\left[G_t \mid S_t = s\right]$$

It is possible to estimate $v(s)$ from samples by averaging the return seen from that state. Adding actions to a MRP leads to a MDP.
A (finite) Markov decision process is a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$ such that:

- $\mathcal{S}$ is a finite set of states.
- $\mathcal{A}$ is a finite set of actions.
- $\mathcal{P}$ is transition probabilities matrix with entries
$$\mathcal{P}^a_{ss'} = \mathbb{P}\left[S_{t+1} = s' \mid S_t = s, A_t = a\right]$$
- $\mathcal{R}$ is a reward function.
$$\mathcal{R}^a_s = \mathbb{E}\left[R_{t+1} \mid S_t = s, A_t = a\right]$$
- $\gamma \in [0, 1]$ is a discount factor.

The set of action that an agent can take from a state s is $\mathcal{A}(s)$. For episodic tasks we denote by $\mathcal{S}$ the set of non terminal states and by $\mathcal{S}^+$ the set of all states (including terminal states). In Figure 3.3 there is a graphical representation of an MDP. The white circles are the states while the black circles are the actions. The arrow are labeled with two values separated by a comma. The first value is the probability of transitioning from one state to another by taking that action while the second is the reward obtained.

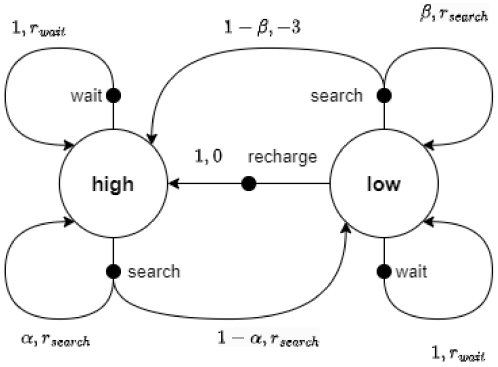

Figure 3.3: Diagram of the MDP of a recycling robot, example from Sutton and Barto.

The policy π defines the behavior of the agent. Formally it is a distribution over actions given the state.

$$\pi(a \mid s) = \mathbb{P}[A_t = a \mid S_t = s]$$

Since the actions are sampled according to the policy π, we consider the following MDP $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}^\pi, \mathcal{R}^\pi, \gamma \rangle$ where

$$\mathcal{P}^\pi_{ss'} = \sum_{a \in \mathcal{A}} \pi(a \mid s) \mathcal{P}^a_{ss'}$$

$$\mathcal{R}^\pi_s = \sum_{a \in \mathcal{A}} \pi(a \mid s) \mathcal{R}^a_s$$

Now we define the state-value function $v_\pi(s)$ and action-value function $q_\pi(s, a)$ for an MDP. The subscript π indicates that the actions are selected according to the policy.

$$v_\pi(s) = \mathbb{E}_\pi[G_t \mid S_t = s] \tag{3.3}$$

$$q_\pi(s, a) = \mathbb{E}_\pi \left[G_t \mid S_t = s, A_t = a \right] \tag{3.4}$$

We can decompose them as done for the return in 3.2

$$v_\pi(s) = \mathbb{E}_\pi \left[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s \right] \tag{3.5}$$

$$q_\pi(s, a) = \mathbb{E}_\pi \left[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) \mid S_t = s, A_t = a \right] \tag{3.6}$$

These two are fundamental quantities in reinforcement learning. For instance if the agent has an estimate of the action-value function, it is possible to derive the following policy (called **greedy policy**): from a state s_t the agent takes the action $a_t = \arg\max_a Q(s_t, a)$, that is the action which maximize the expected return from state s_t. Since the agent can only have an estimate of the action-value function, should it exploit its knowledge and select the greedy action or explore another action that might lead to a higher return? This is known as the **exploration-exploitation dilemma**. One possible trade-off is an **ϵ-greedy policy** defined as follows:

$$a_t = \begin{cases} \arg\max_a Q(s_t, a) & \text{with probability } 1 - \epsilon \\ \text{random action} & \text{with probability } \epsilon \end{cases} \tag{3.7}$$

Where $\epsilon \in [0, 1]$ is the hyper-parameter that manages this trade-off: with $\epsilon = 0$ it becomes a greedy policy while with $\epsilon = 1$ it becomes the random policy that selects all actions with equal probability. Many reinforcement algorithms rely on estimating the action-value function and then derive an implicit policy (e.g. ϵ-greedy policy) or learn the policy directly while maintaining an estimate of the value function.

3.3 Tabular Methods

When the state and action spaces of a problem are small enough it is possible to represent the approximate value function as a lookup table. In this scenario for each state or state action pairs there is an entry corresponding to its estimated value or action-value respectively. There are three main methods to solve finite Markov decision problems. The first one is dynamic programming (DP) which relies on the knowledge of an accurate model of the environment. It works well for small MDPs but it is computationally expensive and the assumption of the knowledge of an accurate model is not often feasible in real problems. On the other hand Monte Carlo methods and temporal-difference learning estimate the value function with only experience. The main idea behind Monte Carlo (MC) methods is to approximate the value function by averaging sample returns from each episode. This can be done both for estimating the value function given the policy (**prediction**) and also for improving the current policy (**control**). The algorithm starts with the policy and the Q function initialized arbitrarily. It runs for a certain number of episodes and the agent acts according to its policy until the end of the episode. Then the return for each state is computed (backwards computation is used for efficiency) and $Q(s, a)$ is updated to average(Returns(s, a)). Finally the policy is updated based on the new Q. There are slight variations of the algorithm based on the specific implementation details (first-visit, every-visit, exploring starts...). Here we show the pseudo-code for the on-policy first-visit MC control for ϵ-soft policies. An ϵ-soft policy is a policy where all actions has probability of selection at least $\epsilon/|\mathcal{A}(S_t)|$ and it is used to ensure enough exploration.

Algorithm 1 On policy first-visit MC control (for ϵ-soft policies), estimates $\pi \approx \pi_*$

Algorithm parameter: small $\epsilon > 0$
Initialize:
$\pi(s) \leftarrow$ an arbitrary ϵ-soft policy
$Q(s, a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
Returns$(s, a) \leftarrow$ empty list, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
for each episode **do**
  Choose $S_0 \in \mathcal{S}, A_0 \in \mathcal{A}(S_0)$ randomly such that all pairs have probability > 0
  Generate an episode from S_0, A_0 following π: $S_0, A_0, R_1, ..., S_{T-1}, A_{T-1}, R_T$
  $G \leftarrow 0$
  for each step of the episode, $t = T-1, T-2, ..., 0$ **do**
    $G \leftarrow \gamma G + R_{t+1}$
    if the pair S_t, A_t does not appear in $S_0, A_0, S_1, A_1, ..., S_{t-1}, A_{t-1}$ **then**
      Append G to Returns(S_t, A_t)
      $Q(S_t, A_t) \leftarrow$ average(Returns(S_t, A_t))
      $A^* \leftarrow \arg\max_a Q(S_t, a)$
      for all $a \in \mathcal{A}(S_t)$ **do**

$$\pi(a \mid S_t) \leftarrow \begin{cases} 1 - \epsilon + \frac{\epsilon}{|\mathcal{A}(S_t)|} & \text{if } a = A^* \\ \frac{\epsilon}{|\mathcal{A}(S_t)|} & \text{if } a \neq A^* \end{cases}$$

      end for
    end if
  end for
end for

The main drawback of this approach is that it needs to wait until the end of the episode to perform the updates. This problem is solved by temporal-difference (TD) learning which combines ideas from MC and DP. TD methods, like MC methods, do not require a model of the environment but learn from raw experience. They are similar to DP since they update estimates based on other estimates without waiting until the end of the episode. This idea of updating estimates based on estimates is called **bootstrapping**. Incremental updates of estimates are often used in reinforcement learning, these updates take the

following form:

$$NewEstimate \leftarrow OldEstimante + StepSize[Target - OldEstimate]$$

In Monte Carlo methods we are estimating value and action-value functions, the algorithms wait until the end of the episode and use the return G_t as the target. On the other hand TD methods need only to wait certain amount of timesteps to perform the update. For example the one-step TD waits until the next step and uses as a target $R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})$. This is the best estimate of the return at the next time step. The update rule is

