# Playing the game of Maraffone with reinforcement learninig

**Davide Ragazzini**
Authonomus and Adapitive Systems Project
davide.ragazzini4@studio.unibo.it

## Abstract

This project is about implementing an environment and a few agents to play Maraffone also known as Beccacino, a 4 players card game, the agents were trained using reinforcement learning and self play, the agents produced beat the random most of the times, but due to lack of a baseline, evaluating the capabilities of the agents is hard and I don't believe they have reached the level of an experienced human player. Code is availiable at url: https://github.com/davideraga/Marafon

## 1 Introduction

Maraffone is a card game typical of Romagna, four players play the game and they are divided into two teams, the game is composed by hands, at the start of an hand the cards are shuffled and each player receives 10 cards, the staring player chooses the briscola and plays first, each hand has 10 rounds.
The first player of a round can make one of three signs or say the respective word word: "busso"-"I knock", "volo"-"I fly", "striscio"-"I swipe", no other form of communication is allowed between the players.
Despite being a simple game it presents some challenges such as interactions between agents, non deterministic environment and partial observable state.
In this project I have used reinforcement learning to train a few agents and dome some experiments.

## 2 Game Environment

I built a simple environment to simulate the game, one hand is considered an episode, here are some details about the observations, actions and reward.

### 2.1 Observation encoding

The state is only partly observable. The encoding of the observation is composed by 780 variables with value 0 or 1. Each card is encoded as the suit: 4 variables and the rank: 10 variables. After the briscola is chosen the suits are flipped so the first is always the briscola, taking advantage of a symmetry of the game.

- card in hands 140 variables
- the briscola seed: 5 varables
- the round: 9 variables
- the player that is winning in this round till now: 4 variables
- the player that played first in the round: 4 variables

- the sign: 4 variables
- the turn's suit: 5 variables
- the cards on the table: 56 variables
- information about past turns: sign, suit, and cards

In this version the last round is solved automatically by the environment, because playing the last card is forced, I also tried a version that separates the 2 last rounds, this creates a different observation, since there are more rounds, this doesn't seem to affect the performance.

### 2.2 Actions encoding

There are 16 possible actions divided in 3 types:

- playing a card: 10 actions based on the index of the cards, the position in the hand; after a card is played the relative action becomes illegal
- choosing the briscola: 4 actions, one for each suit
- making a sign: 2 actions, I decided to make make the agent only choose to do the sign or not, the right sign is calculate by the environment, for simplicity and because bluffing is not allowed.

Not every actions is always available, an action mask is given by the environment to identify the legal actions, the agents need to save the mask with the experience, to calculate the gradient correctly.

### 2.3 Reward

The objective of one hand is to score the most points and make the opposing team score less point.
The reward has been defined as the difference in points between the 2 teams, another option could be the point scored by one team, since the total points in one hand are 11, apart form the rare maraffa's extra points, the 2 rewards produce similar behavior.
It's possible to calculate the points after each turn, so the reward is given to the agents after every action that they take, when they take a new action they receive the reward for the old action, giving the reward at the end of the episode is another possibility not explored in this paper.

## 3 The agents

I implemented agents to play the game using Q-Learning and PPO, these are 2 simple but effective reinforcement learning methods, some hyperparameter tuning was done in a informal way, testing one parameters a time, not all combinations where explored.
All the network used have one hidden layer with 512 units, adding more layers does not seem to improve the performance.

### 3.1 Semi Gradient Q-Learning

Semi Gradient Q-Learning combines neural networks and Q-learning, the network approximates the state-action value function, given the state as input it outputs the expected value for each action, the network is updated using stochastic gradient descent, Q-Learning is an off policy temporal difference method for reinforcement learning, this is one of the most successful RL methods and it achieved a human comparable level of performance on Atari games [3].
The target is defined this way

$$Y_t = R_{t+1} + \gamma \max_a Q(S_{t+1}, a, \theta)$$

#### 3.1.1 Double Q-Network

Double Q-Learning [2] is an improvement of Q-Learning, it reduces the problem of Q-Learning having more chance to select overestimated values, causing overoptimistic value estimates, this

could cause worse performance and less stable training, Double Q-Network divides the selection and evaluation of actions during the calculation of the target, its main feature is using the argmax of the online network, but using the value of the target network.

$$Y_t = R_{t+1} + \gamma \max_a Q(S_{t+1}, \arg\max_a Q(S_{t+1}, a, \theta_{onl}), \theta_{trg})$$

The learning rate used was `10^-5`, no discount was used, ε was 0.9 with exponential decay with coefficient 0.9999, Adam optimizer was used, replay memory with 100'000 buffer length and 256 batch size and no prioritized replay, the weights are updated after 8 steps and the target network is updated every 1000 updates.

