Università di Trieste

Dipartimento di Matematica e Geoscienze

Laurea Magistrale in Data Science
and Scientific Computing

Briscola Bot
Mastering Briscola with model-free Deep Reinforcement Learning

Lorenzo Cavuoti

Relatore:

Prof: Antonio Celani

Correlatore:

Emanuele Panizon

Candidato:

Lorenzo Cavuoti

ANNO ACCADEMICO 2021/2022

ABSTRACT

Recent developments in reinforcement learning demonstrate that it can be applied in a variety of domains, including card games. In this thesis, we introduce BriscolaBot, a reinforcement learning agent that learns to play the Italian card game of Briscola in a one-on-one setting. Utilizing the Proximal Policy Optimization (PPO) algorithm and training a deep neural network to learn the game's policy. Our study marks the first attempt to evaluate the performance of a reinforcement learning agent in the game of Briscola against human players. In particular, our latest agent, BriscolaBot-v3, was able to achieve a performance similar to that of an average human player. We also analyze the impact of various reward structures on the agent's learning process and identify potential areas for future improvement. To showcase the effectiveness of our agent, we provide a web-based version of the game where users can play against the trained agent, which can be accessed at `https://replit.com/@LorenzoCavuoti/BriscolaBot`. The code for the agent and the web-based game can be found in the GitHub repository `https://github.com/LetteraUnica/BriscolaBot`.

Ringraziamenti

Questo progetto rappresenta la conclusione del mio percorso di studi, tuttavia, sarebbe sbagliato dire che ho fatto tutto da solo. In particolare, ringrazio la mia famiglia che mi ha supportato durante tutti questi anni non facendomi mancare nulla. Mi avete dato la forza di proseguire.
Ringrazio i miei amici, assieme a voi è sempre stato divertente passare le serate insieme.
Ringrazio i miei compagni di corso, anche se per vari motivi non ci siamo potuti vedere di persona, studiare insieme per gli esami è sempre stato piacevole.
Ringrazio il mio relatore Antonio Celani e correlatore Emanuele Panizon che hanno accolto questo progetto con entusiasmo e sono stati prontamente disponibili quando ne ho avuto bisogno.

Contents

Chapter 1

Introduction

Briscola is one of Italy's most popular card games and variations of it are also played in Portugal, Slovenia, Croatia and Montenegro [1]. It can be played from two to six players. Briscola presents unique challenges for reinforcement learning due to its hidden information aspect. Unlike perfect information games like checkers, chess, and Go, established techniques cannot be easily applied to Briscola. Reinforcement learning in Briscola requires finding the best move under incomplete information about the game state, forcing the agent to make educated guesses about the cards held by other players.

In this thesis, we propose a new algorithm to learning to play Briscola in a one-on-one setting based on policy gradient methods with function approximation. Specifically, we adopt PPO with self-play, following the approach of OpenAI on Dota 2 [2].
We have chosen a policy gradient method due to its ability to learn any strategy, rather than being restricted to a deterministic one. This is crucial in learning to play Briscola, an imperfect information game, as a deterministic strategy will not necessarily result in the optimal play. To better understand this point, consider the game of poker. In a given situation, the best move is not always to check or to fold, but rather mix between the two, in order to prevent the opponent from being able to determine what cards the agent is holding. In other words, the value of an action depends on the probability it is played.

Our approach was able to learn how to play Briscola in a one-on-one setting in using only a standard 4-core CPU, making it computationally efficient and easily replicable.
A web-based version of the game where the reader can play against the trained agent is available at `https://replit.com/@LorenzoCavuoti/BriscolaBot`.

1.1 Game Theory

Game theory is a branch of mathematics that focuses on the study of strategies for handling situations where the result of a player's action depends on the choices made by other participants.

Games like chess and Go are referred to as perfect information, zero-sum games. Perfect information means that all players have access to all information about the game state, while zero-sum means that what benefits one player harms the other, with no possibility for a win-win outcome. There are various techniques for finding or approximating the optimal strategy[1] in such games, like minimax search and Monte Carlo tree search (MCTS).

For instance, the Stockfish chess engine, to determine the best move, utilizes a variant of minimax search called alpha-beta pruning with a heuristic evaluation function [3]. Meanwhile, the Leela Chess Zero chess engine uses MCTS guided by a neural network [4].

In contrast, games like Briscola, poker, StarCraft II, Dota 2, and Kriegspiel are imperfect information, zero-sum games. Imperfect information means that the game state is not fully observable, forcing agents to make inferences based on incomplete data. As a result, the optimal play strategy is not always deterministic, as a deterministic agent can be predictable and exploited by the opponent. When choosing the best move, the agent must also take into account the information gained and the information they might reveal to the opponent [5].

Imperfect information games can still be solved by finding or approximating a Nash equilibrium. In a Nash equilibrium, no player has an incentive to change their strategy, as this will lead to a worse outcome for them. John Nash demonstrated in 1952 that every game with a finite number of players and actions has at least one Nash equilibrium [6].

1.2 Related work

Existing approaches to learning Briscola can be broadly classified into three categories:

1. **Rule based agents:** These agents use pre-determined, human-created rules to play the game. However, they lack the ability to adapt to new situations, as they are unable to learn from experience. The website `http://www.solitariconlecarte.it/briscola_2classica.htm` provides a rule based agent that plays Briscola.

[1] The optimal strategy is the strategy that gives the best win chance, assuming that the opponent plays perfectly.

2. **Model-free agents:** These agents use model-free reinforcement learning techniques to learn the optimal strategy. They learn from experience through self-play or playing against other agents, such as a rule-based agent. At the time of writing, there are two known projects that use Deep Q-Learning (DQL) [7] to learn Briscola. The first one [8] trains the DQL agent against a random agent, while the second [9] uses self-play to reach 90% win rate against a random agent.

3. **Search-based agents:** These agents employ Monte Carlo tree search (MCTS) to explore the game tree and find the best move. This approach has been successful in perfect information games like chess [10], Go [11] and backgammon [12], as it requires a model of the environment. In the case of imperfect information games like Briscola, this model is not available as the strategies of other players are unknown. To overcome this, some approaches assume the other player will randomly play a card from the set of unseen cards [13]. A more flexible method could involve learning a model of the other player to predicts their moves, allowing the agent to adapt to their strategy.

These approaches have also been combined, although not in Briscola. For instance, one of the earliest versions of AlphaGo [14] learned a Go strategy by imitating professional human players, then refined the learned strategy using model-free reinforcement learning techniques. Lastly, it utilized MCTS to further enhance its performance during actual gameplay.

While developing our algorithm, we also looked at other card games such as poker, which has received significant attention over the years due to its popularity. In 2017, Libratus [15] became the first program to defeat the world's best poker players by using the Counterfactual Regret Minimization (CFR) [16] algorithm. CFR is an iterative method that converges to a Nash equilibrium in two-player zero-sum games like poker or Briscola. Recently, in 2020, ReBeL [17] became the first program to beat the world's best poker players in full-table, not just one-on-one, no-limit Texas hold'em. ReBeL used a similar approach to CFR, but with the addition of a deep neural network to guide the algorithm, reducing the computational time required to reach the Nash equilibrium.

1.3 Contributions

The key contributions of this thesis are:

- Presenting a novel algorithm for learning to play Briscola in a one-on-one setting using policy gradient methods with function approximation.

- Demonstrating that the algorithm achieves human-level play when trained on a standard 4-core computer, highlighting both its effectiveness and computational efficiency.
- Exploring the impact of the reward structure on the agent's performance and examining the influence of additional hyperparameters.
- Developing a web-based version of the game where users can play Briscola against the trained agent, available at `https://replit.com/@LorenzoCavuoti/BriscolaBot`.

1.4 Outline

The rest of the thesis is organized as follows:
Chapter 2 provides a theoretical introduction to Reinforcement Learning (RL), its relationship to Markov Decision Processes (MDPs), and the fundamental concepts and elements involved. It defines rewards, policies, and value functions, and delves into the algorithmic techniques used to solve RL problems such as Dynamic Programming, Monte Carlo methods, Temporal-Difference Learning, and n-step methods. The chapter concludes by exploring the challenges posed by Multi-Agent Reinforcement Learning. The chapter concludes by introducing the problem of Multi-Agent Reinforcement Learning and explores its challenges.
Chapter 3 describes policy gradient methods with function approximation. The chapter begins with an overview of function approximation techniques, in particular neural networks, followed by the presentation of the policy gradient theorem - a key result in reinforcement learning. Next, policy gradient algorithms such as REINFORCE and its baseline version are discussed in detail. Lastly, the chapter delves into actor-critic methods such as TRPO and PPO, with a focus on the latter which has been utilized in major publications, ranging from natural language [18] to the game of Dota 2 [2].
Chapter 4 presents the core contribution of this thesis. It details the design and implementation of the agent and the environment in which it operates, like the definition of the reward structure and observation space. The chapter also describes thoroughly the training procedure that we followed, including the choice of hyperparameters and their impact on the agent's performance. At the end of the chapter, we evaluate the last version of the agent, BriscolaBot-v3, against human players, showcasing the effectiveness of the proposed solution to the problem.
Chapter 5 summarizes the key results of the presented approach, highlighting the major difficulties posed by the problem, and offers suggestions for future improvements.

Chapter 2

Background

When a human first plays a game, let it be chess, poker, Minecraft, CS:GO or any other game, it is not given any instructions on how to play. It is not told which moves are good and which are bad, maybe it doesn't even know the rules of the game. By interacting and playing the game, losing to some opponents and starting to beat others, the player gradually learns what moves are best in which situation and starts to understand the dynamics of the game. During all of this the player doesn't have any external feedback telling him which actions to perform in a given situation, but has to learn it by itself, by performing an action and experiencing the outcome. Learning from interaction is a foundational idea underlying nearly all theories of learning and intelligence [19].

In this chapter we provide an overview of reinforcement learning. First we will discuss Markov Decision Processes (MDPs) that allow us to define an environment (for now we can think of an environment as a game) in mathematical terms, then we will cover Dynamic Programming, that allows to solve exactly the reinforcement learning problem in case we know everything about the environment. After that, we consider model-free methods, that allow us to learn even if we know nothing about the environment, these methods learn by interacting directly with the environment, obtaining experience and exploiting it to find the best actions to perform in any situation. Finally, we will delve into multiagent reinforcement learning, a scenario where multiple agents interact with the same environment. This chapter is based on the first part of the Sutton and Barto book [19]

2.1 Reinforcement Learning

Reinforcement learning is the field of machine learning that tries to maximize a numerical reward signal. The agent is not said which actions to take, but instead must discover the actions that lead to the highest reward by try-

ing them. Furthermore, actions do not only influence the immediate future reward but also the next situation and consequently subsequent rewards, so the agent must learn to balance immediate and future rewards.

Reinforcement learning differs from supervised learning because there is no external supervisor, which tells the agent which actions to perform in a given situation, instead the agent must discover the actions that lead to the highest reward by trying them. Reinforcement learning is also different from unsupervised learning, because even though both approaches do not need labels to learn from, their goal is different. In unsupervised learning the goal is to find structure in the data, while in reinforcement learning the goal is to maximize a reward signal.

One important aspect of reinforcement learning is the exploration-exploitation trade-off, which is not present in supervised or unsupervised learning. The agent must weigh the benefits of exploring new actions against the benefits of utilizing the best actions it has already discovered. A focus on exploration alone may lead to the discovery of the best actions, but with infrequent use, while a focus on exploitation alone may prevent the discovery of potentially better actions.

2.2 Markov Decision Processes

We can formalize the problem of learning from interaction to achieve a goal with Markov Decision Processes (MDPs). In the MPD framework the learner is called the agent, and the thing it interacts with is called the environment. The agent continually interacts with the environment by performing actions, on each action the environment changes state, and presents a new situation to the agent. The environment is also responsible to give the reward signal the agent must maximize 2.1.

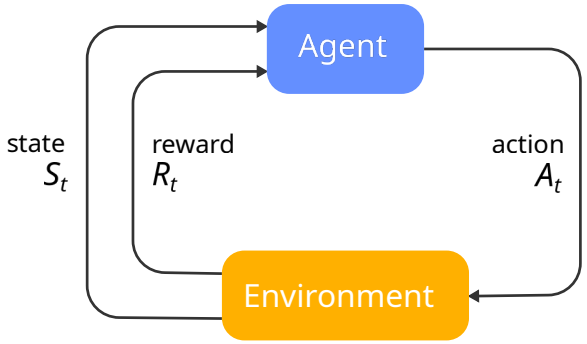

Figure 2.1: Illustration of the agent-environment interaction in a Markov Decision Process, image adapted from [19].

More specifically, the agent and environment interact in discrete time steps ($t = 0, 1, 2 \dots$). At each time step t the agent receives an observation of the environment's state $S_t \in S$ and based on that chooses an action $A_t \in A$. In the next time step $t+1$ the agent receives from the environment a numerical reward $R_{t+1} \in R \in \mathbb{R}$ and a new observation $S_{t+1} \in S$ and so on. This generates a sequence of observations, actions and rewards, which we call trajectory: $S_0, A_0, R_1, S_1, A_1, R_2, \dots$.

