Reinforcement Learning Project: RL for the darts game

Reinforcement Learning
Spring Semester 22

Antoine Demont
Boris Mottet

Joint Master of Computer Sciences
Switzerland
June 5, 2022

Contents

Chapter 1

Introduction

This project takes place in the context of a Master's course on Reinforcement learning at the university of Neuchâtel. The goal was to create a custom gym environment from *openai* [4] and apply different reinforcements algorithms on it.
These algorithms are then tested through some experiments to evaluate their performance. In our case, we decided to create an environment representing a game of darts.
This report is split in the following sections:

1. **Introduction**: This contains a brief introduction to the project as well as a refresher of the game of darts' rules.
2. **Environment**: This contains a description of the environment, how it relates to Openai Gym and how it was adapted to the rules of dart.
3. **Algorithms**: This explains which algorithms were chosen to train on the dart environment and how it applies to it.
4. **Experiments**: This analyse how the implemented algorithms performed on the environment.
5. **Conclusions**: This contains a conclusion on this project.

1.1 About the game of darts

The game of darts is widely spread and the basics are known by a great part of the population, the circular board (fig. 1.1) is composed of twenty numbers and the bullseye. For each number, two smaller sized cells give a point multiplier of 2 and 3 for the outer and inner one respectively. The bullseye if worth 25 points and its red part acts as a times 2 multiplier, identically to the numbered cells.

This distinction is important with the rule set in mind. Each player starts at a score of 501 and can throw three darts. To win, a player must reach exactly zero and finish with a times 2 multiplier, hence the precision on the bullseye counting as a double. If a player were to reach a score of 1 or below, or end on a cell with the wrong multiplier, their throw is ignored and their turn ends directly regardless of the number of darts thrown this turn. This subtlety makes it harder for an agent to learn to play darts because it cannot aim only to the cell that rewards the most points but must set itself up for the end of a leg[1]. Indeed, to be at 40 at the beginning of one's turn is way better than to be at 51 ; in the first case, a player can aim to 20 times 2, and if they score only 20 without the multiplier, they can still win with a 10 times 2.

[1] A leg is a game of dart from 501 to 0 points. A match can be made of multiple legs.

Figure 1.1: Standard dart board, taken from `https://www.nicepng.com/downpng/u2w7r5i1t4a9w7i1_darts-board-wall-sticker-dart-board-cork-wall/`

Chapter 2

Environment

In this chapter, we will present the environment we created.

2.1 Openai gym

Gym is an API widely used for reinforcement learning. This API contains a set of pre-implemented environment accessible with simple line of code. In listing 2.1, we see how to access the *LunarLander-v2* environment already contained in the gym.

```
1 import gym
2 env = gym.make("LunarLander-v2")
```

Listing 2.1: Import and creation of a gym environment

The main strength of this API is that it is extensible, meaning that any developer is able to create their own gym environment using the base functionalities of gym. To do so, one only has to create a class extending the default implementation and then override the following functions:

- **__init__**: This method is responsible for the declaration and initialization of the different variables used. The most important ones are the actions and observation spaces used to indicate the set of possible actions an agent could take and the observations from which could result.
- **reset**: This method is used to re-initialize any variable modified during the course of an episode. This will ensure that every episode starts from a correct initial state.
- **step**: This method contains the logic of the environment. It processes the action taken by the agent and computes the correct observation that the environment should return.
- render (optional): This method uses the PyGame library to render the environment and display its state to the user.

The environment can then be registered to the API and accessed locally by the same way as presented in the listing 2.1.

2.2 Our environment

At creation of our custom darts environment, the user is presented to a few choices:

- n_players: This variable contains the number of players in the game. Each number greater than 1 will fill the game with opponents for the agent to compete with.
- players_level: This variable is a list indicating the capacity of players to do the best possible shot, where each index i contains the level of player i. The level values go from zero the worst player to five the best.
- opponent_mode: This field stores a string indicating the way opponents choose the shot they make. This can be either *best*, meaning that they try to do the best possible shot, or *random*, meaning that they take random shots.

The game's board is represented by an array containing an integer value for each cell representing the number of darts thrown at this position. The player can then choose to throw at any cell, while keeping in mind that a cell already containing a dart will have a risk of resulting in a miss resulting in a score of 0 for the player. This means that the action space takes a discrete value between 0 and 82. This upper bound is obtained by multiplying the 20 cells by the possible multipliers, in order 2, 1, 3 and 1, with the addition of the bullseye ($20 \cdot 4 + 2$).