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)] \tag{3.8}$$

It is called one-step TD or TD(0) because it is a special case of the n-step TD and TD(λ). Two algorithms for TD control are shown. The first, Sarsa, is an **on-policy** algorithm, meaning that it uses experience from the current policy to perform the update.

Algorithm 2 Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\epsilon > 0$
Initialize $Q(s, a) \in \mathbb{R}$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$ arbitrarily except that $Q(terminal, \cdot) = 0$
for each episode **do**
  Initialize S
  Choose A from S using policy derived from Q (e.g. ϵ-greedy)
  while S is not terminal **do**
    Take action A and observe R, S'
    Choose A' from S' using policy derived from Q (e.g. ϵ-greedy)
    $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma Q(S', A') - Q(S, A)]$
    $S \leftarrow S'$
    $A \leftarrow A'$
  end while
end for

The second, Q-learning, is an **off-policy** algorithm, meaning that it directly approximate q_* independent of the policy followed. In fact in Q-learning the estimate is not updated with action-value of the next state and the next

action taken according to the current policy, but the maximum action-value.

Algorithm 3 Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\epsilon > 0$
Initialize $Q(s, a) \in \mathbb{R}$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$ arbitrarily except that $Q(terminal, \cdot) = 0$
for each episode **do**
  Initialize S
  while S is not terminal **do**
    Choose A from S using policy derived from Q (e.g. ϵ-greedy)
    Take action A and observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$
    $S \leftarrow S'$
  end while
end for

Both Monte Carlo and temporal-difference learning methods have their own advantages and disadvantages. Monte Carlo methods are easy to understand, have good convergence properties even with function approximation and the estimates are unbiased, however can have high variance and need to wait until the end of the episode to compute the returns. On the other hand TD methods do not need to wait the end of the episode and are usually more efficient, have low variance estimates but are biased. An intuitive reason why this is the case is that

- the return G_t depends on many transitions (states, actions, rewards)
- the one step TD target depends on one transition (state, action, reward).

There are a class of algorithms that can trade-off the benefits and issues of MC and TD, these are the n-step TD methods. This approach lies in between one-step TD and MC. Instead of just considering one-step TD target it is possible to use n-step return as target:

$$G_{t:t+n} = R_{t+1} + \gamma R_{t+2} + ... + \gamma^{n-1} R_{t+n} + \gamma^n Q_{t+n-1}(S_{t+1}, A_{t+1}) \tag{3.9}$$

It is possible to generalize even further by combining the n-steps returns from all time-steps and obtain TD(λ). We define the λ-return which combines all n-step

returns in a convex combination.

$$G_t^\lambda = (1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}G_{t:t+n} \tag{3.10}$$

If $\lambda = 0$ the TD(0) algorithm is obtained, while if $\lambda = 1$ MC is obtained.

3.4 Function Approximation

Tabular methods do not scale up well with the size of the state and action spaces and for some problems these spaces may even be continuous. Other than the memory required for storing such a large lookup table, it is possible that almost every state will never have been visited before by the agent. In this context it is not possible to find the optimal policy and the objective becomes to find good approximate solutions. Differentiable models such as linear models and neural networks are used as function approximators. In particular what we try to approximate are the value and action-value functions with a parameterized function.

$$\begin{aligned}\hat{v}(s,\boldsymbol{w}) &\approx v_\pi(s)\\ \hat{q}(s,a,\boldsymbol{w}) &\approx q_\pi(s,a)\end{aligned} \tag{3.11}$$

Where $\boldsymbol{w} \in \mathbb{R}^d$ is the parameters or weight vector. Since typically $d << |\mathcal{S}|$, updating one weight will affect the value of multiple states or state-action pairs. This was not the case for tabular methods where updating the value for a particular state would not affect the other states. Since what we want is to model input-output relationships supervised learning techniques are involved when solving reinforcement learning problems with function approximation. However these methods often assume a static training set and i.i.d. samples, which is not the case for RL algorithms since the data generated by following the policy is highly correlated and the target function changes over time as the policy is updated. Furthermore sometimes learning has to be done on-line while the agent is interacting with the environment. Let us consider the approximate value function $\hat{v}(s,\boldsymbol{w})$ and the true value function $v_\pi(s)$. We would like to find the vector of parameters $\boldsymbol{w}$ such that the approximation is good enough. It is possible to do that by minimizing the mean-squared error with stochastic

gradient descent, the update rule is:

$$\begin{aligned} \boldsymbol{w}_{t+1} &= \boldsymbol{w}_t - \frac{1}{2}\alpha\nabla\left[v_\pi(s) - \hat{v}(s, \boldsymbol{w}_t)\right]^2 \\ &= \boldsymbol{w}_t - \frac{1}{2}\alpha\nabla\left[v_\pi(s) - \hat{v}(s, \boldsymbol{w}_t)\right]\nabla\hat{v}(s, \boldsymbol{w}_t) \end{aligned} \tag{3.12}$$

Since the true value function is unknown, a target is used instead. The target, depending on the algorithm, could be the return G_t (MC), the TD(0) target or the λ-return G_t^λ. It is important to notice that divergence and instability problems may arise if function approximation is combined with bootstrapping and off-policy training.

3.5 Reinforcement Learning Algorithms

Reinforcement learning algorithms can be divided two groups: model-based and model-free. The main advantage of model-based reinforcement learning is that the agent can plan-ahead. The main disadvantage is that an accurate model of the environment is not usually available and learning it from experience can be challenging. Model-free algorithms can be classified based on what they are learning. Value-based methods learn an approximate Q-value function and use it to derive an implicit policy (e.g. ϵ-greedy). On the other hand policy-based methods try to learn the policy directly. The policy is represented explicitly and depends on parameters θ. The policy is optimized with gradient ascent to maximize an objective function $J(\theta)$. The main advantage of policy-based methods is that they directly optimize the policy which is the goal of reinforcement learning. Value-based methods, such as Q-learning, are less stable but tends to be more data efficient when they work. Then there are actor-critic methods that lies at the intersection of value-based and policy-based methods and can combine them.