In finite MDPs the set of states S, actions A and rewards R are finite, so the distribution of next state and reward, given the current state and action is well defined. Furthermore, the distribution of the random variables R_t and S_t follows the Markov property, depending only on the previous state and action, $P(S_{t+1}|S_t) = P(S_{t+1}|S_t, S_{t-1}, \dots, S_0)$.

We can formalize mathematically an MDP using the following notation:

- S is the set of states
- A is the set of actions
- R is the set of rewards
- $p(s', r|s, a)$ is the dynamics function, which describes the distribution of the next state s' and reward r, given the current state s and action a, defined as:

$$p(s', r|s, a) = \mathbb{P}[S_{t+1} = s', R_{t+1} = r|S_t = s, A_t = a] \tag{2.1}$$

- $\gamma \in [0, 1]$ is the discount factor, which determines how much importance the agent gives to future rewards

2.2.1 Reward

The goal of the agent is formalized via a reward signal, given from the environment to the agent. At each time step the reward is a single number $R_t \in \mathbb{R}$. The agent goal is to maximize the total reward it receives over time.
For example if we want the agent to play a game, the reward signal can be the number of points the agent has scored. If we want the agent to learn to walk, the reward signal can be the distance the agent has traveled.
The reward signal is not a way of telling the agent how to do, if we were to do that the agent will accomplish side tasks without ever completing the main task that we want to be solved. For example in a racing game we might be tempted to give a reward whenever the car moves forward, but this will lead the agent to move forward and backward in a loop, without ever finishing the race. In fact, in this case finishing the race is the worst thing that could

happen, because then the agent will stop receiving the reward signal.
The reward signal thus specifies what we want the agent to do, not how to do it.

2.2.2 Episodic and Continuing Tasks

We can divide the kinds of problems that reinforcement learning deals with in two classes: episodic problems and continuing ones, which can be both formalized with MDPs 2.2.

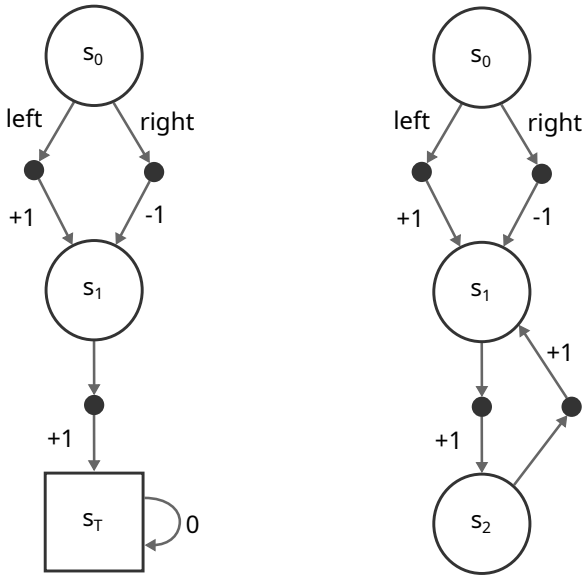

Figure 2.2: Empty circles represent the states, while solid circles the actions state-action pairs. The numbers indicate the reward the agent receives if it goes through the corresponding transition. Left: Illustration of an episodic task, the terminal state s_T is represented with a square. If the agent reaches the terminal state it can no longer receive any reward. Right: Illustration of a continuing task, when the agent reaches s_2 it comes back to s_1, receiving a reward of $r = 1$ in the process, in this MPD the expected return with $\gamma = 1$ would be infinite as the sequence of rewards after the first state is always 1.

In episodic tasks the agent interacts with the environment for a finite (yet random) number of time steps T, and then it reaches a terminal state, ending the episode. Examples of episodic tasks is the game of chess, as the episode ends when a player wins or a draw is made.
In continuing tasks, on the other hand, there is no clear notion of episode, as the agent interacts with the environment continually, potentially for an infinite number of time steps.
We can formalize both problems with MDPs, by defining the concept of terminal state: a terminal state is an absorbing state and corresponds to the end of the episode, when the agent reaches the terminal state it can no

longer escape it and all future rewards are zero.

2.2.3 Goal and Return

In reinforcement learning we formalize the goal of an agent as the maximization of the expected return $\mathbb{E}[G_t]$, where the return, denoted as G_t is the sum of all discounted rewards the agent receives in the future, starting from a given time step t:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.2}$$

$\gamma \in [0, 1]$ is the discount factor and is used for two main reasons:

1. γ allows tuning how much importance the agent gives to future rewards, so we can make the agent more or less patient. If $\gamma = 0$ the agent will only consider the immediate reward R_{t+1}, while if $\gamma = 1$ the agent will consider all future rewards with the same importance.

2. Allows the return to be used also in continuing tasks, because we can set $\gamma < 1$ and keep the sum from diverging.

Returns of subsequent time steps are related to each other in an important way:

$$G_t = R_{t+1} + \gamma G_{t+1} \tag{2.3}$$

Equation (2.3) is also called the one-step return.

2.2.4 Policy and value functions

A policy π is a mapping from states to actions, which specifies what action the agent should take in a given state. The policy is formally defined as the probability of taking action a in state s

$$\pi(a|s) = \mathbb{P}[A_t = a | S_t = s] \tag{2.4}$$

Given a policy we can define a value function $v_\pi(s)$ associated to the policy, which gives us a measure of how good it is for the agent to be in a given state when following policy π. The value function is defined mathematically as the expected return starting from a state s and then following policy π:

$$v_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s] \tag{2.5}$$

We denote with $v_\pi(s)$ the value function of policy π.

Similarly, we can define the value of selecting an action a in state s under policy π, denoted $q_\pi(s, a)$, as the expected return starting from s, taking the action a and then following policy π:

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a] \quad (2.6)$$

We call $q_\pi(s, a)$ the action-value function of policy π.

In practice, it is also convenient to define the advantage, which is the difference between the value of taking action a in state s and the value of the state:

$$A_\pi(s, a) = q_\pi(s, a) - v_\pi(s) \quad (2.7)$$

The advantage is a measure of how much better it is to take action a in state s, with respect to the other possible actions in state s.

2.2.5 Bellman Equation

A fundamental property of the value function is that it satisfies a recursive relationship similar to the one of the return (2.3):

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_t | S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} | S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)(r + \gamma v_\pi(s')) \end{aligned} \quad (2.8)$$

We call equation (2.8) the Bellman equation for v_π. The Bellman equation expresses the relationship between the value of current state s and the value of successive states s'. It tells us that the value of each state must equal the value of the next state plus the reward expected along the way. The Bellman equation forms the basis to compute or approximate v_π which we will see in the next section.

It is also possible to define the Bellman equation for the action-value function:

$$\begin{aligned} q_\pi(s, a) &= \mathbb{E}[G_t | S_t = s, A_t = a] \\ &= \mathbb{E}[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a] \\ &= \sum_{s',r} p(s', r|s, a)(r + \gamma v(s')) \\ &= \sum_{s',r} p(s', r|s, a)(r + \gamma \sum_{a'} \pi(a'|s') q_\pi(s', a')) \end{aligned} \quad (2.9)$$

Where in the last step we used the relationship $v_\pi(s) = \sum_a \pi(a|s) q_\pi(s,a)$ between the value function and action-value function.

2.2.6 Optimal policy and value function

Solving a reinforcement learning problem means finding a policy that achieves a high return. To accomplish this we first need to define an ordering over policies. We define that policy π is better than policy π' if the expected return is higher for π than for π'.

$$\pi \geq \pi' \iff v_\pi(s) \geq v_{\pi'}(s) \quad \forall s \in S \tag{2.10}$$

This way we can define the optimal policy as the policy that is at least better or equal to all other policies.

$$\pi^* \geq \pi' \quad \forall \pi' \tag{2.11}$$

Although there may be more than one optimal policy, all of them will have the same value-function v_*, so we can denote all the optimal policies with π^*. We define the optimal state-value function v_* as:

$$v_*(s) = \max_\pi v_\pi(s) \quad \forall s \in S \tag{2.12}$$

Which is the value function that achieves the maximum return from all states s. We can also define the optimal action-value function q_* as:

$$q_*(s,a) = \max_\pi q_\pi(s,a) \quad \forall s \in S, \forall a \in A \tag{2.13}$$

This function gives the expected return for taking action a in state s and then following policy π^*.

Another property of the optimal state-value function is that it satisfies a special form of the Bellman equation (2.8), called the Bellman optimality equation:

$$\begin{aligned} v_*(s) &= \max_a q_*(s,a) \\ &= \max_a \mathbb{E}_{\pi^*}[G_t | S_t = s, A_t = a] \\ &= \max_a \sum_{s',r} p(s', r|s, a)(r + \gamma v_*(s')) \end{aligned} \tag{2.14}$$

Note that in the final statement we didn't specify any policy, this is because we assume to be following an optimal policy, which is the one that always

selects the best action $a_* = \arg\max'_a q_*(s, a') \; \forall s \in S$. Analogously, we can define the Bellman optimality equation for the action-value function:

$$\begin{aligned} q_*(s, a) &= \mathbb{E}_{\pi^*}[G_t | S_t = s, A_t = a] \\ &= \sum_{s', r} p(s', r | s, a)(r + \gamma \max_{a'} q_*(s', a')) \end{aligned} \tag{2.15}$$

Intuitively, the Bellman optimality equation expresses the fact that the value of a state under an optimal policy must equal the expected return for the best action from that state.
In the next sections we will see how to solve the Bellman optimality equation both when we have a model of the environment (2.1) and when we don't.

2.3 Dynamic Programming

Dynamic programming is a technique that allows us to solve the Bellman optimality equation (2.14), finding the optimal policy π_* and value function v_* given a perfect model of the environment, that is, we completely know the dynamics function $p(s', r|s, a)$. We can split the problem of finding a solution to the Bellman optimality equation into two subproblems:

1. **Policy evaluation**: Given a policy π, find its value function v_π.
2. **Policy improvement**: Given the value function v_π, find a better policy π'.

2.3.1 Policy evaluation

A simple way to evaluate a policy π would be to solve the linear system of equations described by the Bellman equation (2.8). This could be done with any linear system solution method available in the literature, however, in reinforcement learning iterative solution methods are most suitable.

Algorithm 1 Policy evaluation

Require: A policy π, a model p, a discount factor γ, a threshold θ
Initialize a candidate value function $\hat{v}_\pi$
while True **do**
    $\hat{v}'_\pi = \hat{v}_\pi$
    for $s \in S$ **do**
        $\hat{v}'_\pi(s) = \sum_a \pi(a|s) \sum_{s', r} p(s', r|s, a)(r + \gamma \hat{v}_\pi(s'))$
    end for
    if $|\hat{v}'_\pi(s) - \hat{v}_\pi(s)| < \theta$ **then**
        return $\hat{v}_\pi$
    end if
end while

Algorithm 1 is called the iteative policy evaluation algorithm and is guaranteed to always converge to the true value function v_π with an error below the threshold θ if $\gamma < 1$, or if $\gamma = 1$ only if termination is guaranteed from all states under the policy π.
This can be proven mathematically by noting that the operator $B_\pi(v_\pi) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)(r + \gamma v_\pi(s'))$ is contracting, which means that the solution is both unique and is reached no matter how we initialize the value function $\hat{v}_\pi$.

In algorithm 1 we used an additional array to store the updated value function $\hat{v}'_\pi$, however, this is not required, as the algorithm will converge regardless of how we perform the updates, that is, we could have used only one array $\hat{v}_\pi$ and performed the update in line 5 in-place:
$\hat{v}_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)(r + \gamma \hat{v}_\pi(s'))$
In practice, the in-place algorithm usually converges faster than the two-array version because it uses updated values as soon as they are available.

2.3.2 Policy improvement

Our reason for computing the value function for a policy is to help find better policies. A simple improved version of π is a policy that in state s' takes the action that maximizes $q_\pi(s', a)$. This policy will be equal to π in all states $s \neq s'$ but will be better in state s', so $\pi' \geq \pi$.
We can take this idea further, by defining π' as the policy that takes the greedy action with respect to $q_\pi(s, a)$ in all states, with ties broken evenly. This policy is called the greedy policy and is defined as:

$$\pi'(a|s) = \arg\max_a q_\pi(s, a) = \arg\max_a \sum_{s',r} p(s', r|s, a)(r + \gamma v_\pi(s')) \quad (2.16)$$

The greedy policy always takes the action that looks best in the short term with just one-step look ahead, according to v_π. It can be shown that the greedy policy is always better or equal to π due to the policy improvement theorem, demonstrated on page 78 of the Sutton and Barto's book [19].

2.3.3 Policy iteration

A simple way to find the optimal policy is to iterate the policy evaluation and policy improvement steps until the policy doesn't change anymore. This process is guaranteed to converge to the optimal policy and value function with the same conditions that apply to the policy evaluation algorithm 1, that is: convergence is guaranteed to the true value function v_π with an error below the threshold θ if $\gamma < 1$, or if $\gamma = 1$ and the task is episodic.