For the observation space, the environment returns in each observation a dictionary containing the board state, the player scores and the remaining shots. By simple combination, this could result in a huge amount of states. We need some kind of conversion for our algorithms, this will be treated more in depth in the algorithms part of the report.

2.2.1 Penalty system

As mentioned before, we included a penalty system (table 2.1). This is necessary to balance the risk-to-reward ratio. Otherwise, there would be no interest in not hitting a triple twenty at each shot. For example, a player of level 2, wanting to hit their third shot in a triple twenty would have $2 \cdot 1 + 7.5 + (5 - 2) = 12.5\%$ chances to miss.

Missing can mean three things: A dart is hit, resulting in a score of 0, the throw is deviated vertically, resulting in a different multiplier, or the throw is missed horizontally, resulting in a different number being hit. To continue with our previous example, the player with a 12.5% chance to miss the triple twenty could end up with a score of 0, 20, 1 or 5 (with or without the triple multiplier) depending if the player misses to the right or left, 5 and 1 being the neighbouring cells to 20, see figure 1.1.

Cause	**Penalty** (probability to miss)
Dart in the cell	1% per dart
Throwing at a double	5%
Throwing at a triple	7.5%
Throwing a the bullseye	2.5%
Throwing a the double bullseye	15%
Level of the player	(max level - 1)% per level

Table 2.1: Penalties and their conditions

2.2.2 Reward System

To motivate the agent to win before other players, the reward of 100 is given if a game is won and a reward of -100 if the game is lost. Moreover, since to win a player must reach zero with times 2 multiplier, under penalty of having the last shot cancelled and the turn ended, a reward of -10 is given after such a shot.

2.2.3 Finding the best shot

This method (listing 2.2) returns the best possible shot based on a player's score. What it does is check whether the player is in winning range, meaning that they have a score of 50 or below 41 due to the condition of ending on a double. It that is not the case, the player focuses on doing the greater amount possible of points. If that is the case then they either try to make their score even, such that they can end on a double, or directly make the exact score needed to end the game while still having a probability to miss due to the penalties presented in table 2.1.

```
1   def find_best_shot(self, level, player_idx, shots):
2       # find the best possible shot. If the player is in range of winning then the best shot
            is the one making them win, otherwise we assume the best shot is
3       # the one reducing their score by the largest value
4       all_scores = POSSIBLE_SCORES
5       if (self.is_winning_range(player_idx)):
6           # Winning shot should be a double
7           if self.players_score[player_idx] % 2 == 0:
8               cell_to_win = self.players_score[player_idx] / 2
9               cell_to_win = BULLSEYE if cell_to_win > 20 else cell_to_win
10              score = self.compute_score(
11                  cell_to_win, 2, MULTIPLIERS.index(2), self.players_level[player_idx])
12              return score
13          else:
14              # if the player cannot end on a double, it should aim for an even score to end on
                    the next
15              all_scores = [x for x in all_scores if x <
16                            self.players_score[player_idx] and x % 2 == 1]
17
18      # Opponents also have the possibility to miss
19      miss = shots * DART_PENALTY
20      if random.random() < miss:
21          return 0
22      limit = (1 + MAX_LEVEL - level) * SHOTS_RANGE
23      shot = random.choice(all_scores[:limit]) if limit < len(
24          all_scores) + 1 else random.choice(all_scores + [0])
25      return shot
```

Listing 2.2: Method finding the best possible shot

Chapter 3

Algorithms

In this chapter, we present the algorithms trained on the darts environment. Multiple algorithms are presented to see if they have different behavior on the problem.

As mentioned in the previous chapter, a simple combination of the observation state would create millions of states which could become problematic. To avoid this, we decided to group the greater values of the player's score into two separate states (fig. 3.1), since in this case, one throw is not very different and keep a large number of states for the endgame where decisions are more crucial. In addition to this, we multiply this score value with the positions of previously thrown darts. The boundaries were set arbitrarily by us.

For example, the state for a triple 20 with one dart previously thrown in this cell and a score of 250 will then be

$$s = 101 \cdot 20 \cdot 3 = 6060$$

.