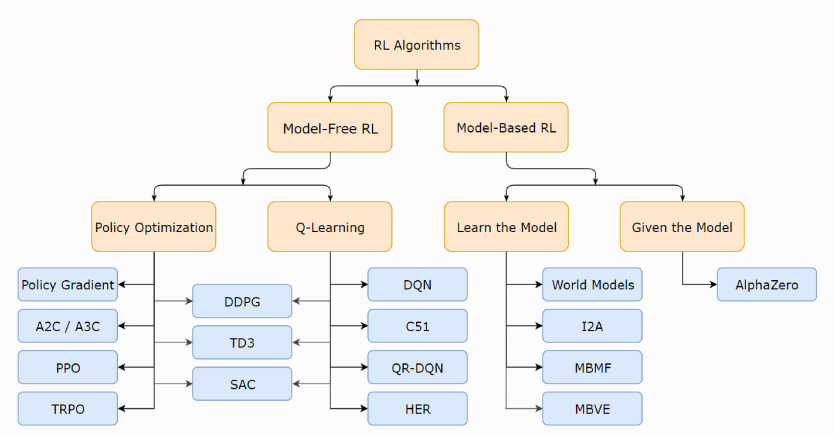

Figure 3.4: RL algorithms, picture from OpenAI Spinning Up website

4 Deep Q-learning

Tabular reinforcement learning algorithms can be combined with function approximation to make them effective for problems with large state and action spaces. Several learning models can be used to achieve this but since the recent advances in deep-learning (artificial) neural networks are often the choice. The **Deep Q-learning** [5] algorithm was introduced in "Playing Atari with Deep Reinforcement Learning" where a variant of Q-learning was combined with a convolutional neural network. The algorithm was updated in a later paper with the addition of several improvement such as a target network. The model achieved impressive results and was able to surpass a human expert on three games.

4.1 The Algorithm

At the beginning of the algorithm the network Q and the target network $\hat{Q}$ are initialized with the same weights. To the tabular Q-learning an experience replay buffer is added, which contains a collection of past experiences of the agent. These are transitions in the form of tuples $(s_t, a_t, r_{t+1}, s_{t+1})$. The agent follows an ϵ-greedy policy and gathers experience and stores the transitions in the replay buffer. Once the buffer contains enough samples a random batch of transitions is sampled and used to update the network Q with a variant of stochastic gradient descent. The target values are computed using the target network $\hat{Q}$ instead of Q, this is done to avoid instability. The presence of the replay buffer should

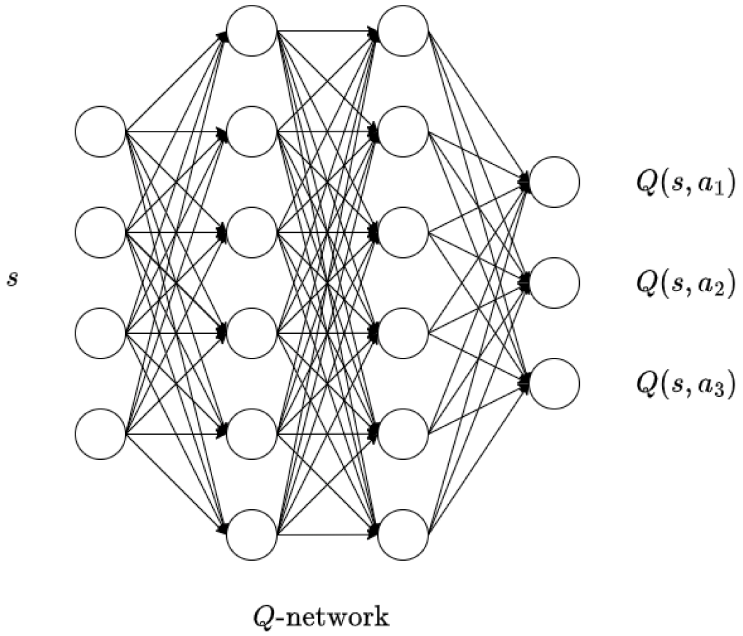

Figure 4.1: Representation of a Q-network

reduce the correlation of the samples which helps learning since deep-learning methods assume independent and identically distributed data, and if absent it could be problematic since sequences of transitions are highly correlated. Also this allows a more efficient use of data since past experiences can be still used to update Q. After a certain number of steps C the weights of the network Q are copied to the target network $\hat{Q}$.
The algorithm pseudo-code is shown on the next page. The stored transitions have an additional term $done$, a boolean, true if the episode is ended and false otherwise. The main limitation of Deep Q-learning with ϵ-greedy policy, other than the fact that it requires discrete action space, is that the learned policy is near-deterministic, which may not be optimal for all kinds of reinforcement learning problems.

Algorithm 4 Deep Q-learning with Experience Replay

Initialize network Q and target network $\hat{Q}$ with random weights.
Initialize replay memory D
while not converged **do**
  $\epsilon \leftarrow$ setting new epsilon with ϵ-decay
  Choose an action a from a state s using policy ϵ-greedy(Q)
  Agent takes action a, observes reward r, and next state s'
  Store transition $(s, a, r, s', done)$ in the experience replay memory D
  if enough experiences in D **then**
    Sample a random minibatch of N transitions from D
    for each transition $(s_i, a_i, r_i, s'_i, done_i)$ in the minibatch **do**
      if $done_i$ **then**
        $y_i = r_i$
      else
        $y_i = r_i + \gamma \max_{a' \in \mathcal{A}} \hat{Q}(s'_i, a')$
      end if
    end for
    Calculate the loss $\mathcal{L} = \frac{1}{N} \sum_{i=0}^{N-1} (Q(s_i, a_i) - y_i)^2$
    Update Q using SGD by minimizing the loss $\mathcal{L}$
    Every C steps copy the weights from Q to $\hat{Q}$
  end if
end while

5

Proximal Policy Optimization

Value-based reinforcement learning algorithms learn the action-value function and derive the policy based on it (e.g. ϵ-greedy). Since the actual objective is to find a good policy, why not learn it directly? This is what policy-based reinforcement learning algorithms try to achieve. These methods have better convergence properties and can learn truly stochastic policies while ϵ-greedy policies are near-deterministic, furthermore they are effective also in continuous action spaces. The policy is parameterized and depends on parameters $\boldsymbol{\theta}$

$$\pi_\theta(a \mid s) = \pi(a \mid s, \boldsymbol{\theta}) = \mathbb{P}(A_t = a \mid S_t = s, \boldsymbol{\theta}_t = \boldsymbol{\theta}) \tag{5.1}$$

We would like to optimize $\boldsymbol{\theta}$ such that good actions have higher probability than bad actions. The parameters are optimized by maximizing a performance measure $J(\boldsymbol{\theta})$. The update rule is

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \widehat{\nabla J(\boldsymbol{\theta}_t)} \tag{5.2}$$

$\widehat{\nabla J(\boldsymbol{\theta}_t)}$ is a stochastic estimate whose expectation approximates the gradient of $J(\boldsymbol{\theta}_t)$ with respect to $\boldsymbol{\theta}_t$. The choice of the performance measure is different whether the task is continuing or episodic.