Algorithm 2 Policy iteration

Require: A model p, a discount factor γ, a threshold θ

Initialize a policy π
while True **do**
    $v_\pi = \text{Policy evaluation}(\pi, p, \gamma, \theta)$ ▷ Algorithm 1
    $\pi' = \arg\max_a \sum_{s',r} p(s', r|s, a)(r + \gamma v_\pi(s'))$ ▷ Policy improvement
    if $\pi' = \pi$ **then**
        return π, v_π
    end if
end while

Algorithm 2 is called the policy iteration algorithm. The algorithm iterates the policy evaluation and policy improvement steps until the policy doesn't change anymore.

2.3.4 Value iteration

The drawback of the policy iteration algorithm 2 is that it requires an entire policy evaluation for each policy improvement step, requiring multiple sweeps over the state space. In practice, however, we can stop before policy evaluation finished without losing any of the convergence properties of policy iteration 2. One important special case is when policy evaluation is stopped after just one sweep, using the update:

$$v_{k+1}(s) = \max_a \sum_{s',r} p(s', r|s, a)(r + \gamma v_k(s')) \tag{2.17}$$

This update is called the value iteration update and combines one pass of policy evaluation and policy improvement into one, requiring only 1 sweep over the state space. The update (2.17) is similar to the policy evaluation update (2.8) except that we are using the maximum instead of the expectation.

Algorithm 3 Value iteration

Require: A model p, a discount factor γ, a threshold θ

Initialize a value function v_0
while True **do**
    $v_{k+1} = v_k$
    for $s \in S$ **do**
        $v_{k+1}(s) = \max_a \sum_{s',r} p(s', r|s, a)(r + \gamma v_k(s'))$
    end for
    if $|v_{k+1}(s) - v_k(s)| < \theta$ **then**
        return v_{k+1}
    end if
end while

Algorithm 3 usually converges faster than policy iteration 2. Furthermore, like the policy iteration algorithm, this algorithm converges even if we perform all updates in place, without needing to keep in memory both v_{k+1} and v_k, thus reducing memory requirements.

The value iteration algorithm 3 has a limitation in that it necessitates a complete examination of the entire state space, which can be an issue if the state space is large. However, the algorithm is still guaranteed to converge to the optimal value function even if updates are performed on certain states more frequently than others, as long as we continue to update all the states. These algorithms are called asynchronous DP algorithms because they update the values of the states in asynchronous fashion.

2.3.5 Generalized policy iteration

We can view both the policy iteration algorithm 2 and the value iteration algorithm 3 as special cases of a more general process referred to as generalized policy iteration (GPI). Almost all RL algorithms are described by this process where the policy is improved towards the value function and vice-versa, as shown in figure 2.3. This loop ends only when we reach a policy that is greedy with respect to its own value function, which implies that the value function satisfies the Bellman optimality equation (2.14).

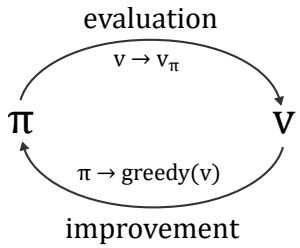

Figure 2.3: Generalized policy iteration loop. Image adapted from [19].

Another way to see the process is in terms of two different goals, the two converging lines represented in figure 2.4. The first goal is to compute the value function of a policy $v = v_\pi$, which is achieved by policy evaluation. The second goal is to make the policy greedy with respect to its own value function $\pi = \text{greedy}(v)$, which is achieved by policy improvement. The two goals are complementary and the process converges to the optimal value function when the two goals are achieved simultaneously.

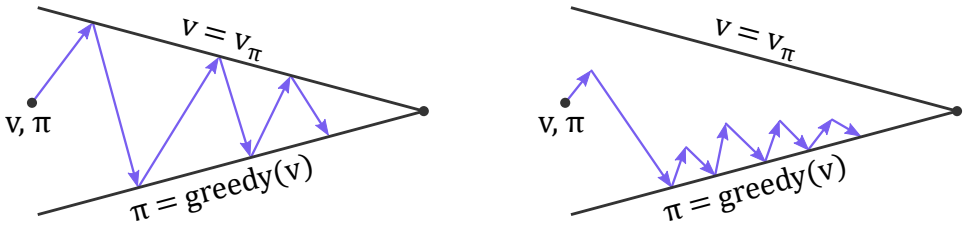

Figure 2.4: Another way to see generalized policy iteration. Left: policy iteration with full policy evaluation at each step. Right: policy iteration with truncated policy evaluation at each step. Image adapted from [19].

2.4 Monte Carlo methods

Monte Carlo methods are a class of algorithms that learn directly from experience, without the need for a model of the environment. The main idea is to estimate the value function v_π by averaging the returns G_t following each state. These methods can only be applied to episodic tasks because we update the value function using G_t, which is only available after an episode has finished.
As with dynamic programming we can split the problem of finding the optimal policy into two parts.

1. Monte Carlo prediction: estimate the value function v_π corresponding to a given policy π
2. Monte Carlo control: find the optimal policy π^*

2.4.1 Monte Carlo prediction

Suppose we have a policy π, and we want to estimate the value function v_π given a set of episodes obtained by following policy π, and passing through state s at time t. We call each occurrence of state s in an episode a visit to s. Of course, s may be visited multiple times in the same episode, so we can call the first time s is visited in an episode the first visit to s. We can then identify two main methods to perform Monte Carlo policy evaluation:

1. First-visit MC method: estimate $v_\pi(s)$ as the average of the returns following first visits to s.
2. Every-visit MC method: estimate $v_\pi(s)$ as the average of the returns following all visits to s.

For example, in the first visit case we estimate the value function v_π as follows:

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi(G_t|S_t = s) \\ &= \frac{1}{N}\sum_{i=1}^{N} G_t^{(i)} \\ &= \frac{1}{N}\sum_{i=1}^{N}\sum_{t'=t}^{T} \gamma^{t'} R_{t'+1}^{(i)} \end{aligned} \tag{2.18}$$

Where $G_t^{(i)}$ is the return after visiting state s in the i-th episode, N is the number of episodes and T is the episode length.
As the number of visits to state s increases, both first-visit and every-visit Monte Carlo methods converge to $v_\pi(s)$. However, the two methods have different theoretical characteristics. First-visit MC provides independent and identically distributed estimates of $v_\pi(s)$, whereas every-visit MC does not. Despite its less straightforward nature, every-visit MC still maintains the same convergence properties as first-visit MC.

Monte Carlo methods can be utilized to estimate the action-value function $q_\pi(s, a)$ as well, which is critical for control tasks. The concept of visits is redefined to apply to state-action pairs, rather than states alone. A state-action pair (s, a) is considered visited if action a was selected while in state s.

2.4.2 Monte Carlo control

In the control problem, MC methods draw from the idea of generalized policy iteration (GPI) already presented: alternating between policy evaluation and policy improvement.
In the policy evaluation step we estimate the action-value function q_π using the first visit or every visit MC method. However, we need to ensure that all states will be visited, to accomplish this we can generate the episodes with exploring starts, that is, all episodes begin with state-action pairs randomly selected to cover all possibilities.
The policy improvement step, on the other hand, is performed by greedily selecting the action that maximizes the action-value function (2.19), which is the same as selecting the action that maximizes the expected return.

$$\pi'(s) = \arg\max_a q_\pi(s, a) \tag{2.19}$$

A major problem with this approach is the exploring starts assumption, because, if we are not given a model of the environment, we cannot know all state-action pairs beforehand, and even if we did, we would still need to generate many episodes to ensure that all state-action pairs are visited.

Without exploring starts, the only general way to ensure that all actions are selected infinitely often is for the agent to continue to select them. This can be done in two ways:

1. We can use a different policy to generate the episodes, a behavioral policy, different from the one we are trying to improve, the target policy. This is called off-policy learning.

2. We can avoid making the policy completely greedy like in equation (2.19) and instead use an ϵ-soft policy, which is a policy that is not completely greedy, but gives at least probability $\epsilon/|A|$ to each action. An example of such policy is the ϵ-greedy policy, defined as follows:

$$\pi(a|s) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{|A|} & \text{if } a = \text{argmax}_{a'} q_\pi(s, a') \\ \frac{\epsilon}{|A|} & \text{otherwise} \end{cases} \tag{2.20}$$

2.5 Temporal-Difference learning

Temporal-difference learning, also called TD-Learning, is a class of algorithms that combine both Monte Carlo and dynamic programming ideas. Like Monte Carlo, TD-learning methods learn directly from experience, not requiring a model of the environment, and like DP it updates the value function with already learned estimates, using a technique known as bootstrapping. This allows TD methods to be also applied to continuing tasks because they can update the value of a state as soon as new information is available.
Like we did with Monte Carlo methods, we first focus on the prediction problem, then the control one.

2.5.1 TD prediction

TD prediction uses the one-step return, represented as $G_{t:t+1} = R_{t+1} + \gamma G_{t+1}$ (as defined in equation (2.3)), rather than the full return (defined in equation (2.2)) to estimate the value function. To obtain the expected return, which is needed to estimate the value function, we take the expected value of the recursive formula.

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi(G_t | S_t = s) \\ &= \mathbb{E}_\pi(R_{t+1} + \gamma G_{t+1} | S_t = s) \end{aligned} \tag{2.21}$$

This expectation (2.21) can be directly estimated by sampling episodes according to the policy π. The value function v_π is then updated as follows:

$$V(S_t) = V(S_t) + \alpha(R_t + V(S_t) - V(S_{t+1})) \tag{2.22}$$

Where we introduced a parameter α which is the learning rate of the algorithm, the learning rate controls how much the new estimate of the value gathered in the episode should count with respect to the previous value estimate $V(S_t)$.
Finally, the quantity in parentheses $\delta = R_t + V(S_t) - V(S_{t+1})$ is known as the TD error, and measures the difference between the estimated value of the current state $V(S_t)$ and the better estimate $R_t + V(S_{t+1})$, given by the episode.
Updating the value function in this way (2.22) introduces bias in the value estimate, however, this also reduces the variance, which is why TD methods usually learn faster Monte Carlo methods.

2.5.2 TD control

In TD control we need to estimate the action-value function $q_\pi(s, a)$, rather than a state-value function. There are three main methods to do so, Sarsa, Expected Sarsa and Q-learning.

Sarsa

Sarsa is the simplest of the three, it is an on-policy TD control method that uses the following update rule to estimate the action-value function:

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha(R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)) \tag{2.23}$$

The method is called Sarsa because in state S_t it selects action A_t, then observes reward R_{t+1} and next state S_{t+1}, finally it selects the next action A_{t+1}.
In Sarsa $Q(S_{t+1}, A_{t+1})$ is a sample of the value of the next state $V(S_{t+1})$, because the next action A_{t+1} is selected according to policy π.

$$\begin{aligned} v_\pi(s') &= \sum_a q(s', a)\pi(a|s') \\ &= \mathbb{E}_\pi(Q(S_{t+1}, A_{t+1})|S_{t+1} = s') \end{aligned}$$

Expected Sarsa

Expected Sarsa, estimates the value of the next state by using directly its expected value $V(S_{t+1})$. The update rule of expected Sarsa is thus the following.

$$\begin{aligned} Q(S_t, A_t) &= Q(S_t, A_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - Q(S_t, A_t)) \\ &= Q(S_t, A_t) + \alpha(R_{t+1} + \gamma \sum_a Q(S_{t+1}, a)\pi(a|S_{t+1}) - Q(S_t, A_t)) \end{aligned} \tag{2.24}$$

This update reduces the variance of the value estimate, but it also increases its computational cost, because to compute the value of the next state we must sum over all actions, $V(S_{t+1}) = \sum_a Q(S_{t+1}, a)\pi(a|S_{t+1})$, which can be computationally expensive when the number of actions is high.

Q-learning

Q-learning is an off-policy TD control method, that uses the following update rule to estimate the action-value function:

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha(R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)) \quad (2.25)$$

Q-learning assumes that the next action A_{t+1} is selected according to a greedy policy with respect to the action-value function $Q(S_{t+1}, a)$. In this algorithm, the learned action-value function directly approximates the optimal action-value function q_*, independently of the policy being followed. However, the policy π still has an important effect because it determines which state-action pairs are visited and is thus called the behavioral policy.

2.6 n-step methods

n-step methods merge the gap between MC and TD methods, so that one can shift from one to the other smoothly. This is important because the best method for a given problem is often intermediate between these two extremes figure 2.5.

n-step methods estimate the value function using the n-step return, which is the sum of the rewards obtained in the next n steps, plus the value of the state reached after n steps.

$$G_{t:t+n} = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n V(S_{t+n}) \quad (2.26)$$

Where the subscript $t : t + n$ indicates that we consider the rewards up to time $t + n$ and approximate subsequent rewards with the value of the state S_{t+n}.
We then update the value function as follows:

$$V(S_t) = V(S_t) + \alpha(G_{t:t+n} - V(S_t)) \quad (2.27)$$

If n is greater than the episode length T, then the n-step return is equal to the full return, and we have Monte Carlo prediction. While, if n is equal to 1, then we have TD prediction.