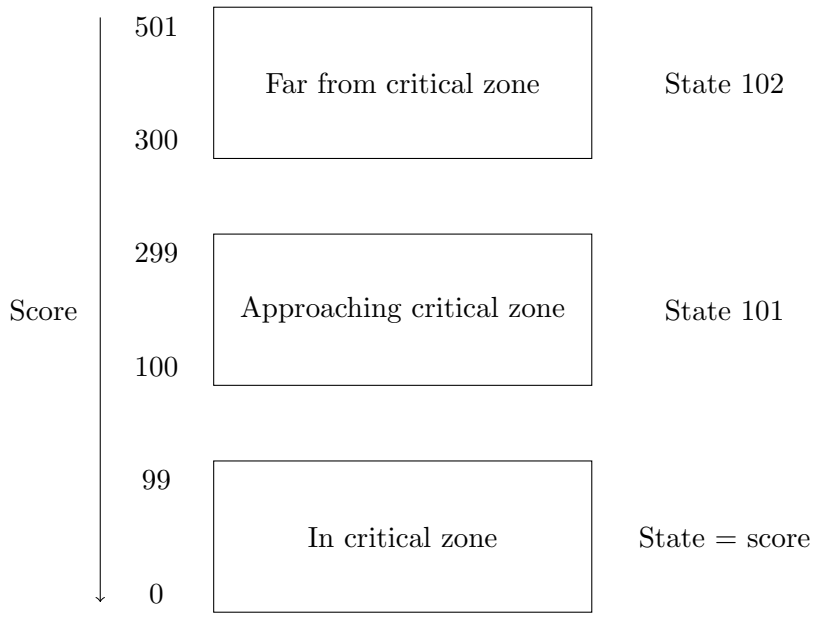

Figure 3.1: States distribution

3.1 Q-Learning

Q-learning is a classical algorithm in reinforcement learning. It aims at computing the function Q which is the expected reward for an action taken in a given state. This is done with a value iteration on Q according to this equation :

$$Q^{new}(s_t, a_t) = \alpha(r_t + \gamma \max_a Q(s_{t+1}, a_t)) + (1 - \alpha)Q(s_t, a_t) \tag{3.1}$$

With enough iteration, this Q can be accurate enough to approximate the optimal policy. The best action a to take at state s is the one that maximize $Q(s, a)$.

3.2 SARSA

The SARSA algorithm is very similar to Q-learning but also takes into account the next action. Where (s_t, a_t, r_t, s_{t+1}) are used with Q-learning, it is $(s_t, a_t, r_t, s_{t+1}, a_{t+1})$ that interests SARSA, thus its name. The update equation for Q function is :

$$Q^{new} = Q(s_t, a_t) + \alpha[r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \tag{3.2}$$

This algorithm is interesting for the dart environment because it is more concerned with the next move, which allow for better set-ups regarding the score. Indeed, it should take into account the next shot when updating the Q function.

3.3 Curiosity-Driven Learning

Since the rewards are rather sparse, being received only near the end of a leg, the curiosity-driven learning as presented by Deepak Pathak et al.[5] could perform well on the darts environment. With such a learning, the model could look for the best set-ups in the critical zone in order to be in good position in the winning range. Indeed, the curiosity-driven learning introduces the concept of intrinsic and extrinsic rewards. The extrinsic reward is given by the environment ; in the darts environment, it is given when an agent overflow by reaching zero without a 2 times multiplier or being at score 1 or below, or when a game end by a loss or a win. In the best case, the reward is given only once per game and rarely more which makes the extrinsic reward pretty sparse. The intrinsic reward is computed by the Intrinsic Curiosity Module (ICM). This module contains a forward model and an inverse model as shown in figure 3.2. The intrinsic reward is computed with the difference between the predicted feature of next state and the actual feature of the next stage. This encourage this agent to explore the environment better.

The implementation was inspired by this github[3]. The forward and inverse models can be implemented with the help of keras. The module takes as arguments a state, the next state and the action taken to go from one state to the next. Due to the potential inhomogeneity of the inputs, it is important to preprocess it well. Unfortunately, due to an error in the base code that we couldn't manage to solve, we could not train this model.