5.1 Reinforce

Let us consider episodic tasks with discrete action space. A typical parametrization of the policy is the Exponential Softmax Distribution.

$$\pi(a \mid s, \boldsymbol{\theta}) = \frac{e^{h(s,a,\boldsymbol{\theta})}}{\sum_b e^{h(s,b,\boldsymbol{\theta})}} \tag{5.3}$$

$h(s, a, \boldsymbol{\theta})$ represents a parameterized numerical preference, the higher the preference the more often an action is selected. The action preferences can be the output of a linear model or a neural network. Without loss of generality, we can assume that each episode starts in a particular non random state s_0. The performance measure is defined as

$$J(\boldsymbol{\theta}) = v_{\pi_\theta}(s_0) \tag{5.4}$$

The policy gradient theorem for the episodic case states

$$\nabla J(\boldsymbol{\theta}) \propto \sum_s \mu(s) \sum_a q_\pi(s, a) \nabla \pi(a \mid s, \boldsymbol{\theta}) \tag{5.5}$$

$\mu(s)$ denotes the on-policy state distribution under the policy π. It is implicit that the gradient is with respect to $\boldsymbol{\theta}$ and also that the policy depends on $\boldsymbol{\theta}$. The constant of proportionality is the average length of an episode for the episodic case while for the continuing case it is equal to 1.

$$\begin{aligned}
\nabla J(\boldsymbol{\theta}) &\propto \sum_s \mu(s) \sum_a q_\pi(s, a) \nabla \pi(a \mid s, \boldsymbol{\theta}) \\
&= \mathbb{E}_\pi \left[\sum_a q_\pi(S_t, a) \nabla \pi(a \mid S_t, \boldsymbol{\theta}) \right] \qquad \text{(policy } \pi \text{ is followed)} \\
&= \mathbb{E}_\pi \left[\sum_a \pi(a \mid S_t, \boldsymbol{\theta}) q_\pi(S_t, a) \frac{\nabla \pi(a \mid S_t, \boldsymbol{\theta})}{\pi(a \mid S_t, \boldsymbol{\theta})} \right] \\
&= \mathbb{E}_\pi \left[q_\pi(S_t, A_t) \frac{\nabla \pi(A_t \mid S_t, \boldsymbol{\theta})}{\pi(A_t \mid S_t, \boldsymbol{\theta})} \right] \qquad \text{(replacing } a \text{ with } A_t \sim \pi) \\
&= \mathbb{E}_\pi \left[G_t \frac{\nabla \pi(A_t \mid S_t, \boldsymbol{\theta})}{\pi(A_t \mid S_t, \boldsymbol{\theta})} \right] \qquad \text{(since } q_\pi(S_t, A_t) = \mathbb{E}[G_t \mid S_t, A_t]\text{)}
\end{aligned} \tag{5.6}$$

The update rule is

$$\begin{aligned}\boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t + \alpha G_t \frac{\nabla_{\boldsymbol{\theta}} \pi(A_t \mid S_t, \boldsymbol{\theta})}{\pi(A_t \mid S_t, \boldsymbol{\theta})} \\ &= \boldsymbol{\theta}_t + \alpha G_t \nabla_{\boldsymbol{\theta}} \ln \pi(A_t \mid S_t, \boldsymbol{\theta}_t)\end{aligned} \tag{5.7}$$

This is the REINFORCE algorithm, it takes the Monte Carlo approach to estimate the gradient.

Algorithm 5 REINFORCE (episodic), for estimating $\pi_{\boldsymbol{\theta}} \approx \pi_*$

Input: a differentiable policy parameterization $\pi(a \mid s, \boldsymbol{\theta})$
Algorithm parameter: step size $\alpha > 0$
Initialize policy parameters θ
for each episode **do**
  Generate an episode $S_0, A_0, R_1, ..., S_{T-1}, A_{T-1}, R_T$ following π
  for each step of the episode $t = 0, ..., T-1$ **do**
    $G \leftarrow$ return from step t (G_t)
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha G_t \nabla_{\boldsymbol{\theta}} \ln \pi(A_t \mid S_t, \boldsymbol{\theta})$
  end for
end for

Being a Monte Carlo algorithm it suffers from high variance since some episodes could end well with very high returns while other could end with very low returns. To reduce the variance a baseline is subtracted from the return. The baseline $b(S_t)$ should only depend on the state and not on the action. If that is the case the introduction of the baseline results in lower variance. A choice for the baseline is a learned value function.

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha(G_t - b(S_t)) \nabla_{\boldsymbol{\theta}} \ln \pi(A_t \mid S_t, \boldsymbol{\theta}_t) \tag{5.8}$$

5.2 Trust Region Policy Optimization

Policy gradient methods compute the steepest ascent direction for the performance measure $J(\boldsymbol{\theta})$, and take a gradient step to update the policy. Finding a good step size is challenging. If the step is too small learning becomes slow, but if it is too large the updated policy can change too much. If a lot of large "bad" steps are taken the policy can go too far in the parameters space. It could become very hard to recover from these bad updates. The main idea of the Trust Region Policy Optimization (TRPO) algorithm is to take the largest possible step

which improves performance without going too far from the current policy. To achieve this TRPO adds an additional constraint to the problem: the Kullback-Leibler divergence between the updated and current policy should be less than a threshold δ. The KL-divergence is a measure of how different two probabilities distributions are. If the policy is $\pi_{\boldsymbol{\theta}}$ with parameters $\boldsymbol{\theta}$, the theoretical update for TRPO is

$$\begin{aligned} \theta_{k+1} &= \arg\max_{\theta} \mathcal{L}(\theta_k, \theta) \\ &\text{subject to } \overline{D}_{KL}(\theta \mid\mid \theta_k) \leq \delta \end{aligned} \tag{5.9}$$

Where $\overline{D}_{KL}(\theta \mid\mid \theta_k)$ is the average KL-divergence between the new and old policy on the state visited following the old policy. $\mathcal{L}(\theta_k, \theta)$ measures how the the policy π_θ performs relative to the old policy π_{θ_k}

$$\mathcal{L}(\theta_k, \theta) = \mathbb{E}_{s,a\sim\pi_{\theta_k}} \left[\frac{\pi_\theta(a \mid s)}{\pi_{\theta_k}(a \mid s)} A^{\pi_{\theta_k}}(s, a) \right] \tag{5.10}$$

$A^\pi(s, a)$ is the advantage function, it measures how good is to take action a relative to all other actions from state s.

$$A^\pi(s, a) = q_\pi(s, a) - v_\pi(s) \tag{5.11}$$

The advantage has to be estimated, for example using generalized advantage estimation (GAE) [6]. This is the theory behind TRPO. The actual implementation of the algorithm makes approximations which we will not discuss.

5.3 PPO with Clipped Objective

The Proximal Policy Optimization (PPO) [7] algorithm tries to solve the same problem as TRPO with a different approach. Instead of adding an optimization constraint like TRPO, the PPO-Clip maximize a surrogate objective function.

$$L(s, a, \theta_k, \theta) = \min \left(\frac{\pi_\theta(a \mid s)}{\pi_{\theta_k}(a \mid s)} A^{\pi_{\theta_k}}(s, a), \text{clip} \left(\frac{\pi_\theta(a \mid s)}{\pi_{\theta_k}(a \mid s)}, 1 - \epsilon, 1 + \epsilon \right) A^{\pi_{\theta_k}}(s, a) \right)$$

A simplified version is

$$L(s, a, \theta_k, \theta) = \min\left(\frac{\pi_\theta(a \mid s)}{\pi_{\theta_k}(a \mid s)} A^{\pi_{\theta_k}}(s, a), g(\epsilon, A^{\pi_{\theta_k}}(s, a))\right) \tag{5.12}$$

where

$$g(\epsilon, A) = \begin{cases} (1+\epsilon)A & A \geq 0 \\ (1-\epsilon)A & A < 0 \end{cases}$$

If the ratio $\pi_\theta(a \mid s)/\pi_{\theta_k}(a \mid s)$ is > 1 then the action a becomes more likely in the updated policy with respect to the old policy. If the ratio is between 0 and 1 then the action becomes less likely. If the advantage of state-action pair is positive the algorithm should increase the probability of taking that action. On the other hand if the advantage is negative the algorithm should decrease the probability of taking that action. However the presence of the clipping and the $\min(\cdot, \cdot)$ ensures that these updates do not change the policy dramatically. The clipping parameter ϵ controls how conservative the algorithm is in the updates.