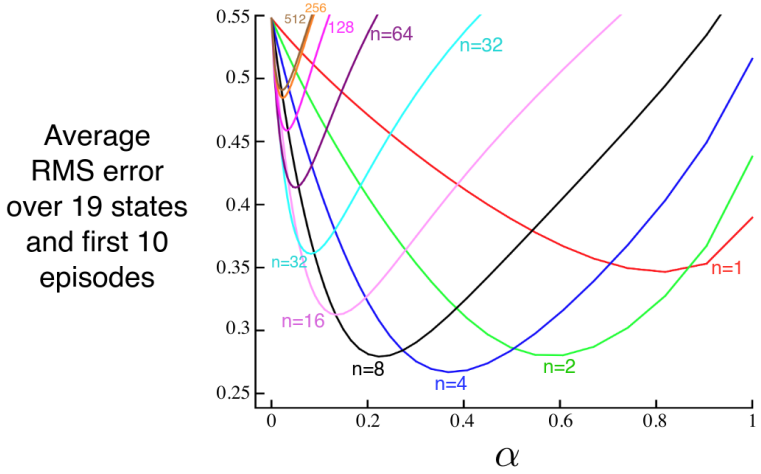

Figure 2.5: Comparison between n-step methods as a function of α for various values of n in a random-walk task. An intermediate value of $n = 4$ performs best, converging faster than both TD methods ($n = 1$), and MC methods ($n = 512$). Image taken from [19].

It can be shown that all n-step TD methods converge to the correct value function v_π under appropriate technical conditions. The n-step TD methods hence form a family of sound methods, with one-step TD methods and Monte Carlo methods as extreme members.

For control problems, we can generalize one-step TD methods to their respective n-step versions by defining the n-step return in terms of estimated action values.

$$G_{t:t+n} = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n Q(S_{t+n}, A_{t+n}) \tag{2.28}$$

The update rule of n-step Sarsa is then the following:

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha(G_{t:t+n} - Q(S_t, A_t)) \tag{2.29}$$

In essence, a new TD control algorithm can be constructed by incorporating a formula for the return into Equation (2.29).
For example, we can define n-step Expected Sarsa by defining the n-step return in terms of the expected value of state S_{t+n}.

$$G_{t:t+n} = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n \sum_a Q(S_{t+n}, a)\pi(a|S_{t+n}) \tag{2.30}$$

And n-step Q-learning by defining the n-step return in terms of the maximum value of state S_{t+n}.

$$G_{t:t+n} = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n \max_a Q(S_{t+n}, a) \tag{2.31}$$

It can be shown that all n-step methods converge to the optimal value function if the leaning and that the convergence rate is independent of n.

2.7 Multiagent reinforcement learning

In multiagent reinforcement learning (MARL), multiple agents interact with the same environment. In the most general case, each agent receives from the environment the reward $R_t^{(i)}$ and observation $S_t^{(i)}$, and can execute an action from set $A_t^{(i)}$, where i indicates the agent's index and t the time, figure 2.6. While in single-agent RL we only need to maximize G_t, in multiagent RL each agent maximizes its own expected future reward $G_t^{(i)}$.

$$G_t^{(i)} = \mathbb{E}\left[\sum_{t'=0}^{\infty} \gamma^{t'} R_t^{(i)}\right] \tag{2.32}$$

Where γ is the discount factor. The MARL setting is very general, as it can describe a wide range of games, ranging from chess to CS:GO, encompassing all finite-player games with either perfect or imperfect information, basically covering all games humans have played so far. Solving a MARL problem can be hard, because each player must take into account not only the interaction with the environment but also the actions of other players, which might counter or aid the player. However, it can be shown that in the case of a finite number of agent, finite time horizon and finite action spaces there exists an optimal policy for each agent, which is the Nash equilibrium of the game [6].

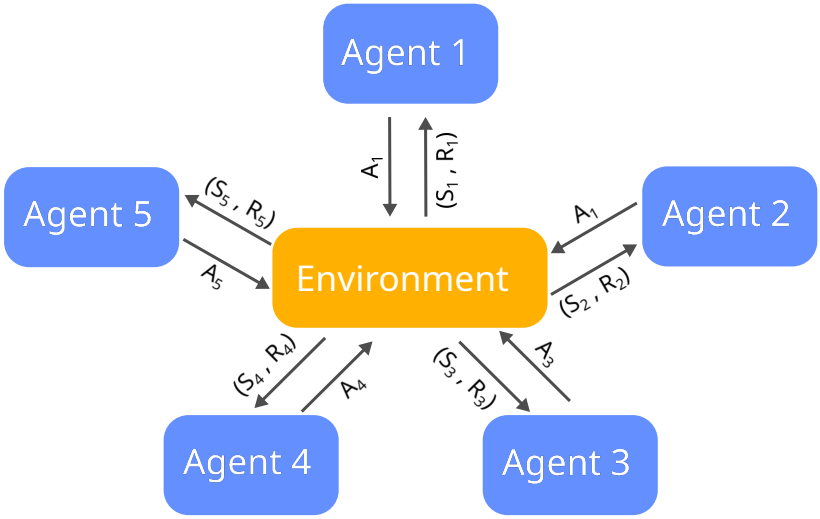

Figure 2.6: Illustration of the multiagent reinforcement learning scenario with 5 agents. Each agent receives from the environment its reward R_i and next state S_i, then chooses an action A_i and executes it. In this figure we omitted the time index t to simplify the notation.

So far we described the general MARL problem, however interesting problems often fall into two main categories: cooperative and competitive environments. In a cooperative setting, the agents work together to achieve a common goal, and can thus be treated as a single-agent problem by defining the reward function as the sum of individual rewards, and the action space as the cartesian product of individual action spaces. In contrast, competitive environments consist of adversaries that try to outperform each other and cannot be reduced to a single-agent problem.

Multiagent reinforcement learning brings new challenges, such as the curse of dimensionality, which becomes even more pronounced with the exponential growth of the state-action space with the number of agents. Additionally, the exploration-exploitation trade off is more complex as agents not only need to learn about the environment, but also about the behavior of other agents [20].

One prominent example of how to approach a MARL problem is the game of Dota 2, where OpenAI beat the 2019 world champions using a centralized learning algorithm. In this approach, a single policy was learned for all agents of the same team, while the other team used previous iterations of the same policy [2]. This was possible because Dota 2 is mostly symmetric, meaning that the two teams are presented with similar choices and the actions a player can take are the same regardless of their team.

2.8 Discussion

In this chapter we introduced the main concepts of reinforcement learning, we saw that reinforcement learning is a framework for learning in sequential decision-making problems, where the agent interacts with the environment and learns to maximize its reward.

We then presented Monte Carlo methods, which are a class of algorithms that use the full return (2.2) to estimate the value function, and TD methods, which use the one-step return (2.3). TD methods are usually better than MC methods because they can be applied to continuing tasks, and reduce the variance of the value estimate.

After that, we discussed n-step methods, which estimate the value function using the n-step return formula (2.26). These methods have been proven to converge to the correct value function v_π under specific conditions and thus represent a family of reliable techniques, with one-step TD methods and Monte Carlo methods being their two ends of the spectrum.

Finally, we discussed multiagent reinforcement learning, which is a general framework for describing environments that involve multiple agents.

Chapter 3

Policy Gradient Methods

So far we focused on action-value methods, those methods learned the action-value function q and used it to derive a policy. In this chapter, we will focus on methods that learn the policy directly, without consulting a value, or action-value function. The value function still has an important role as it's often used to learn the policy, however, it is not required for action selection. This class of methods is called policy gradient methods.

Policy gradient methods can be applied without function approximation. However, in this discussion, we will focus on the function approximation setting, where the policy is expressed as a function of the state s and a vector of parameters $\boldsymbol{\theta}$, expressed as $\pi(a|s;\boldsymbol{\theta})$.

We will begin by covering the basics of function approximation, specifically neural networks, which will serve as the foundation for all the algorithms discussed in this chapter. We will then delve into the policy gradient theorem for discrete action spaces and introduce REINFORCE, a Monte Carlo policy gradient algorithm.
Our main focus will be on actor-critic methods, which have achieved significant success in various domains, such as playing games like Dota 2 [2], StarCraft II [21, 22], Go [14], Minecraft [23, 24] with also applications to language like ChatGPT [18]. We will not provide an exhaustive overview of actor-critic methods, but we will discuss key ones, such as TRPO and PPO. The latter will also be used to in the subsequent chapter to learn to play Briscola.

3.1 Function Approximation

The methods discussed so far represent the action-value function $q(s, a)$ as a table and are thus called tabular methods. These methods have been shown to converge to an optimal policy under certain conditions, such as proper

selection of the step-size α and the policy ϵ parameter. However, tabular methods can be impractical to use in certain scenarios, such as when the state space is extremely large (e.g. in the game of chess) or when the state space is continuous. In these cases, function approximation must be used to represent the value function, action-value function, and policy.

To keep things clear, we will focus on approximating the value function, denoted as $\hat{v}(s; \boldsymbol{w})$, where $\boldsymbol{w}$ is a parameter vector. The same ideas can also be applied to the action-value function or the policy. Furthermore, we assume to perform function approximation with neural networks, as they are widely used in practical applications and offer differentiable function approximation [25], which is a critical requirement for the policy gradient theorem. Consider the problem of finding the value function corresponding to a policy π, thus far we implemented the update rule for the value function by moving the value prediction $V(s)$ closer to the return G_t, which represents the target:

$$V(s) = V(s) + \alpha(G_t - V(s)) \tag{3.1}$$

With function approximation the idea is the same, moving the prediction closer to the target. But instead of updating the value of the table entry $V(s)$ directly, we need to update the parameter vector $\boldsymbol{w}$. The way we usually update the parameter vector $\boldsymbol{w}$ in neural networks is with gradient descent, that is, we update $\boldsymbol{w}$ in the negative direction of the gradient of a loss function L.

$$\boldsymbol{w} = \boldsymbol{w} - \alpha \nabla_w L(\hat{v}(s; \boldsymbol{w}), G_t) \tag{3.2}$$

The loss function measures how far the value prediction $\hat{v}(s; \boldsymbol{w})$ is from the target G_t. A natural loss function for learning the value is the *mean squared error loss* denoted as L_{MSE}:

$$L_{\text{MSE}} = \frac{1}{2}(G_t - \hat{v}(S_t; \boldsymbol{w}))^2 \tag{3.3}$$

The gradient of the loss function (3.3) with respect to $\boldsymbol{w}$, can be quickly computed:

$$\nabla_w L_{\text{MSE}} = (\hat{v}(S_t; \boldsymbol{w}) - G_t) \nabla_w \hat{v}(S_t; \boldsymbol{w}) \tag{3.4}$$

Where $\nabla_w \hat{v}(s; \boldsymbol{w})$ is the gradient of the output of the function with respect to the parameter vector $\boldsymbol{w}$ and can be quickly found with automatic differentiation techniques.

Tabular methods can be viewed as a special case of function approximation, where we have one parameter for each state representing the value of that state. In this setting, the gradient of the loss function is simply the one-hot vector corresponding to state S_t:

$$\nabla_w L_{\text{MSE}} = (\hat{v}(S_t; \boldsymbol{w}) - G_t) \, \mathbf{1}_{S_t} \tag{3.5}$$

Where the one-hot vector $\mathbf{1}_{S_t}$ is a vector of zeros with a one in the position corresponding to the state S_t. The update rule (3.2) then becomes:

$$\boldsymbol{w} = \boldsymbol{w} - \alpha \left(\hat{v}(S_t; \boldsymbol{w}) - G_t \right) \mathbf{1}_{S_t} \tag{3.6}$$

Which is exactly the update rule (3.1) for tabular methods. Function approximation is hence a generalization of tabular methods, where the parameter vector $\boldsymbol{w}$ replaces the table of values.

Note that in the gradient derivation (3.4) we assumed $\nabla_w G_t = 0$. While this is true for Monte Carlo methods, where $G_t = R_{t+1} + \gamma R_{t+2} + \cdots + R_T$, it is not correct for TD methods, where the one-step return is used $G_t = R_{t+1} + \gamma \hat{v}(S_{t+1}; \boldsymbol{w})$. In this case, the gradient is not a true-gradient but a semi-gradient, because we left out the $\nabla_w G_t = \gamma \nabla_w \hat{v}(S_{t+1}; \boldsymbol{w})$ term.
It can be shown that semi-gradient methods have better convergence properties than true-gradient ones, in particular true-gradient methods even in the tabular case won't converge to the true value function v_π in stochastic environments. The interested reader can find an example of this behavior in chapter 11.5 of the Sutton and Barto book [19].