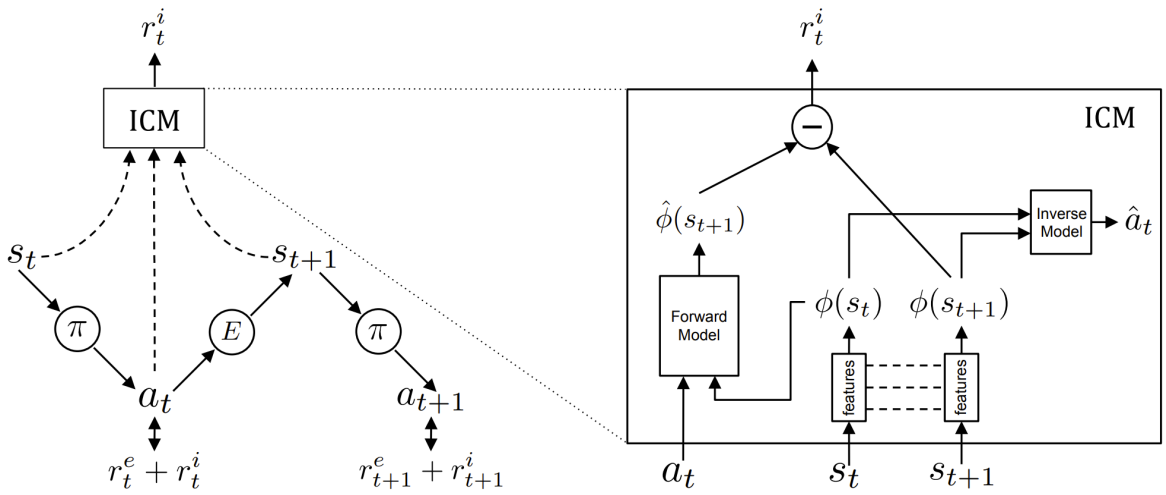

Figure 3.2: ICM from Deepak Pathak et al.[5]

Chapter 4

Experiments

For each of the previously presented baseline algorithms, we ran three experiments to tune and evaluate their performance. The first one highlighted the influence of the number of players in the game as well as give an optimal set of parameters for the algorithm. The second one ran a comparison on the action taken with the parameters discovered at the previous step and the optimal shot calculated inside of our environment. The third and last experiment focused on the winrate of the agent against opponents.

4.1 Number of players

For this experiment, we look at the influence other players have on the agent. The main reason for this experiment is that when the agent plays alone, it is able to play until it reaches the target score of 0 and then receive the maximal reward for winning the game. With opponents, the game can be terminated before this winning state since any opponent can reach the winning score, thus making the agent loose.

For this experiment, we initially set values for $\alpha, \epsilon, \gamma = 0.1$ and iteratively increase these values by 0.1. The first training step is done on α. Then the same process is applied to ϵ with the newly found optimal value for α and finally to γ with optimal α, ϵ.

The value with the greater total reward is taken as the optimal value. We can see this process in the listing 4.1. The last parameter is the decay for the γ value used during training and is fixed before the experiment based on the number of episodes and a minimal value of 0.1.

```
1
2 ----------------------------------
3 Max reward: 116100.0, with parameters: (0.1, 0.30000000000000004, 0.9, 0.9977000638225533),
      while searching alpha
4 ----------------------------------
5 ----------------------------------
6 Max reward: 124480.0, with parameters: (0.1, 0.1, 0.9, 0.9977000638225533), while searching
      epsilon
7 ----------------------------------
8 ----------------------------------
9 Max reward: 140810.0, with parameters: (0.1, 0.30000000000000004, 0.1, 0.9977000638225533),
      while searching gamma with optimal parameters
10 ------------------------------------------------------------------
11 Max reward: 148860.0, with parameters: (0.1, 0.1, 0.1, 0.9977000638225533), optimal parameters
12 ----------------------------------
```

Listing 4.1: Logs from the tuning experiment

With this first experiment, we already start to see some limitations to our model. The game of darts might be too simple for the reinforcement learning approach as the optimal steps can be easily computed with a simple algorithm. If for a single player setting the agent learns and end up winning in every case (4.1). For a multiplayer setting we see in the figure 4.2 that after a first increase in performance in the first episodes, the agent is not able to find a consistent way of beating its opponents on the first try and thus getting the perfect reward of 100.

This problem is even greatly amplified when the opponents take the best possible actions (fig. 4.3). Here the agent is rarely able to get a positive reward.