Algorithm 6 PPO-Clip

1: Input: intial policy parameters θ_0, initial value function parameters ϕ_0
2: **for** $k = 0, 1, 2, \ldots$ **do**
3: Collect a set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running the policy $\pi_k = \pi(\theta_k)$ in the environment.
4: Compute rewards-to-go $\hat{R}_t$.
5: Compute advantage estimates, $\hat{A}_t$ (Using any method of advantage estimation) based on the current value function V_{ϕ_k}.
6: Update the policy by maximizing the PPO-Clip objective:

$$\theta_{k+1} = \arg\max_\theta \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min\left(\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_k}(a_t \mid s_t)} A^{\pi_{\theta_k}}(s_t, a_t), g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t))\right)$$

typically via stochastic gradient ascent with Adam.
7: Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_\phi \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t\right)^2$$

typically via some gradient descent algorithm.
8: **end for**

Since the policy is stochastic it should be able to achieve a good level of exploration by itself. Usually during the training process the policy progressively becomes less random since some actions start to be preferred over others. The level of "randomness" of the policy is measured by its entropy. The entropy starts high and decreases over time until it reaches a stable value. If this process happens too quickly then the agent has not done enough exploration. To avoid this an entropy bonus can be added to the algorithm.

6

Problem Formalization

In this chapter we will formalize the two-player Briscola game as a reinforcement learning problem. Our primary objective is to find an appropriate representation of the state and a suitable reward function. This is a very important step since the reward function determines the goal of the agent and ultimately its behavior. A state that is a good description of the agent-environment system is also necessary to learn an effective policy.

6.1 State of the Agent

To derive the state we can start by considering what a human player "sees" and "thinks" during the game. At each step of the game the human player sees its own cards, the played card on the table, if there is any, and knows which card is the briscola. It is reasonable to think that this information should be contained also in the state of the agent. The human player also knows how many points he has so far and has an idea of what cards have already been played and cannot be in the remaining deck or in the opponent's hand. The human player knows at which step the game is, and depending on how many hands are left, can play more or less aggressively. To summarize the agent state must contain information about

- its own cards
- the played card on the table, if there is any
- which is the briscola card (or just the briscola suit) for that episode

- how many points has the agent so far
- at which step the game is

And for each card (its own or played on the table)

- the name of the card (whether it is Two, King, Ace...)
- the suit of the card (whether it is Coins, Swords, Cups or Batons)

The choice of the state vector embedding this information is not unique. Let us start by considering a single card. We can label the card names with ten numbers from 0 to 9 and label the four seeds with number from 0 to 3 or use a one-hot encoding. One possible labeling could be

name	label
Ace	0
Two	1
Three	2
Four	3
Five	4
Six	5
Seven	6
Jack	7
Knight	8
King	9

Table 6.1: An example of labeling

We can then use a vector of six features to represent a single card:

- a number from 0 to 9 (the name of the card)
- a boolean: 1 if is a briscola, 0 otherwise
- one-hot encoding of the suit

suit	one-hot encoding
Batons	$[1, 0, 0, 0]$
Cups	$[0, 1, 0, 0]$
Coins	$[0, 0, 1, 0]$
Swords	$[0, 0, 0, 1]$

Table 6.2: One-hot encoding of the four suits

Following the labeling and encoding of Table 6.1 and Table 6.2, the vector encoding the Three of Coins is:

- $[2, 0, 0, 0, 1, 0]$ if the briscola suit is different from Coins.
- $[2, 1, 0, 0, 1, 0]$ if the briscola suit is Coins.

The state vector has dimension 26 and is obtained as follows:

- an integer for the points of the agent so far
- number of hands played so far (called "steps" in Figure 6.1)
- four stacked vectors representing cards (the first three are its own and the fourth is the one played on the table, if there is any)

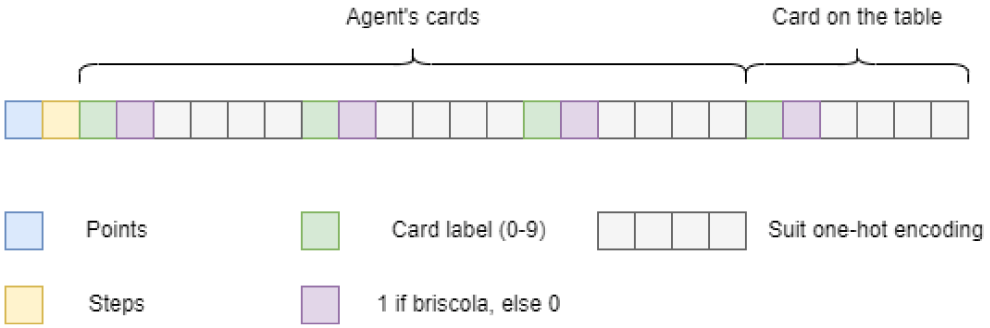

Figure 6.1: State vector

We can assume that the agent's cards are ordered such that there is a first, a second and a third card. Once a card is played, the card drawn from the deck does not take the place of the played card but is added at the end. To give an example suppose the first card is played, then the second and third cards become the first and the second card respectively and the newly drawn card becomes the third card. If there is not a played card on the table or if the agent has less than three cards the corresponding entries in the state are set to zero, since $[0, 0, 0, 0, 0, 0]$ does not represent any card. Let us denote this state representation as *state-1*. A human player remembers during the game whether or not a card has already been played. We can augment the state-1 with such information by appending to it a vector of length 40 where each entry is associated with one card in the deck. At the beginning of the episode all entries are set to 0, and at each step the agent observes its own cards and the

card one the table and sets the corresponding entries to 1. Now the state vector has dimension 66 (26 + 40). Let us call this state representation *state-2*.