3.1.1 Policy Approximation

In policy gradient methods, the policy can be parameterized in any way, as long as it is differentiable with respect to the parameter vector $\boldsymbol{\theta}$. Approximating the policy directly has many advantages over approximating the action-value function:

- The approximate policy can approach a deterministic policy, whereas with ϵ-greedy action selection the policy always has a probability of selecting a random action.
- It enables the selection of actions with arbitrary probabilities, for example, in imperfect information games performing always the same action is not optimal, because the opponent can learn the player's strategy and exploit it. Think about the game of poker, the best strategy isn't to always bluff or always check, but to do a mix of both, so that the opponent can't predict the player's behavior.
- The action probabilities change smoothly with the parameter vector $\boldsymbol{\theta}$, whereas with ϵ-greedy action selection the probability can change dramatically for a small change in the action-value function.
- Because of the previous point, stronger convergence guarantees are available for policy-gradient methods.

3.2 Policy Gradient Theorem

To understand the policy gradient theorem, we need to define a performance measure of a policy π with respect to a parameter vector $\boldsymbol{\theta}$:

$$J(\boldsymbol{\theta}) = v_{\pi_\theta}(s_0) = \sum_a \pi_\theta(a|s_0) q_{\pi_\theta}(s_0, a) \tag{3.7}$$

Where v_{π_θ} represents the true value function for the policy π_θ, determined by the parameter vector $\boldsymbol{\theta}$.

The problem with equation (3.7) is that its gradient depends also on the gradient of the action-value function q_{π_θ}, which in turn depends on the state distribution $\mu(s)$ which is unknown. Fortunately $\nabla_\theta J(\theta)$ can still be computed via the policy gradient theorem, proven in chapter 13.2 of the Sutton and Barto book [19]:

$$\begin{aligned} \nabla_\theta J(\boldsymbol{\theta}) &\propto \sum_s \mu(s) \sum_a q_\pi(s, a) \nabla_\theta \pi_\theta(a|S_t) \\ &= \mathbb{E}_{\pi_\theta}\Big[\sum_a q_\pi(S_t, a) \nabla_\theta \pi_\theta(a|S_t) \Big] \end{aligned} \tag{3.8}$$

The proof relies on approximating the state distribution $\mu(s)$ with the empirical distribution of states visited during policy evaluation. This enables us to estimate the gradient of the performance measure (3.7) with respect to the policy parameters without needing the derivative of the state distribution.

The policy gradient theorem is a fundamental result in reinforcement learning that gives a way to compute the gradient of the performance measure with respect to the policy parameters.
We can further simplify expression (3.8) by multiplying and dividing by $\pi_\theta(a|S_t)$, removing the internal sum over actions:

$$\begin{aligned} \nabla_\theta J(\boldsymbol{\theta}) &\propto \mathbb{E}_{\pi_\theta}\Big[\sum_a q_\pi(S_t, a) \nabla_\theta \pi_\theta(a|S_t) \Big] \\ &= \mathbb{E}_{\pi_\theta}\Big[q_\pi(S_t, A_t) \frac{\nabla_\theta \pi_\theta(A_t|S_t)}{\pi_\theta(A_t|S_t)} \Big] \\ &= \mathbb{E}_{\pi_\theta}\Big[q_\pi(S_t, A_t) \nabla_\theta \log \pi_\theta(A_t|S_t) \Big] \end{aligned} \tag{3.9}$$

Where in the last step we used the equality $\nabla_\theta \pi_\theta(a|s)/\pi_\theta(a|s) = \nabla_\theta \log \pi_\theta(a|s)$, which is true for any differentiable function $\pi_\theta(a|s)$.
This is the most commonly used form of the policy gradient theorem and allows the computation of the policy gradient even in continuous action spaces.

3.3 REINFORCE

REINFORCE estimates the action-value function $q_{\pi_\theta}(S_t, A_t)$ of the policy gradient (3.9) using Monte Carlo.

$$q_\pi(S_t, A_t) = G_t = \sum_{t'=t}^{T} \gamma^{t'-t} R_{t'+1} \tag{3.10}$$

Thus the update rule for the policy parameter vector $\boldsymbol{\theta}$ is:

$$\theta_{t+1} = \theta_t + \alpha G_t \frac{\nabla_\theta \pi_\theta(A_t|S_t)}{\pi_\theta(A_t|S_t)} \tag{3.11}$$

This update increases the probability of actions proportional to the return and inversely proportional to the probability of the action. This makes sense because we move towards actions that lead to a higher return and perform a bigger update for actions with a lower probability, which are selected and hence updated less often.

3.3.1 REINFORCE with Baseline

The policy gradient theorem (3.8) can be generalized by adding a baseline function $b(s)$ to the action-value function without loss of generality:

$$\begin{aligned}
&\sum_s \mu(s) \sum_a (q_\pi(s,a) - b(s)) \nabla_\theta \pi_\theta(a|S_t) = \\
&= \sum_s \mu(s) \sum_a (q_\pi(s,a)) \nabla_\theta \pi_\theta(a|S_t) - \sum_s \mu(s) b(s) \sum_a \nabla_\theta \pi_\theta(a|S_t) \\
&= \nabla_\theta J(\boldsymbol{\theta}) - \sum_s \mu(s) b(s) \nabla_\theta \sum_a \pi_\theta(a|s) \\
&= \nabla_\theta J(\boldsymbol{\theta})
\end{aligned} \tag{3.12}$$

Where in the last step we used the fact that $\nabla_\theta \sum_a \pi_\theta(a|s) = 0$. The baseline function $b(s)$ can be any function of the state s and is used to reduce the variance of the gradient estimator. However, one natural choice for the baseline function is the value function $v_{\pi_\theta}(s)$, which gives the REINFORCE with baseline update:

$$\theta_{t+1} = \theta_t + \alpha (G_t - \hat{v}_{\pi_\theta}(S_t)) \frac{\nabla_\theta \pi_\theta(A_t|S_t)}{\pi_\theta(A_t|S_t)} \tag{3.13}$$

Where $\hat{v}_{\pi_\theta}(S_t)$ is estimated with Monte Carlo. The improvement of the REINFORCE with baseline algorithm over the REINFORCE algorithm is shown in Figure 3.1.

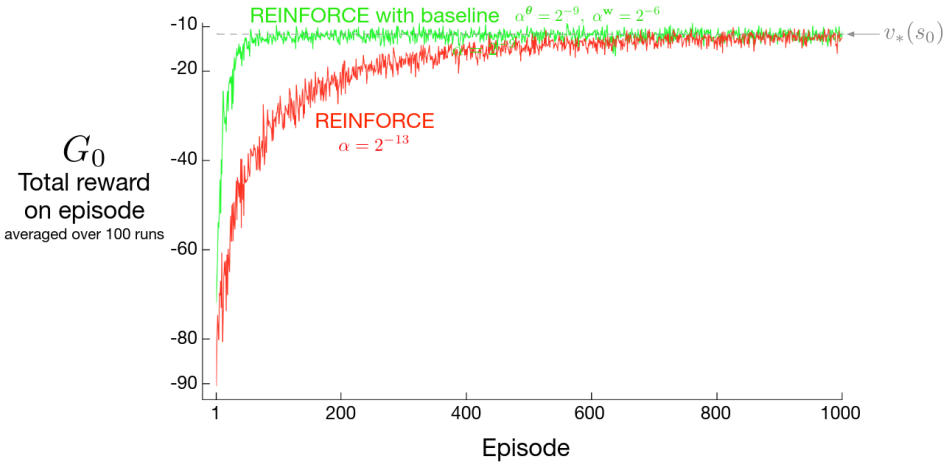

Figure 3.1: Comparison of the REINFORCE and REINFORCE with baseline algorithms on the short-corridor gridworld, with the optimal step sizes indicated. The results show that the baseline algorithm outperforms the REINFORCE algorithm, reaching the best return in just 100 steps, while the REINFORCE algorithm takes roughly 1000 steps. Image taken from [19].

3.4 Actor-Critic Methods

The REINFORCE with baseline algorithm is unbiased, but it has a high variance and is therefore slow to converge. Additionally, it is challenging to extend it to continuous tasks. In order to address these issues, Actor-Critic methods merge policy gradient and bootstrapping by using the one-step return $G_{t:t+1}$ instead of the full return G_t to update the policy.

More in detail, the policy parameter vector $\boldsymbol{\theta}$ is updated using policy gradient and TD-learning is used to update the value function $v_{\pi_\theta}(s; \boldsymbol{w})$. This adds a bias in the policy gradient update, but it also reduces variance and speeds up learning.
In particular, one-step actor-critic methods update the policy as following:

$$\theta_{t+1} = \theta_t + \alpha(G_{t:t+1} - \hat{v}_{\pi_\theta}(S_t; \boldsymbol{w}))\frac{\nabla_\theta \pi_\theta(A_t|S_t)}{\pi_\theta(A_t|S_t)} \tag{3.14}$$

Which is similar to the REINFORCE with baseline update (3.13), but uses the one-step return $G_{t:t+1} = R_{t+1} + \gamma\hat{v}(S_{t+1}; \boldsymbol{w})$ instead of the full return G_t (2.2).
At the same time, the value function parameters $\boldsymbol{w}$ are updated towards the bootstrapping target $G_{t:t+1}$:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \alpha(G_{t:t+1} - \hat{v}_{\pi_\theta}(S_t; \boldsymbol{w}))\nabla_{\boldsymbol{w}} \hat{v}_{\pi_\theta}(S_t; \boldsymbol{w}) \tag{3.15}$$

The actor-critic update can be generalized to n-step returns by simply replacing the one-step return $G_{t:t+1}$, with the n-step return $G_{t:t+n}$ (2.26) in

equations (3.14) and (3.15).

3.5 Trust Region Policy Optimization

Trust Region Policy Optimization (TRPO) [26] is an actor-critic method that uses a trust region to constrain the policy update. The TRPO update tries to improve the policy as much as possible while satisfing a special constraint that says how much the old and updated policy are allowed to differ. TRPO fixes a major problem of vanilla policy gradient, where the policy update, even though it changes the parameters only slightly, can have dramatic effects on performance, thus a bad policy update can lead to performance collapse.

Mathematically, the TRPO update is defined as follows:

$$\begin{aligned} \theta_{t+1} = &\arg\max_{\theta} \; L(\theta_t, \theta) \\ &\text{s.t. } \bar{D}_{KL}(\pi_{\theta_t}, \pi_\theta) \leq \delta \end{aligned} \tag{3.16}$$

Where $L(\theta_t, \theta)$ is the surrogate advantage, δ is a trust region parameter and $\bar{D}_{KL}(\pi_{\theta_t}, \pi_\theta)$ is the average KL-divergence across states between the old and new policy.
More in detail, $L(\theta_t, \theta)$ tells us how much better the new policy is according to data from the old policy

$$L(\theta_t, \theta) = \mathop{\mathbb{E}}_{s,a\sim\pi_{\theta_t}} \left[\frac{\pi_\theta(a|s)}{\pi_{\theta_t}(a|s)} A_{\pi_{\theta_t}}(s,a) \right] \tag{3.17}$$

Where $A_{\pi_{\theta_t}}(s,a)$ is the advantage function (2.7), which measures how much better action a is than the average action in state s, while the factor $\frac{\pi_\theta(a|s)}{\pi_{\theta_t}(a|s)}$ is the ratio of the probabilities of the new and old policy. So, if the new policy puts more weight on actions with higher advantage, then the surrogate advantage will be higher and the new policy will be better than the old one. Whereas the constraint $\bar{D}_{KL}(\pi_{\theta_t}, \pi_\theta)$ is the average KL-divergence across states between the old and new policy.

$$\bar{D}_{KL}(\pi_{\theta_t} || \pi_\theta) = \mathop{\mathbb{E}}_{s\sim\pi_{\theta_t}} [D_{KL}(\pi_{\theta_t}(\cdot|s) || \pi_\theta(\cdot|s))] \tag{3.18}$$

The TRPO update in equation (3.16) is a constrained optimization problem, which can be solved using a variety of methods. Here we will provide a simple solution by approximating the objective (3.17) to the first order and the constraint (3.18) to the second order.

$$L(\theta_t, \theta) \approx g^T(\theta - \theta_t) \tag{3.19}$$

$$\bar{D}_{KL}(\pi_{\theta_t}||\pi_\theta) \approx \frac{1}{2}(\theta - \theta_t)^T H(\theta - \theta_t) \tag{3.20}$$

The reason we are approximating the constraint to the second order is because the KL divergence has minima at $\theta = \theta_t$, so its gradient is zero.

With these approximations the TRPO update becomes:

$$\begin{aligned} \theta_{t+1} = &\arg\max_\theta \ g^T(\theta - \theta_t) \\ &\text{s.t. } \frac{1}{2}(\theta - \theta_t)^T H(\theta - \theta_t) \leq \delta \end{aligned} \tag{3.21}$$

The solution to this optimization problem is on the boundary, because we are maximizing a linear function on a convex domain, so it can be solved exactly using the Lagrangian method, yielding the solution:

$$\theta_{t+1} = \theta_t + \sqrt{\frac{2\delta}{g^T H^{-1} g}} H^{-1} g \tag{3.22}$$

Note that no learning rate is needed in the TRPO update, since the trust region parameter δ controls the size of the update.
A problem of the approximate update (3.22) is that the constraint (3.18) can be broken before reaching the boundary, to solve this problem we can use a backtracking line search to find the maximum step size that satisfies the constraint.

$$\theta_{t+1} = \theta_t + \alpha^j \sqrt{\frac{2\delta}{g^T H^{-1} g}} H^{-1} g \tag{3.23}$$

Where $\alpha \in (0, 1)$ is the backtracking coefficient and j is the smallest integer such that constraint (3.18) is satisfied.