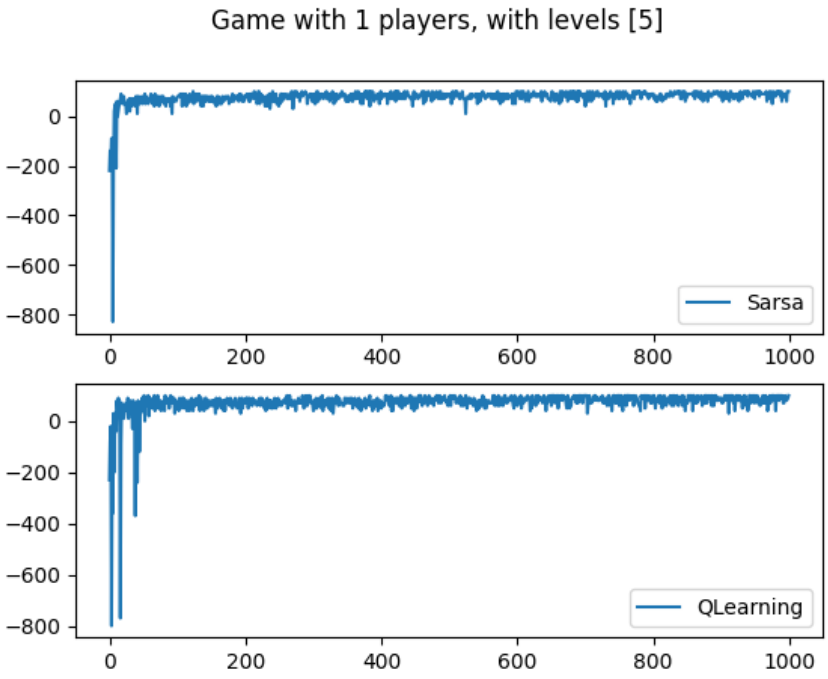

Figure 4.1: Reward distribution for 1000 experiments without opponents

4.2 Difference with the optimal score

Since we had to implement a way to find the optimal shot, using it as an upper bound comparison with the agent choice was almost evident.

For this, we computed the difference between the optimal score and the action taken by the agent and plotted it at each step of the training process (4.4). We also stored the average distance with the best shot (4.2).

This shows that the approach taken by the agent is quite different from the one we found optimal. This could be explained by the fact that our method of finding the optimal shot is not really tested and can surely be improved. We, for example, found the corner case where a player with a score of 51 could be in a bad situation. Since they are above the winning threshold, they will try to do the biggest score possible but this could result in a negative score, resetting the score to 51 and ending their turn. But this algorithm still outperforms the agent as seen previously.

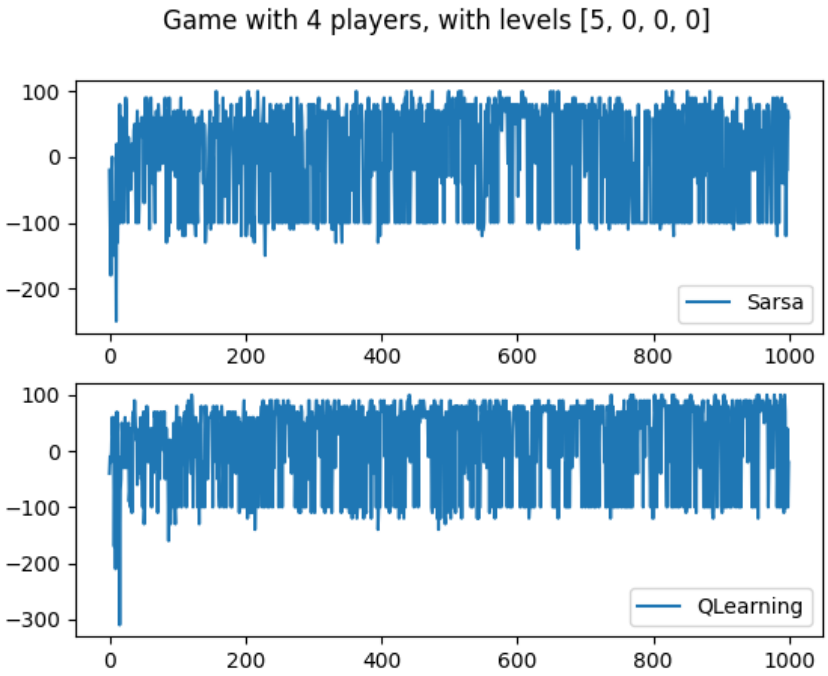

Figure 4.2: Reward distribution for 1000 experiments against 3 players with random actions

```
Average distance to best shot with Sarsa: 9.965612412352678
Average distance to best shot with QLearning: 9.219041446112852
```

Listing 4.2: Average distance to best shot

4.3 Win rate

In this last experiment, we compare the ratio of games won by the agent in games against 1 to 4 opponents taking either random action or the best possible shots.

We see that against opponents with random actions, the agent is able to win most of the games. Showing that even if it does not perform as good as expected, it still is better than random actions (figs. 4.5 and 4.6).

For the situation where the opponents take the best actions possible, there the agent has no chance to compete (figs. 4.7 and 4.8). Confirming that a reinforcement learning approach to this project might be over engineered and a simple algorithmic approach is more than sufficient.