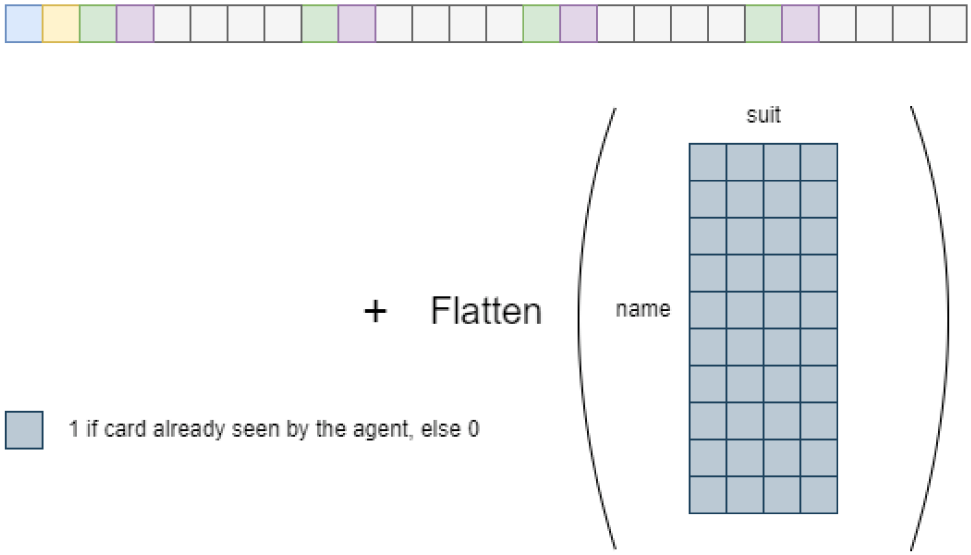

Figure 6.2: State-2

It is important to note that the state space for state-2 is much larger than the state space for state-1. This is due to the fact that there is more than one possible set of played cards, hence multiple vectors to append to state-1 to build state-2. Therefore although state-2 could lead to a better policy because it has more information, exploring such a large state space is more challenging than for state-1.

6.2 Actions

The action space is discrete. At each step of the episode the agent has to decide which card to play. The possible actions are $\{0, 1, 2\}$ whether the agent chooses to play the first, the second or the third card. Not all actions are feasible in every state, because in the last two hands of the episode the agent has less than three cards to play.

6.3 Reward Function

There are several different ways that we can design the reward function depending on what is the desired agent's behavior. One approach could be to give a reward only at the end of the episode: a positive reward of +1 for winning and a negative reward of −1 for losing. This is a reasonable starting point and the policy will be optimized to win. The main problem of this kind of sparse rewards is that if winning is rare the training becomes very slow. This could be especially problematic for on-policy algorithms. We can try to design a more dense reward function giving a +1 reward at each step if the agent wins the hand or −1 if the agent loses the hand. This could lead to a policy where the agent tries to win the most amount of hands as possible, no matter how many points he wins. The learned policy is not ideal. A better approach would be to to give a reward equal to the points, positive if the agent wins the points, negative otherwise. This is a good choice and the agent will try to maximize its points in each episode. Sometimes the agent will lose a hand for potentially more points in the future, even if winning the present hand means winning the game. To mitigate this problem instead of giving the reward for winning at the end of the episode we could give a positive reward of $r_w = 100$ if winning those points will lead to win the game or give a $-r_w$ if losing those points will make the opponent win the game. To summarize the final reward function works as follows:

- at each step the reward $r = points$ if the agent wins the hand, $r = -points$ otherwise.
- additional reward $r_w = 100$ is added or subtracted to r depending on whether winning or losing those points leads the agent or the opponent to win the game.

7 Experiments and Results

In this chapter we will present the training setup and comment the results of the experiments. The agents are trained against a random agent and an "intelligent" agent with predefined moves. Periodically during training the agent performance is evaluated by simulating a thousand games and its win rate is saved. A text-based environment with the random and rules based agents was available along with a TensorFlow implementation of the DQN and DRQN agents. However, these two were reimplemented in PyTorch along with the PPO agent. The implemented agents were tested on well known OpenAI gym environments.

7.1 Random Agent and Rules Based Agent

The random agent at each step of the episode chooses a random action with equal probability. Since this agent does not have a real strategy a simple policy should be able to win against it most of the time. This is not the case for the "intelligent" agent who has hard coded moves or rules (in the form of if-else statements) that it follows depending on the situation. We will now give a brief overview of the general strategy followed by the rules agent without going into all the details. The first thing that the rules agent looks is whether or not there are points to win. If there are no points to win or no cards have been played yet, the agent tries to play the weakest non briscola card. The reason is that if there is no card the agent does not know what the opponent is going play, therefore it cannot play a card with many points and risking losing them. On the other

hand if there are no points to win it does not make sense to play a strong card or a briscola since they can be useful to win more points in future hands. In the scenario where the agent can win some points there are several different possibilities. If winning the hand leads to winning the game then the agent does that, otherwise the best action depends on the amount of points and whether or not it is worth to play the winning card or to use it later. As it is possible to see the agent tries to minimize the points that it can loose while maximizing the points it can win. After 10000 simulated games the rules agent wins against the random agent 79.47% of the time. Designing this kind of agents is far from easy since it requires good knowledge of the game and for some problems it may be very challenging to analyze all the possible outcomes of a decision due to the complexity of the environment.

7.2 Deep Q-Network Agent

The results of training are shown in Figure 7.1. In order to see which state representation, state-1 or state-2, suits better for the task two versions of the DQN agent has been trained. The agents have been trained for 25000 episodes against the random and the rules agent. At each step ϵ is decayed exponentially from a starting value of 1.0. Every thousand episodes 1000 games are simulated and the agents' performance is evaluated and the win rate is saved. Only during evaluation the agents act greedily, i.e. ϵ is set to 0.0 and then restored to its value before the evaluation. The Q network is a fully connected neural network with two hidden layers having 256 neurons each. The network is optimized with the RMSProp optimizer since it was the choice in the original paper but the Huber loss is used instead of the Mean Squared Error loss because of its robustness to outliers. The remaining hyperparameters and their values are shown in Table 7.1.

Hyperparameter	Value
learning rate (α)	10^{-4}
discount factor (γ)	0.95
epsilon decay rate	0.999998
minibatch size	256
target Q update interval	1000 (episodes)
replay buffer size	10^6

Table 7.1: DQN training hyperparameters

Both representations allow the agent to learn an effective policy and achieve a good win rate. However as it is possible to see from the plot, learning is faster for state-1 and it achieves a higher win rate against both the random and rules agent, about 87% and 59% respectively. This is somewhat expected since the state space for state-2 is larger it requires higher degree of exploration than state-1. The difference in performance is not drastic since agent with state-2 is still able to achieve a win rate of 80% against the random agent and 50% against the rules agent.

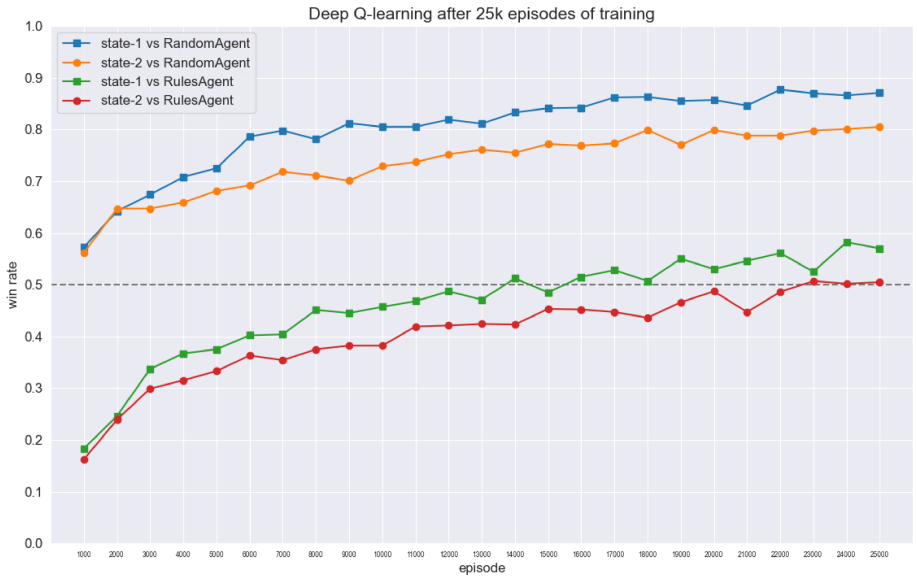

Figure 7.1: DQN results

With additional training and more exploration state-2 should be able to reach or

even outperform state-1. Certainly it is useful to know when a card has already been played and cannot be in the opponent's hand but it is not easy to estimate the amount of advantage that the augmented state is providing to the agent since this additional information may be useful more in multiplayer games or towards the end of the episode. In those cases using state-2 might give a slight edge over state-1. However at this point it is likely that one of the two players would have already accumulated enough points to win the game, therefore the difference in the final win rate might not be that significant. Because of the faster training and better performance state-1 has been used for training the other agents and for comparison in Table 7.4.