### 3.2 PPO-Clip

Proximal Policy Optimization is a type of policy gradient methods, first proposed by OpenAI [1], the main intuition of PPO-Clip is that we want to use the same data for more steps of stochastic gradient descent, but we don't want the policy to become too different from the one that we used to sample the data, or the data wouldn't be relevant anymore, so PPO clips the ratio between the new policy and the old policy, if the ratio is too far from 1 the gradient relative to that component is not considered.

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

The objective function is clipped this way

$$min(r_t(\theta) * A_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon) * A_t)$$

The min function is for not clipping if it's outside the range but the update will bring it closer.
The parameter that defines the clip range ε used was 0.1. The learning rate was `10^-4` for the value network and `10^-5` for the policy, Adam optimizer was used, batch size was 512 and 4 batches for buffer, 4 update epochs, a entropy loss was added to encourage exploration. I used Generalized Advantage Estimation [4] to calculate the advantage with lambda = 0.92, and no discount. Clipping the objective for the value function does not seem to improve performance, it wasn't used in the final version.

## 4 Experiments

### 4.1 Performance versus random agent

The agents were trained by playing versus a random agent, one team is composed by two agents sharing the same network and only one agent updates the network, this seems to produce better results than training 2 separate nets at the same time, the trajectories for each agent are separated but they share the same buffer, one agent considers the others as part of the environment, both have been trained for 400'000 episodes, the training for each agent took about 10-12 hours, using CPU seems faster than GPU for this task. Two other agents have been trained with self play, in this case all agents share the same network, the episodes are half of the normal version, because agents gather experience from both teams, training time is about the same as the normal version. Self play with reinforcement learning was used in previus works like TD-Gammon[5].
The average performance during training is reported in figure 4.1, the average reward is calculated across the last 1000 episodes, the DQN seems to perform better than PPO.
The agents have been tested by playing 1000 games versus a random agent, the results are reported in 1, as expected scoring more point results in a better win ratio, DQN wins more than PPO, PPO_sp performs worse than PPO in this case. All agents score more points when starting the episode first, as expected this is a big advantage in this game because it allows to choose the briscola, but the agents still score more points than the random agent even when second.

#### 4.1.1 Signs

PPO does the sign 19% of the times, DQN 41%, PPO_sp 98%, DQN_sp 0.49%, signs can be used to give the teammate information, but has the drawback of giving information to the opposing team too,

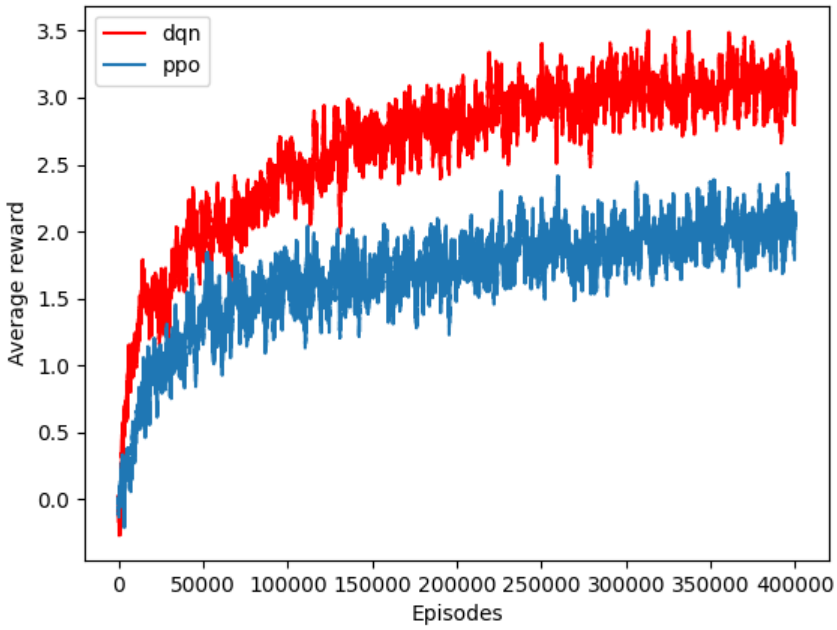


Figure 1: Average reward during training.