3.6 Proximal Policy Optimization

Proximal Policy Optimization (PPO) [27] tries to answer the same question as TRPO, how can we take the biggest policy improvement without having a performance collapse? The main difference between PPO and TRPO is that PPO solves the problem using a first order method, which is much simpler to implement and faster to train, as we don't need to compute the inverse Hessian-gradient product (3.23). There are two versions of PPO, PPO-penalty and PPO-clip. We will focus on PPO-clip, as it is the most popular one.

PPO updates the policy using the following equation:

$$\theta_{t+1} = \arg\max_{\theta} \mathop{\mathbb{E}}_{s,a\sim\pi_{\theta_t}} [L(s, a, \theta_t, \theta)] \tag{3.24}$$

Where L is defined as:

$$\begin{aligned} L(s, a, \theta_t, \theta) = \min \Bigg(& \frac{\pi_\theta(a|s)}{\pi_{\theta_t}(a|s)} A_{\pi_{\theta_t}}(s, a), \\ & \text{clip}(\frac{\pi_\theta(a|s)}{\pi_{\theta_t}(a|s)}, 1-\epsilon, 1+\epsilon) A_{\pi_{\theta_t}}(s, a) \Bigg) \end{aligned} \tag{3.25}$$

The objective function $L(s, a, \theta_t, \theta)$ in equation (3.25) is defined as the minimum between two terms. The first term represents the surrogate advantage function and the second term is a clipped version of the policy ratio $\frac{\pi_\theta(a|s)}{\pi_{\theta_t}(a|s)}$, where ϵ is a small hyperparameter (typically set near 0.1) that determines the extent to which the policy can deviate from the previous policy. The purpose of the clipping is to prevent the policy from making too big of a change, which can lead to performance collapse. The clipping also ensures that the old policy data remains representative of the new policy if the policy ratio is significantly different from 1.

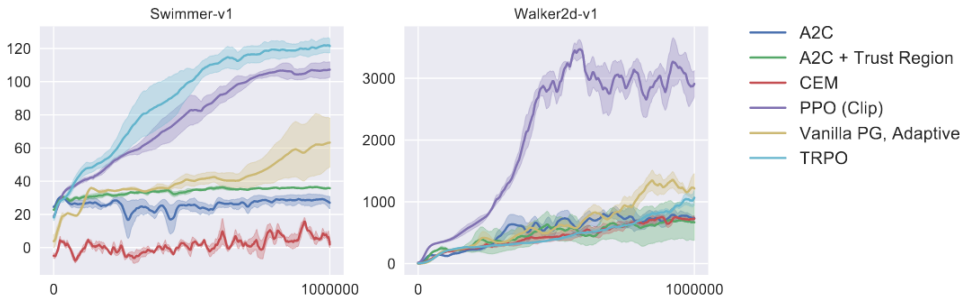

Figure 3.2: Comparison of several algorithms on two different environments. PPO exceeds the performance of TRPO and VPG. Image taken from [27].

3.7 Generalized Advantage Estimation

In the previous two sections we focused on the policy update, leaving details on how we can compute and learn the advantage $A_{\pi_{\theta_t}}(s, a)$.
There are many estimators of the advantage, for example in vanilla policy gradient (3.15) we used $A_{\pi_{\theta_t}}(S_t, A_t) = G_{t:t+1} - V_{\pi_{\theta_t}}(S_t; \boldsymbol{w})$ which is the TD-error, while in REINFORCE we used the full return G_t 3.11. In general, the policy gradient can be written in a more general way [28]:

$$\nabla_\theta J = \mathbb{E}\left[\sum_{t=0}^{\infty} \Psi_t \nabla_\theta \log \pi_\theta(A_t|S_t)\right] \tag{3.26}$$

Where Ψ_t can be any method of advantage estimation, like:

- $R_{t+1} + \gamma V_{\pi_{\theta_t}}(S_{t+1}; \boldsymbol{w}) - V_{\pi_{\theta_t}}(S_{t+1}; \boldsymbol{w})$ One-step TD target.
- $Q_{\pi_{\theta_t}}(S_t, A_t)$ Action-value function.
- $\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$ Full return.
- $A_{\pi_{\theta_t}}(S_t, A_t)$ Advantage function.

It can be proven that among these estimators the advantage function is the lowest variance estimator [29].

Another advantage estimation method that is used in practice is Generalized Advantage Estimation (GAE) [28]. GAE is a popular advantage estimation method that combines the strengths of both Monte Carlo and TD methods. The GAE advantage is expressed as:

$$A_t^{\text{GAE}}(s, a) = \sum_{k=0}^{\infty} \gamma^k \lambda^k \delta_{t+k} \tag{3.27}$$

Where δ_{t+k} is the TD-error at time $t+k$ and $\lambda \in (0, 1)$ is a hyperparameter that determines the weight given to future TD-errors. In particular, if $\lambda = 0$ then $A_t^{\text{GAE}} = \delta_t$, while if $\lambda = 1$ then $A_t^{\text{GAE}} = G_t$. The λ parameter thus makes a compromise between the bias and variance of the estimator, like the parameter n does for n-step TD methods 2.26. Figure 3.3 shows the performance of GAE with different values of λ.
Currently, GAE is used in all major reinforcement learning libraries, like CleanRL [30], Tianshou [31] and Stable Baselines3 [32].

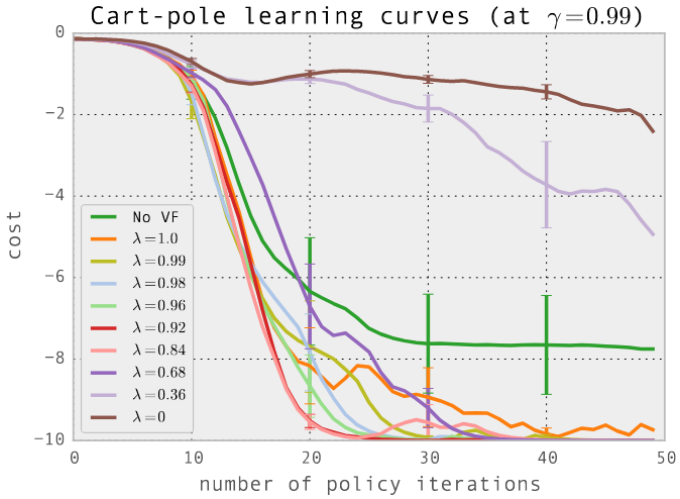

Figure 3.3: Comparison of GAE with various values of λ in the Cart-pole environment using TRPO to learn the policy. We can see that $\lambda \in [0.9, 0.99]$ performs best. Image taken from [28].

Chapter 4

Original work

In this chapter we explain the approach that was used to create a reinforcement learning agent that learns to play Briscola. The code implementation is available at `https://github.com/LetteraUnica/BriscolaBot`.

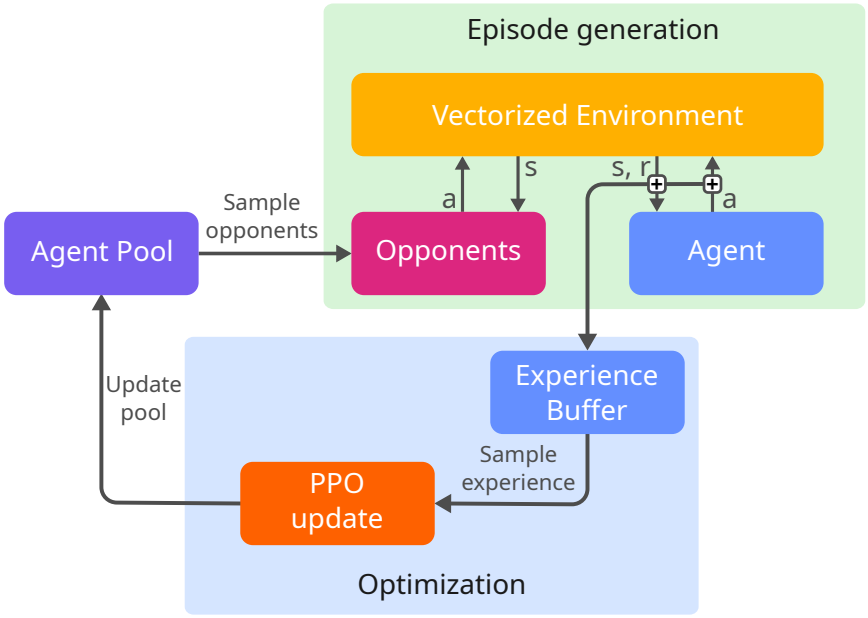

Figure 4.1: Starting from the Agent Pool, 4 opponents with frozen parameters are sampled. If the pool contains fewer than 4 agents, sampling is continued with replacement. The agent then plays 512 games against each opponent, resulting in a total of 2048 games (corresponding to 40960 tricks), which are played in parallel using a Vectorized Environment. During episode generation, the agent also collects experience and stores it in an Experience Buffer. Afterwards, the agent is trained on the collected experience using the PPO algorithm [27]. Finally, a copy of the updated agent is inserted into the Agent Pool with frozen weights. The process is repeated until the agent reaches the desired performance.

The general architecture 4.1 is composed of four main components, described in their respective sections in more detail:

- **Vectorized Environment:** Implements the Briscola environment and allows for parallel execution of multiple games.
- **Agent Pool:** Stores a pool of agents with frozen weights. The agents are sampled during training to play against the agent being trained.
- **Experience Buffer:** Stores the experience collected by the agent during gameplay.
- **Optimization:** The agent is trained on the experience collected during gameplay using the PPO algorithm [27].

After outlining the general architecture, we will explain the training process for the Briscola agent, along with the hyperparameters employed and performance evaluation. Ultimately, we will showcase the effectiveness of our approach by pitting the latest iteration of our agent, BriscolaBot-v3, against human players. To that end, interested readers can access the game at `https://replit.com/@LorenzoCavuoti/BriscolaBot`.

4.1 Environment implementation

To implement the environment we followed the guidelines of the PettingZoo library, which is a Python library for conducting research in multiagent reinforcement learning [33]. It is similar to OpenAI Gym, but it is designed for multiagent environments. We aimed to optimize the speed of the environment implementation to maximize the time spent in the training loop and minimize the time spent on executing game logic.

4.1.1 Agent observation

The agent has access to various information during gameplay, including the cards played so far, the cards in its hand, the Briscola card, the card on the table (if present), as well as its and the opponent's score. This information is represented in a vector with 162 components, as detailed in Table 4.1.

Feature	n. components
Cards played	40
Cards in hand	40
Briscola card	40
Table card	40
Agent points	1
Opponent points	1
Total	162

Table 4.1: Features used as input to the agent in Briscola. The first set of features consists of one-hot encoded vectors, with each element indicating a card. For example, if a card has been played, the corresponding element is set to 1, while if it hasn't, it's set to 0. The same goes for the cards in hand, the Briscola card, and the card thrown by the opponent. The last two features show the agent's and opponent's scores, normalized by the highest possible score in Briscola to keep them within the range [0, 1].

4.1.2 Reward structure

The reward structure is a weighted combination of two elements: win or loss and points earned in each turn:

$$R = \alpha R_{\text{win}} + (1 - \alpha) R_{\text{points}} \tag{4.1}$$

The reward structure consists of two terms: R_{win} and R_{points}. R_{win} is equal to 1 if the agent wins the game and 0 otherwise, and is only given at the end of the game. On the other hand, R_{points} are the points the agent gains in each turn normalized by the highest possible score in Briscola and are given at every turn. The relative importance of these two terms is controlled by α, which is a hyperparameter. Ideally, $\alpha = 1$ should give the best results as it prioritizes winning. However, this can also lead to high variance in the reward structure and sparse rewards, with only 0 or 1 being given at the end of the game. In practice, we started training with $\alpha = 0.1$ and the gradually increased it until we reached $\alpha = 1$. This allows to learn fast a the beginning of training and then focus on winning the game.

4.1.3 Action space

We explored two different ways to represent the action space. The first representation consisted of three discrete actions, corresponding to each of the three cards the agent could play, making it an intuitive choice that mimics how humans play the game. In this representation all actions are valid except when near to the end of the game, where the player has fewer than 3 cards in hand. The second representation consisted of 40 discrete actions, each representing a card from the deck. This representation is less intuitive and most of the actions, around 93%, are invalid. Despite these

limitations, the second representation outperformed the first (as shown in Figure 4.2). We believe this is because the agent doesn't have to consider the position of the cards in its hand and can instead focus solely on the cards it holds.