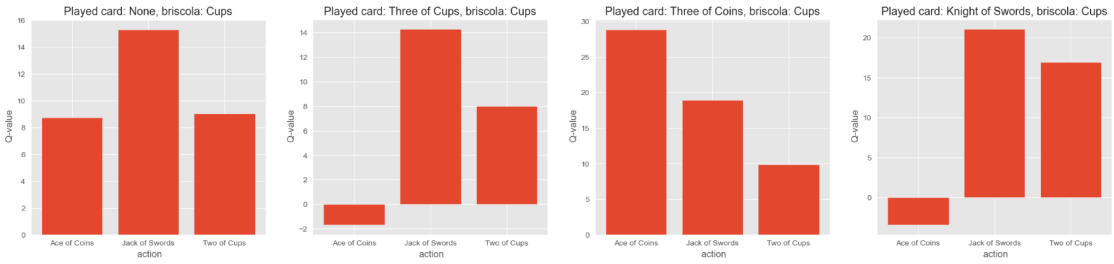

Figure 7.2: Learned Q-values

The learned Q-values are shown in Figure 7.2 for different scenarios. This situations refers to the beginning of the game when the agent has zero points. In the first case when there is no played card the best action according to the learned policy is to play the Jack since playing Ace may lead to loose too many points while the briscola could be more useful in future hands. The same is true for the second scenario where the Q-value for the first action is even lower because the opponent will get a total of 21 points if Ace is played, while in the third scenario the situation is the opposite. In the fourth the agent looses on purpose instead of choosing the briscola which is reserved for a later hand.

7.3 Deep Recurrent Q-Network Agent

The Deep Recurrent Q-learning [4] algorithm chooses an action based not only on the current state but also the past history. This is achieved through a recurrent neural network which maintains an hidden state.

Hyperparameter	Value
learning rate (α)	10^{-4}
momentum	0.99
discount factor (γ)	0.95
epsilon decay rate	0.999998
minibatch size	64
target Q update interval	500 (episodes)
replay buffer size	10^6
sequence length	4

Table 7.2: DRQN training hyperparameters

At the beginning of each episode the hidden state is reset. At each step the output and the hidden state of the network is computed based on its input (the agent's state) and the hidden state at the previous step. The DRQN agent is trained for 25000 episodes, as the DQN agent, using state-1 representation on sequences of length 4. The Q network used has a fully connected layer and an LSTM layer, both of size 128. The network is optimized with the RMSProp optimizer and the Huber loss is used. The remaining training hyperparameters are shown in Table 7.2.

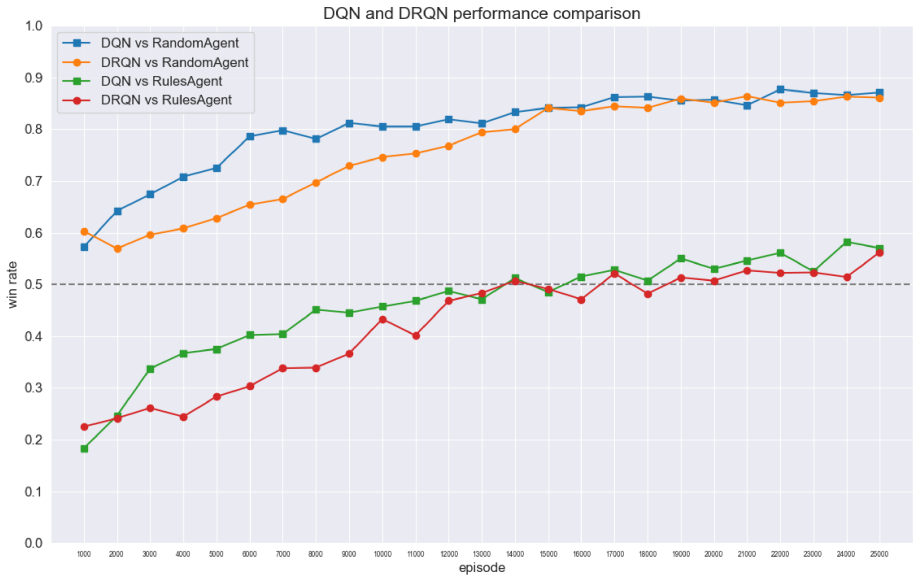

Figure 7.3: DRQN results

As it is possible to see from Figure 7.3 although training is slower for the recurrent agent eventually it reaches the same win rate as the non-recurrent agent.

7.4 PPO Agent

The separate networks version of the PPO algorithm has been used. Given the state vector the policy network, or the actor, outputs the actions probabilities while the value network, or the critic, outputs an estimate of the state's value. The two networks have the same architecture: two fully connected layers having 128 neurons each. The experience is collected (on-policy) for 64 episodes then the policy and value function parameters are updated separately for PPO steps epochs. The Adam optimizer is used with the same step-size α for each network, and an entropy bonus is added to ensure exploration.

Hyperparameter	Value
learning rate (α)	10^{-3}
discount factor (γ)	0.90
GAE parameter (λ)	1.0
entropy coefficient	0.03
PPO clip factor (ϵ)	0.2
PPO steps	10

Table 7.3: PPO training hyperparameters

The results of training are show in Figure 7.4, a win rate of 76% against the random agent and of 44% against the rules agent has been achieved by the agent after 250k episodes. This approach required more training than DQN and DRQN and was not able to surpass the 50% win rate threshold against the rules agent.