Table 1: Performance versus random agent

| Agent | Win ratio | Average difference | Average points | Average points first | Average points second |
|---|---|---|---|---|---|
| DQN | 0.97 | 3.1 | 7.1 | 7.7 | 6.5 |
| DQN sp | 0.97 | 3.0 | 7.1 | 7.7 | 6.5 |
| PPO | 0.89 | 2.0 | 6.6 | 7.1 | 6.1 |
| PPO sp | 0.85 | 1.7 | 6.4 | 7.0 | 5.9 |

unless they are the random agent that can't interpret the signs. This data doesn't suggest that the agent are able to use the signs effectively.

#### 4.1.2 Reward by round

During testing versus random agent I measured the average reward for each round, to see when the agents gain the most of the points and if they are able to predict the last round well, results in 4.2. For all agents the average score increases each round, and in the last round the increase is bigger, so they are able to set up a good last round, agents trained with self play score less points in the earlier rounds but more in the last turn compared to the one trained versus random, in proportion PPO scores more points in the last round, but DQN scores more in total.

### 4.2 Performance versus other trained agents

The agent were tested by playing 1000 games versus each other to observe which agent is best, the results are reported in 2
The rankings of the agents are similar to the previus experiment, but the self play versions are better in this case, DQN still wins more than PPO, the latter seems to benefit less from self play.
All agents score more points on average than the opposing team when starting the hand first (not reported in the table).

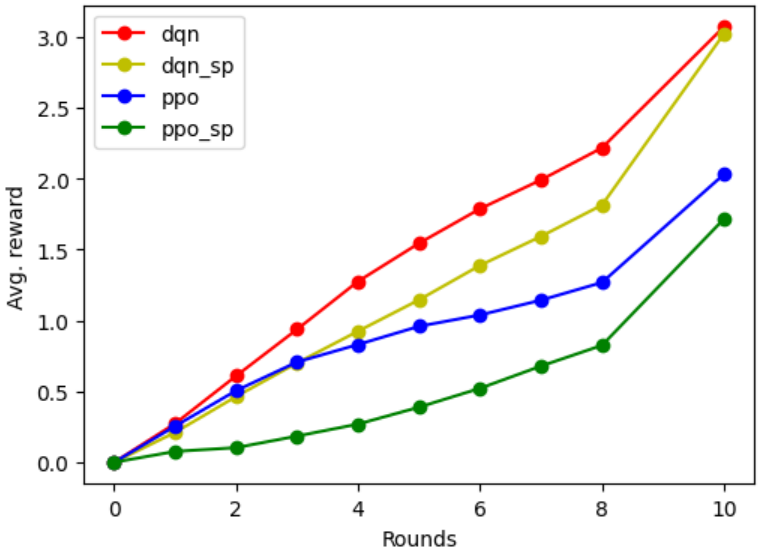


Figure 2: Average reward by round vs rnd.

Table 2: Performance vs trained agent

| Agent 0 | Agent 1 | Agent 0 win ratio | Avg. difference | Agent 0 avg. point | Agent 1 avg. point |
|---|---|---|---|---|---|
| DQN | PPO | 0.75 | 1.2 | 6.2 | 5.0 |
| DQN sp | PPO sp | 0.78 | 1.3 | 6.2 | 5.0 |
| DQN sp | DQN | 0.57 | 0.2 | 5.7 | 5.4 |
| PPO sp | PPO | 0.53 | 0.2 | 5.7 | 5.4 |
| DQN | PPO sp | 0.76 | 1.2 | 6.2 | 5.0 |
| DQN sp | PPO | 0.79 | 1.3 | 6.2 | 4.9 |

### 4.3 Training versus a fixed agent

An angent was trained to play versus the DQN agent and another versus DQN_sp, in the same way the DQN agent was trained and for the same number of episodes, this agents will be called DQN_exploit and DQN_sp_exploit 4.3.
Then they have been tested for 1000 games, DQN_exploit achieves 0.58 win ratio and an average difference of 0.28, DQN_sp_exploit 0.5 win ratio and 0.05 difference, DQN_exploit beats DQN performing a little better than DQN_sp that has no training versus DQN, DQN_sp_exploit is not able to surpass DQN_sp, this shows how self play produces a better agent that is harder to exploit.