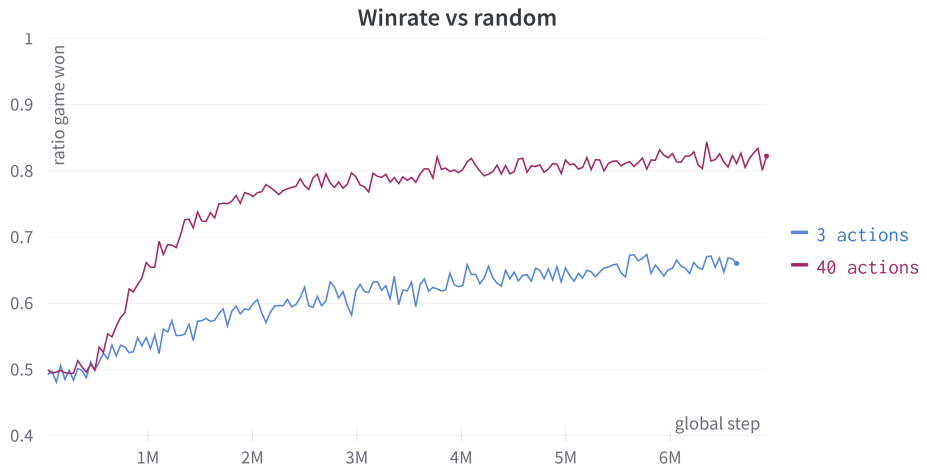

Figure 4.2: Performance of the two action space representations against a random player. The representation with 40 actions learns faster and reaches a win rate higher than 80%, while the 3 action representation learns more slowly and reaches a win rate of 65% by the end of training. In both representations we used invalid action masking as described in [34].

Invalid action masking

Both in the first and second representation of the action space, the agent must not play a card that is not in its hand. This can be ensured through different methods, such as penalizing the agent for playing an invalid action or masking the invalid actions. We opted to implement the latter approach, as it has been shown empirically to be superior to penalizing invalid actions [34].

The masking is done by setting the logits corresponding to the invalid actions to a small number, this way of performing the masking can be demonstrated to still produce a valid policy gradient as we are applying a differentiable state-dependent transformation to the policy [34].

$$\pi'(\cdot|s) = \text{softmax}(\text{mask}(l_i))$$

$$\text{mask}(l_i) = \begin{cases} l_i & \text{if } a_i \text{ is valid} \\ M & \text{otherwise} \end{cases}$$

Where l is the logits vector, a is the action vector, s is the state and M is a small number, in our implementation we used $M = -10^6$.

4.1.4 Vectorized environment

In order to make the training process faster, we utilized a vectorized environment. This is a wrapper around the original environment that allows multiple instances of it to run simultaneously. In this setup, the agent receives all observations from all environments at once, which enables batching of policy evaluation and potentially performing it on a GPU, resulting in a significant speed-up of the training process.

4.2 Agent

When using policy-gradient methods the agent consists of two parts, the Actor and the Critic. The Actor, implemented as a multi-layer perceptron (MLP) with ReLU activations, defines the policy by mapping the agent's observation to a probability distribution over the action space (Figure 4.3). On the other hand, the Critic uses the same MLP architecture as the Actor but outputs a single real value that represents the state's value (Figure 4.3). We used two separate neural networks for the Actor and Critic as suggested in [35].

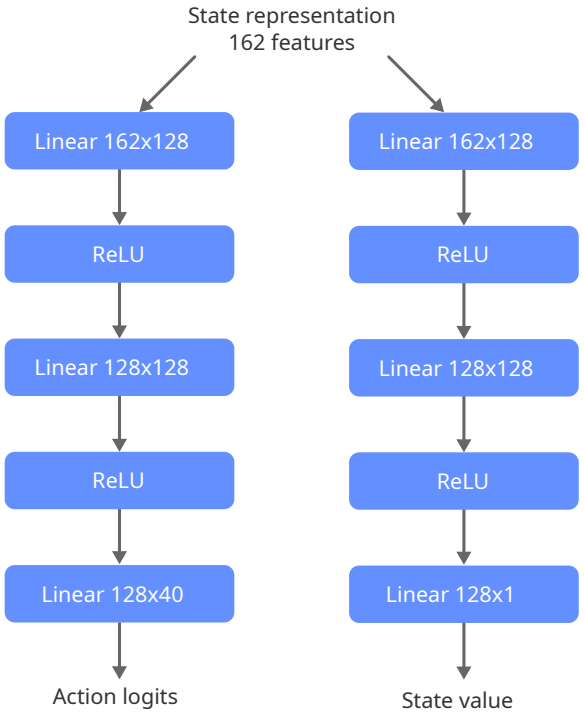

Figure 4.3: Diagram of the Actor-Critic Network Architecture. The Actor (left) transforms the agent's observation into a probability distribution over the 40 available actions, while the Critic (right) outputs a single real value, representing the state's value.

4.3 Agent Pool

The agent pool serves as a repository for past versions of the agent undergoing training, and keeps track of each version's win probability against the current agent. In detail, the pool holds a list of agents and their ratings E_i against the agent being trained[1]. When opponents are selected for training, the pool samples them based on their ratings, with higher-rated agents having a higher probability of being chosen. This sampling is done using the following formula:

$$P_i = \frac{e^{E_i}}{\sum_{j=1}^{n} e^{E_j}} \tag{4.2}$$

The ratings of the opponents in the agent pool are updated after the agent has played games against them, with the following equation:

$$E_i \to (1-\nu)E_i + \nu \operatorname{logit}(\bar{W}_i) \tag{4.3}$$

Where E_i represents the rating of opponent i, $\bar{W}_i$ is the average win rate of opponent i against the agent being trained. The hyperparameter ν determines the influence of the old rating E_i and the new rating $\operatorname{logit}(\bar{W}_i)$ on the updated rating. If $\nu = 1$, the updated rating would be an unbiased estimator of the true rating. However, this would result in high variance. To reduce the variance, $\nu = 0.1$ is used, which biases the estimator towards E_i, reducing its variance.

The agent pool is a crucial component as it prevents the agent from forgetting how to overcome previous versions of itself. With the agent pool, the agent is exposed to a diverse range of opponents, including those it has previously defeated, and must be able to overcome them once again, thus ensuring its overall improvement.

4.4 Optimization

The optimization process is done using a variant of the Proximal Policy Optimization (PPO) algorithm, which includes several modifications that have been shown to improve the performance of the algorithm [35]. Some of them are:

- The learning rate decreases gradually during training, a common practice in deep learning that has been shown to also enhance PPO's performance.

[1]The rating is very similar to the ELO system

- Generalized Advantage Estimation (GAE): The advantage function is calculated using GAE [28], reducing variance in the advantage function and shortening training time.
- Normalization of Advantages: After GAE is used to calculate the advantages, they are normalized at the mini-batch level to have mean zero and unit variance. This operation doesn't change the policy gradient, but it helps the optimization process as the data is centered around zero.
- Loss entropy bonus: The loss function is enhanced with an entropy bonus, encouraging exploration and preventing the policy from becoming too deterministic too early in training.

The optimization of the agent is performed using the Adam optimizer [36], with a starting learning rate of $3 \cdot 10^{-3}$, which gradually decreases to $1 \cdot 10^{-4}$ during the training process. To accelerate training, a large batch size of 1024 to 2048 is employed, as it has been observed to increase efficiency, and it aligns with the typical use of large batch sizes in reinforcement learning [37]. The agent's performance is optimized by minimizing the following loss function.

$$L = L_{\text{policy}} + \beta_{\text{v}} L_{\text{value}} - \beta_{\text{e}} L_{\text{entropy}} \tag{4.4}$$

In equation (4.4), L_{policy} represents the PPO policy loss as in equation (3.25), L_{value} is the mean squared error between the predicted value and the actual episode return, and L_{entropy} is the entropy of the policy. The hyperparameters β_{v} and β_{e} control the relative importance of these terms and are commonly set around 0.5 and 0.01 respectively.

4.5 Training Procedure

We trained three agents with increasing level of performance using an incremental approach, similar to the one used when developing chess engines [3, 4]. More in detail, when developing the next version of Stockfish, the developers measure its performance against the previous version, if the new version wins more games than the previous version, it is considered an improvement. This is a simple yet powerful approach as it allows to test the agent improvement without having to define an ELO rating, which can be hard to do, as it depends on the distribution of the opponents.

4.5.1 BriscolaBot-v1

The first version of the agent didn't use self-play with the agent pool, instead was only trained against a random opponent for 10 million steps.

Before starting the final training run, however, we performed an extensive hyperparameter search using the Weights & Biases bayesian search method [38]. The algorithm was tasked to maximize the win probability against a random opponent in 1 million training steps. The method successfully discovered hyperparameters that reduced the time taken to reach an 80% win rate against a random agent by half, as depicted in Figure 4.4. The optimized hyperparameters are displayed in Table 4.2.

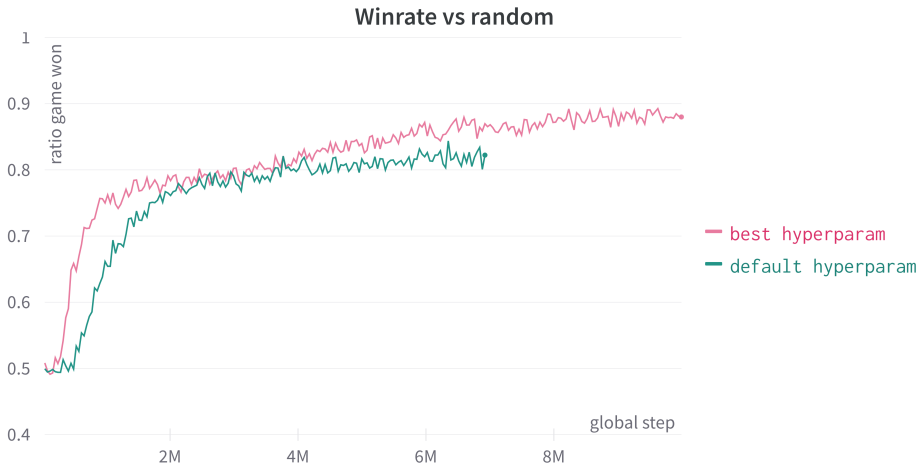

Figure 4.4: Comparison of the default hyperparameters and the best hyperparameters found using Weights & Biases bayesian search method.

Hyperparameter	Default value	Best value
Learning rate	0.001	0.003
Clip coefficient (3.25)	0.2	0.3
Entropy bonus (4.4)	0.001	0.01
GAE λ (3.27)	0.95	0.9
Discount factor γ (2.3)	1	1
Reward for win (4.1)	0	0.1
Update epochs	4	2

Table 4.2: Best hyperparameters found using Weights & Biases bayesian search method. The default parameters were taken from the CleanRL implementation of PPO [30] which provide a good starting point for reinforcement learning tasks. Update epochs is the number of passes over the collected episode data the optimization process performs.

The final training run, which produced BriscolaBot-v1, was performed using the optimized hyperparameters. The agent was able to achieve a win rate of 80% against a random agent in 1 million steps and almost 90% in 10 million steps, as shown in figure 4.4.

4.5.2 BriscolaBot-v2

In the second version of the agent, we added self-play with the agent pool, because the agent manages beat a random opponent very quickly. Furthermore, we increased the neural network size to 256, changed the activation function from ReLU to Mish [39] and added learning-rate decay. The effect of some of these changes can be seen in figure 4.5. With these changes the new agent was able to beat BriscolaBot-v1 60% of the time when trained for 10 million steps, which is a very good result considering the randomness of the game of Briscola.

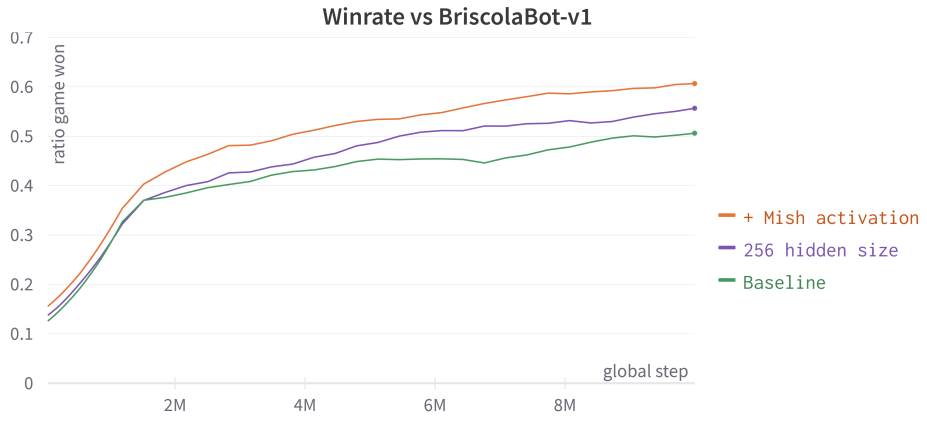

Figure 4.5: Winrate against BriscolaBot-v1 when increasing the neural network size from 128 to 256 (purple line) and on top of that changing the activation function from ReLU to Mish (orange line).