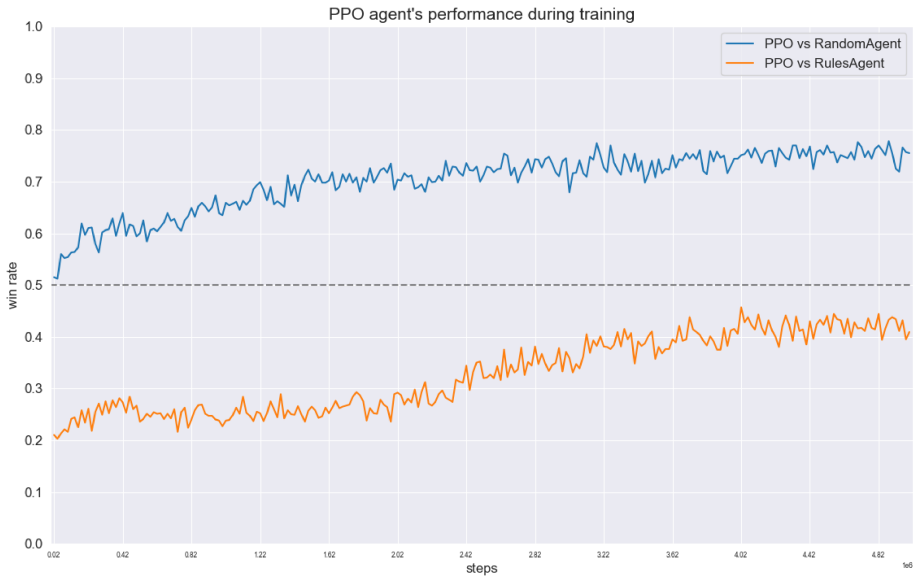

Figure 7.4: PPO results

7.5 Comparison

To do a more accurate comparison of the trained agents, 10000 games where simulated. During the evaluation DQN and DRQN acted greedy while PPO still followed its stochastic policy. The results of this simulated games are shown in Table 7.4 where for instance "DQN (Random)" is the DQN agent trained against the random agent.

	Random Agent	Rules Agent
Rules Agent	79.47%	-
DQN (Random)	87.01%	37.93%
DQN (Rules)	76.29%	57.01%
DRQN (Random)	86.09%	33.94%
DRQN (Rules)	79.75%	54.60%
PPO (Random)	75.69%	27.92%
PPO (Rules)	67.92%	42.11%

Table 7.4: Win rates comparison

The best results against the random agent are obtained by DQN (Random),

outperforming the rules agent by 8%. The DQN (Rules) only gets to 76%, this seems reasonable since the agent interacted with a different environment and could not have learned to exploit the vulnerabilities of the random agent as well as the algorithm trained directly against it. DQN also obtains the highest win rate against the rules agent followed by DRQN. The PPO agent performed worse compared to both DQN and DRQN. There could be several reasons why this is the case. One possibility is that the presence of a replay buffer allows the Q-learning based agents to learn efficiently from past experience. Another possibility is that the PPO hyperparameters used are sub-optimal for this specific environment.

	Random Agent	Rules Agent
DQN (Random)	90.13%	40.39%
DQN (Rules)	85.00%	64.49%

Table 7.5: Win rates after 100k episodes of training (on 10k games)

Since DQN gave the best results it was further trained for another 75k episodes for a total of 100k episodes with a minimum ϵ set to 0.1. As it is possible to see from Figure 7.5 the additional training did not improved significantly the results against the random agent since the win rate plateaued at 90%. On the other hand against the rules agent DQN was able to touch a 67% win rate during training, resulting in a 10% increase. The results of 10000 simulated games are shown in Table 7.5

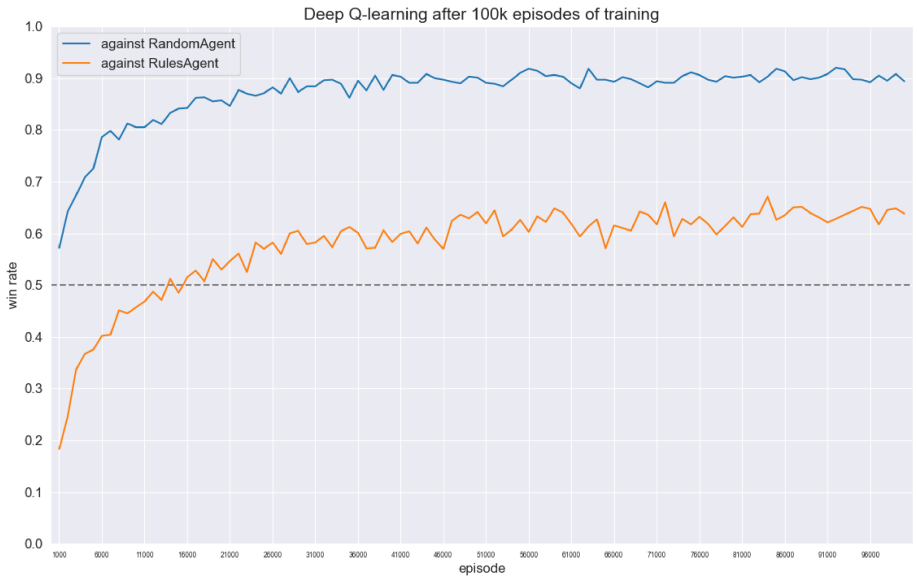

Figure 7.5: DQN after further training

8

Conclusions and Future Works

In this thesis we investigated the effectiveness of model-free reinforcement learning approaches for the game of Briscola. After an introduction to the game and the main concepts in reinforcement learning, the algorithms used for training were described: Deep Q-learning and Proximal Policy Iteration. Next we formalized the two-player Briscola as a reinforcement learning problem. This meant defining a state and reward function. In particular, we derived two possible representations, state-1 and state-2. While both were effective in learning a good policy over time, state-1 was faster and achieved a higher win rate when tested with DQN. Because of this, state-1 was used for training also DRQN and PPO agents. To make a more accurate comparison all the agents were evaluated after training on 10k simulated games in which the DQN and the DRQN agents acted greedily while the PPO agent followed its stochastic policy. The results showed that Deep Q-learning appears to be more effective for this task since it obtains better results against both the random and rules based agent. DRQN achieved similar results while PPO showed slower progress and reached a lower win rates. Since hand-crafted features were used to to build the state it could be interesting to test the algorithms with features learned by a convolutional neural network on a GUI version of the environment. A future challenge could be to derive a better stochastic policy by optimizing the PPO hyperparameters or applying a different algorithm. The next step could be to extend this approach to multiplayer Briscola. It would be interesting to see in a teams based setting how the agents cooperate to achieve a common goal.

References

[1] Marcin Andrychowicz et al. *What Matters In On-Policy Reinforcement Learning? A Large-Scale Empirical Study*. 2020. arXiv: 2006.05990 [cs.LG].

[2] William Fedus et al. *Revisiting Fundamentals of Experience Replay*. 2020. arXiv: 2007.06700 [cs.LG].

[3] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[4] Matthew Hausknecht and Peter Stone. *Deep Recurrent Q-Learning for Partially Observable MDPs*. 2017. arXiv: 1507.06527 [cs.LG].

[5] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning*. 2013. arXiv: 1312.5602 [cs.LG].

[6] John Schulman et al. *High-Dimensional Continuous Control Using Generalized Advantage Estimation*. 2018. arXiv: 1506.02438 [cs.LG].

[7] John Schulman et al. *Proximal Policy Optimization Algorithms*. 2017. arXiv: 1707.06347 [cs.LG].

[8] John Schulman et al. *Trust Region Policy Optimization*. 2017. arXiv: 1502.05477 [cs.LG].

[9] David Silver. *Lectures on Reinforcement Learning*. https://www.davidsilver.uk/teaching/. 2015.

[10] *Spinning Up OpenAI website*. https://spinningup.openai.com/.

[11] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

Acknowledgments

I would first like to thank my supervisor, Prof. Gian Antonio Susto, for his availability and for helping me throughout the duration of this thesis by providing constant feedback.

I would also like to thank Alessio Arcudi for providing interesting suggestions for me to test.

Special thanks to Michelangelo Conserva and Alberto Soragna, developers of the Briscola environment that was the starting point for this project.