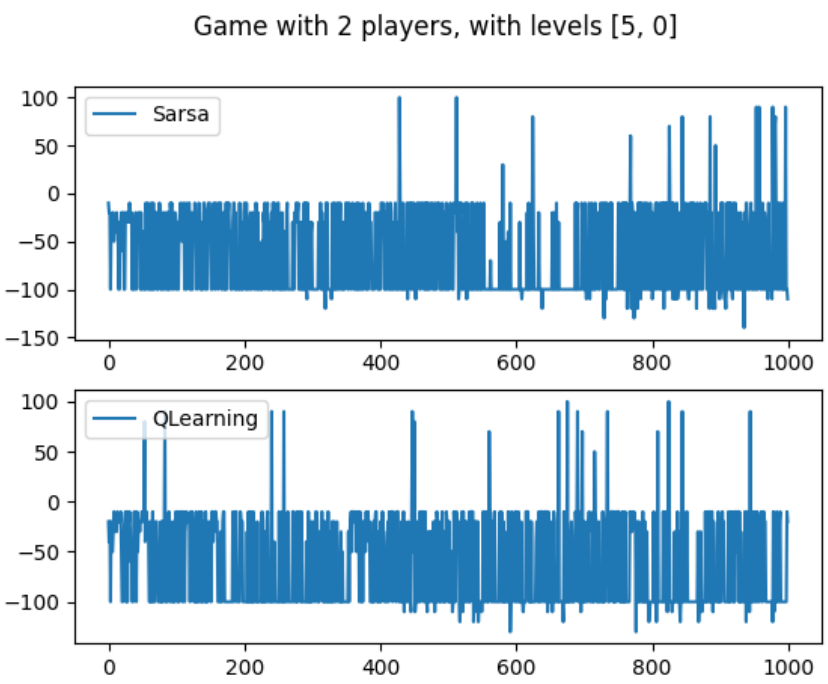

Figure 4.3: Reward distribution for 1000 experiments against 1 player with best actions