To produce the final version of the BriscolaBot-v2 agent we continued training until 20 million steps. BriscolaBot-v2 was able to achieve a win rate of 90% against a random agent and 64% against BriscolaBot-v1.

4.5.3 BriscolaBot-v3

A problem with BriscolaBot-v2 is that it throws the ace and the 3 of briscola very often, instead of waiting for the right opportunity. This is a problem, because the ace and the 3 of briscola are the most powerful cards in the game, and wasting them can lead to a loss.

We think that this problem is caused by giving too much importance to the number of points scored in the loss function, which, combined with $\gamma = 0.95$ causes the agent to collect points as soon as possible. In this version, we tried to address this problem by gradually increasing the ν parameter in

equation (4.1) from 0.1 up to 1 during training. Another change that we applied was to decay the entropy β_e parameter in equation (4.4) from 0.01 down to approximately zero, as this is an exploration parameter and should decrease when reaching optimal play [2]. The effect of these changes can be seen in figure 4.6.

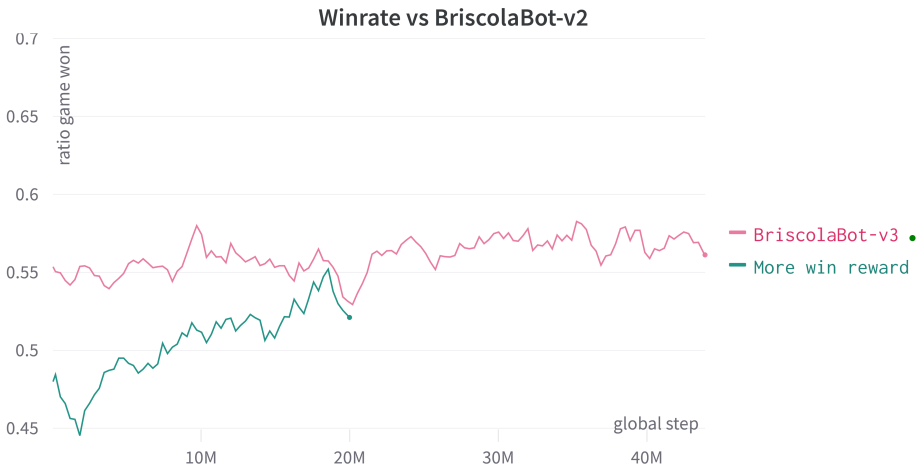

Figure 4.6: Winrate against BriscolaBot-v2 when gradually increasing the ν parameter in equation (4.1) and decreasing the entropy during training, we can see that these two changes improved the performance of the agent. Since these two runs started from the weights of BriscolaBot-v2, the ratio of games won starts near 0.5.

The final agent was trained starting from the BriscolaBot-v2 weights, with the modifications listed above for 50 million steps, corresponding to approximately 4 hours of training on an intel i5-4690K CPU. The agent was able to achieve a win rate of 57% against BriscolaBot-v2 and 93% against a random agent, surpassing previous approaches to the problem [9]. The final list of hyperparameters used for the final agent is shown in Table 4.3.

Hyperparameter	Value	Hyperparameter	Value
Learning rate	0.003	Learning rate decay	0.997
Clip coefficient	0.3	Batch size	2048
Discount factor γ	1	GAE λ (3.27)	0.9
Entropy bonus β_e (4.4)	0.01	β_e decay	0.998
ν (4.1)	0.1	ν increase	0.001
Update epochs	2	hidden network size	256

Table 4.3: Hyperparameters used for the final agent.

4.6 Human evaluation

We evaluated the agent's performance by pitting it against human players using a website we developed, which can be accessed at `https://replit.com/@LorenzoCavuoti/BriscolaBot`. The results of the human evaluation are presented in Table 4.4.

Opponent	Games played	Games won	Games lost	Win probability
Me	0	0	0	0
Antonio	3	0	3	1
Emanuele	3	3	0	0
Simone	39	19	20	0.5
Andrea	39	19	20	0.5
Ilaria	39	19	20	0.5
Teresa	39	19	20	0.5
Giacomo	39	19	20	0.5
Total	45	22	23	0.5

Table 4.4: Results from the human evaluation.

The agent was able to beat human players more than 50% of the time. Simone found the agent quite frustrating to play against as he said: "I was up 19 to 16 and then he won 4 games in a row, beating me with a final score of 19-20". Furthermore, Simone added that the agent seemed to predict his hand cards, because sometimes it threw a briscola when he had his hand full of aces and threes of different seed than briscola.

While the agent is performing well, there is still room for improvement, as it occasionally makes suboptimal moves. For instance, it sometimes chooses to throw the 7 of briscola instead of taking the trick with the 4 of briscola, which would have allowed it to save the 7 for later use.

Chapter 5

Discussion

In this thesis, we proposed a novel approach to solving the game of Briscola by using the PPO algorithm with function approximation to learn the optimal policy. To evaluate the performance of our latest agent, BriscolaBot-v3, we developed a website `https://replit.com/@LorenzoCavuoti/BriscolaBot` where the agent could play against human players. Our results showed that BriscolaBot-v3 achieved a 50% win rate against average human players, confirming the effectiveness of our method. Moreover, the agent was trained on a standard 4-core computer, ensuring reproducibility of our results and making it feasible to train a competitive agent on any machine in a reasonable amount of time.

5.1 Future Work

We have identified several potential improvements to further enhance the performance of the agent, including:

- **Exact solving of endgame positions:** when all cards have been extracted the cards of the opponent are known, so it becomes possible to solve the game of Briscola exactly with minimax search. This improvement can be achieved at a low computational cost, since the number of possible ways of playing out a given endgame is only 36, which can be further reduced through the application of dynamic programming techniques.
- **Improve agent performance with MCTS:** previous methods in the game of Briscola used Monte Carlo Tree Search (MCTS) to identify the best move [13, 40]. However, these approaches relied on a model of the opponent rather than learning from it, such as assuming the opponent throws a random card from the set of unseen cards. We suggest improving this approach by training a model that predicts the

opponent's possible move in a given situation, which can then be used to perform MCTS search.

- **Approximating the Nash equilibrium:** we can use imperfect information game-solving approaches to directly find or approximate the Nash equilibrium of the game of Briscola, similar to how they have been successfully used in poker to beat the world's best players [15].
- **Extend the observation space of the agent:** currently, the agent only observes the cards that have been played, but not the order in which they were played. By observing the order in which cards were played, the agent could potentially better predict the cards that the opponent has in their hand. For example, if the agent plays an ace of cups and the opponent does not use briscola to take the trick, it is likely that the opponent does not have any briscola in their hand. This information could allow the agent to be more aggressive in the following tricks.

Possible other directions for future work include applying the same technique to other Italian card games, such as Scopa, Tressette, Macchiavelli, and Scala 40. Additionally, the approach can be extended to Briscola when played with multiple players, as the game can be played by up to six players in teams of two or three.

Bibliography

[1] Wikipedia, *Briscola — wikipedia, the free encyclopedia*, 2023. [Online]. Available: `https://en.wikipedia.org/wiki/Briscola`.

[2] C. Berner *et al.*, "Dota 2 with large scale deep reinforcement learning," *CoRR*, vol. abs/1912.06680, 2019. arXiv: `1912.06680`. [Online]. Available: `http://arxiv.org/abs/1912.06680`.

[3] S. D. Team, *Stockfish*, `https://stockfishchess.org/`, 2023.

[4] T. L. Team, *Lc0: Open-source neural network chess engine*, `https://lczero.org/`, 2023.

[5] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 4th. Pearson Education Limited, London, 2021.

[6] Wikipedia, *Nash equilibrium — wikipedia, the free encyclopedia*, 2023. [Online]. Available: `https://en.wikipedia.org/wiki/Nash_equilibrium`.

[7] V. Mnih *et al.*, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[8] fezriva, *Briscola*, `https://github.com/fezriva/briscola`, 2022.

[9] alsora, *Deep-briscola*, `https://github.com/alsora/deep-briscola`, 2019.

[10] D. Silver *et al.*, "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," *arXiv preprint arXiv:1712.01815*, 2017.

[11] D. Silver *et al.*, "Mastering the game of go without human knowledge," *nature*, vol. 550, no. 7676, pp. 354–359, 2017.

[12] G. Tesauro *et al.*, "Temporal difference learning and td-gammon," *Communications of the ACM*, vol. 38, no. 3, pp. 58–68, 1995.

[13] S. Gazda, "A game-playing algorithm for briscola," 2021.

[14] D. Silver *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[15] N. Brown and T. Sandholm, "Superhuman ai for heads-up no-limit poker: Libratus beats top professionals," *Science*, vol. 359, no. 6374, pp. 418–424, 2018.

[16] 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.

[17] N. Brown, A. Bakhtin, A. Lerer, and Q. Gong, "Combining deep reinforcement learning and search for imperfect-information games," *Advances in Neural Information Processing Systems*, vol. 33, pp. 17 057–17 069, 2020.

[18] L. Ouyang *et al.*, *Training language models to follow instructions with human feedback*, 2022. DOI: 10.48550/ARXIV.2203.02155. [Online]. Available: https://arxiv.org/abs/2203.02155.

[19] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd edition. MIT Press, 2018.

[20] L. Buşoniu, R. Babuška, and B. De Schutter, "Multi-agent reinforcement learning: An overview," in *Innovations in Multi-Agent Systems and Applications - 1*, D. Srinivasan and L. C. Jain, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 183–221, ISBN: 978-3-642-14435-6. DOI: 10.1007/978-3-642-14435-6_7. [Online]. Available: https://doi.org/10.1007/978-3-642-14435-6_7.

[21] O. Vinyals *et al.*, "Grandmaster level in starcraft ii using multi-agent reinforcement learning," *Nature*, vol. 575, no. 7782, pp. 350–354, 2019.

[22] X. Wang *et al.*, "Scc: An efficient deep reinforcement learning agent mastering the game of starcraft ii," in *International conference on machine learning*, PMLR, 2021, pp. 10 905–10 915.

[23] B. Baker *et al.*, *Video pretraining (vpt): Learning to act by watching unlabeled online videos*, 2022. DOI: 10.48550/ARXIV.2206.11795. [Online]. Available: https://arxiv.org/abs/2206.11795.

[24] D. Hafner, J. Pasukonis, J. Ba, and T. Lillicrap, *Mastering diverse domains through world models*, 2023. DOI: 10.48550/ARXIV.2301.04104. [Online]. Available: https://arxiv.org/abs/2301.04104.

[25] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

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

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

[28] 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.

[29] E. Greensmith, P. L. Bartlett, and J. Baxter, "Variance reduction techniques for gradient estimates in reinforcement learning.," *Journal of Machine Learning Research*, vol. 5, no. 9, 2004.

[30] S. Huang *et al.*, "Cleanrl: High-quality single-file implementations of deep reinforcement learning algorithms," *Journal of Machine Learning Research*, vol. 23, no. 274, pp. 1–18, 2022. [Online]. Available: `http://jmlr.org/papers/v23/21-1342.html`.

[31] J. Weng *et al.*, "Tianshou: A highly modularized deep reinforcement learning library," *Journal of Machine Learning Research*, vol. 23, no. 267, pp. 1–6, 2022. [Online]. Available: `http://jmlr.org/papers/v23/21-1127.html`.

[32] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann, "Stable-baselines3: Reliable reinforcement learning implementations," *Journal of Machine Learning Research*, vol. 22, no. 268, pp. 1–8, 2021. [Online]. Available: `http://jmlr.org/papers/v22/20-1364.html`.

[33] J. K Terry *et al.*, "Pettingzoo: Gym for multi-agent reinforcement learning," *arXiv preprint arXiv:2009.14471*, 2020.

[34] S. Huang and S. Ontañón, "A closer look at invalid action masking in policy gradient algorithms," *arXiv preprint arXiv:2006.14171*, 2020.

[35] S. Huang, R. F. J. Dossa, A. Raffin, A. Kanervisto, and W. Wang, "The 37 implementation details of proximal policy optimization," in *ICLR Blog Track*, https://iclr-blog-track.github.io/2022/03/25/ppo-implementation-details/, 2022. [Online]. Available: `https://iclr-blog-track.github.io/2022/03/25/ppo-implementation-details/`.

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

[37] S. McCandlish, J. Kaplan, D. Amodei, and O. D. Team, "An empirical model of large-batch training," *arXiv preprint arXiv:1812.06162*, 2018.

[38] L. Biewald, *Experiment tracking with weights and biases*, Software available from wandb.com, 2020. [Online]. Available: `https://www.wandb.com/`.

[39] D. Misra, "Mish: A self regularized non-monotonic activation function," *arXiv preprint arXiv:1908.08681*, 2019.

[40] A. VILLA, "Algoritmi di ricerca ad albero monte carlo applicati all'intelligenza artificiale nel gioco della briscola a cinque," 2013.