### 4.4 Qualitative observations

I played a few games with the agent DQN_sp, the agent controlled 1 teammate and 2 opposing player, my team won the majority of the games, the agent plays in a different way from an human, with some plays that I would consider mistakes, like playing an important card at the wrong time. It appears to understand how to use the briscola and which card to play to win a round and it seems to play an ace if possible when the teammate is winning the round, to score more points.

## 5 Conclusion

Self play improves the agents a little, even if the effect of the choice of opponent for training seems limited, a better method and tuning appear to be more impactful. The trained agents perform better than a random agent, due to the latter not being a good baseline, it's not easy to evaluate the strength of the agents, I don't think the agents have reached human level of play, I believe that a more complex

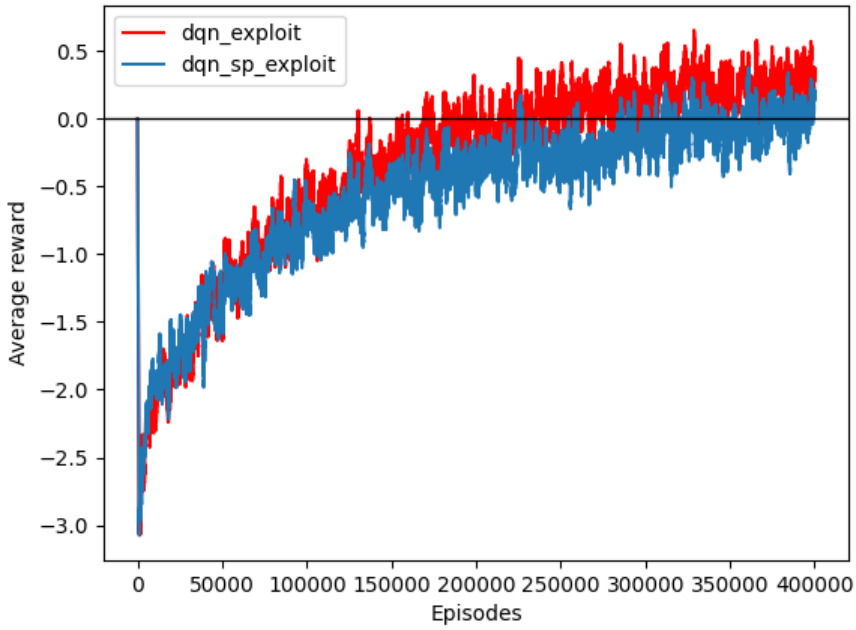


Figure 3: Average reward during training of DQN exploit.

method is needed to achieve that, having data from human players could improve the results. I think that they could achieve better performance with more training time, I don't know if a deeper network could be more effective. Probably an approach that combines search algorithms and reinforcement learning will improve the result.

## References


[1] Schulman et al. *Proximal Policy Optimization Algorithms*. 2017. DOI: 10.48550/ARXIV.1707.06347. URL: https://arxiv.org/abs/1707.06347.

[2] Hado van Hasselt, Arthur Guez, and David Silver. *Deep Reinforcement Learning with Double Q-learning*. 2015. DOI: 10.48550/ARXIV.1509.06461. URL: https://arxiv.org/abs/1509.06461.

[3] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning*. 2013. DOI: 10.48550/ARXIV.1312.5602. URL: https://arxiv.org/abs/1312.5602.

[4] John Schulman et al. *High-Dimensional Continuous Control Using Generalized Advantage Estimation*. 2015. DOI: 10.48550/ARXIV.1506.02438. URL: https://arxiv.org/abs/1506.02438.

[5] Gerald Tesauro. "Temporal Difference Learning and TD-Gammon". In: *Communications of the ACM* (1995). DOI: https://www.bkgm.com/articles/tesauro/tdl.html#references4.