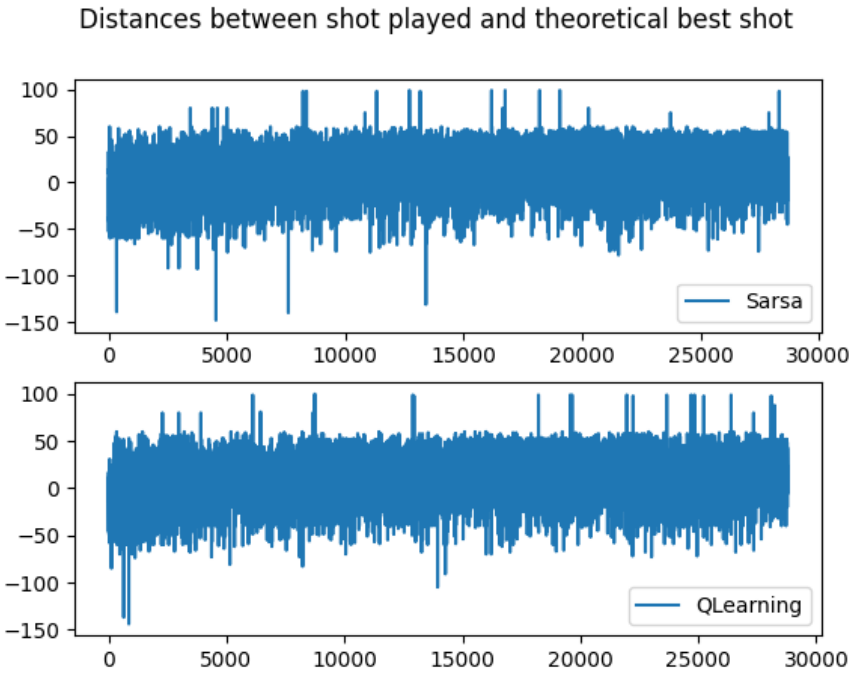

Figure 4.4: Difference between the optimal shot and action of the agent

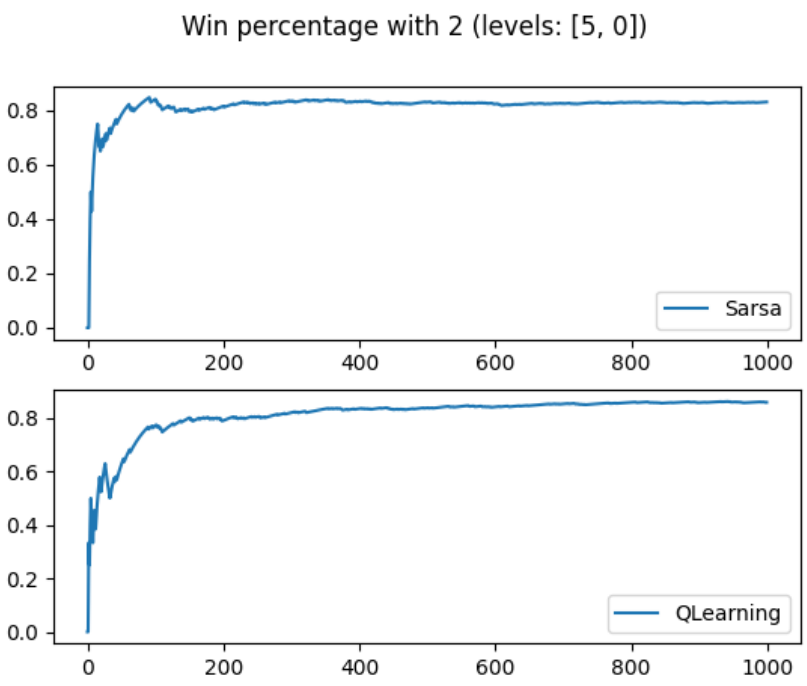

Figure 4.5: Win rate against one opponent with random actions

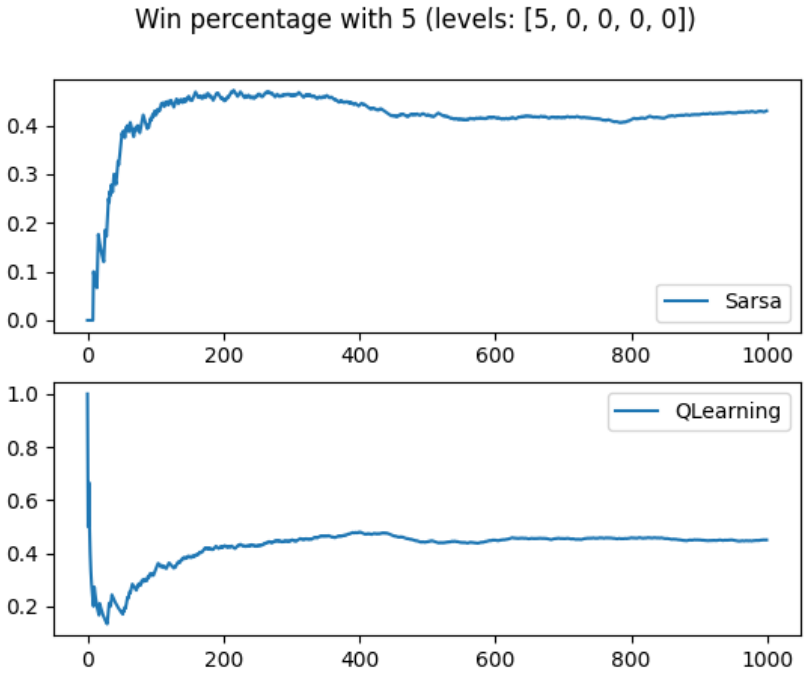

Figure 4.6: Win rate against 4 opponents with random actions

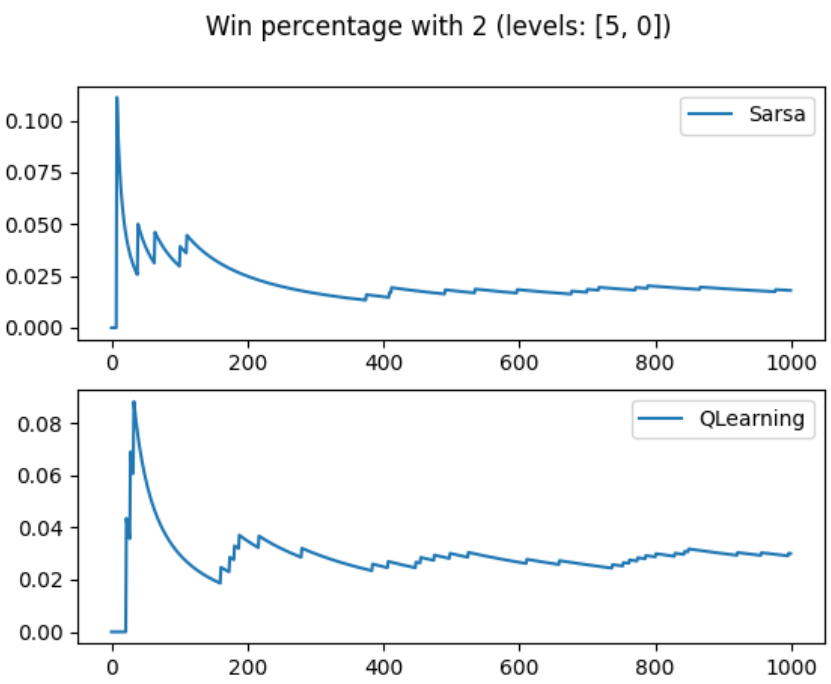

Figure 4.7: Win rate against one opponent with best actions

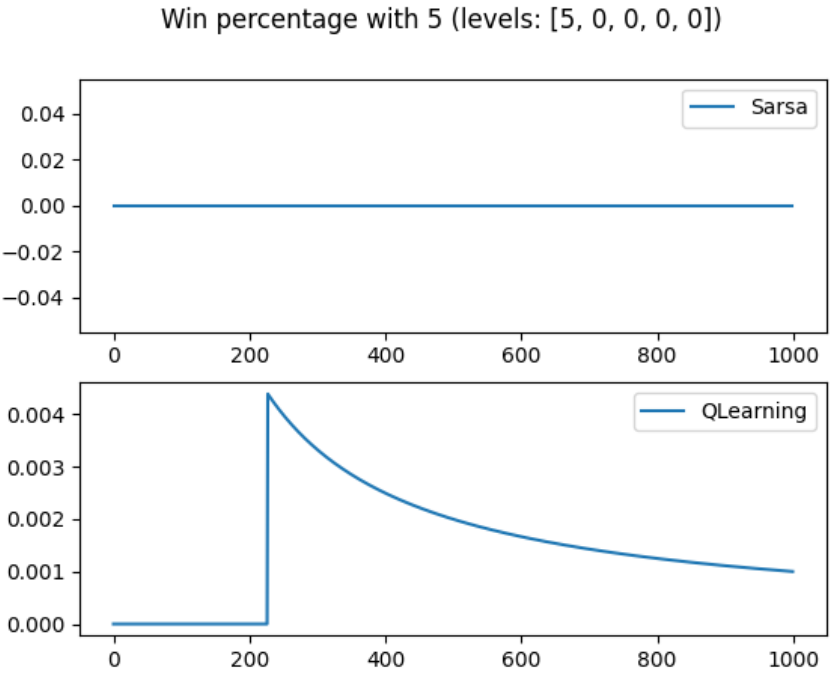

Figure 4.8: Win rate against 4 opponents with best actions

Chapter 5

Conclusion

The algorithms presented did not perform as well as expected. As observed in the experiments, an algorithmic approach is sufficient to obtain good results on a problem that was maybe too simplistic.

One way to improve this project could be to look into algorithms capable of learning efficient solutions in only a few training step and to evaluate their performance with the shortest possible training. An other interesting outlook would be to successfully implement the third planned algorithm that we could not test against the other in the experiments. Allowing us to have a comparison with more elaborated algorithms rather than the most basic we implemented. In conclusion, this project was for us a great insight of bigger reinforcement learning problems but suffered a bit from its small complexity. A more complex problem without a simple algorithmic solving could have better shown the power of reinforcement learning whereas the darts game could be solved even without the use of a computer.

A github repository with the code and figures used in this report is accessible at the address: `https://github.com/a2mont/rl_darts`

Bibliography

[1] Graham Baird. "Optimising darts strategy using Markov decision processes and reinforcement learning". In: *Journal of the Operational Research Society* 71.6 (2020), pp. 1020–1037. DOI: 10.1080/01605682.2019.1610341.

[2] *The Dartboard*. https://pages.cs.wisc.edu/~bolo/darts/dartboard.html 05 June 2022.

[3] *ICM implementation*. https://github.com/karnigili/ReinforcementLearning/blob/master/Tutorials/ICM.ipynb 03 June 2022.

[4] *Openai gym documentation*. https://www.gymlibrary.ml/ 03 June 2022.

[5] Deepak Pathak et al. "Curiosity-driven Exploration by Self-supervised Prediction". In: *CoRR* abs/1705.05363 (2017). arXiv: 1705.05363. URL: http://arxiv.org/abs/1705.05363.