CS 594

Reinforcement Learning

Final Project Report

Giuseppe Cerruto

Edoardo Stoppa

May 1, 2022

1 Abstract

Our project is based on a python library called "RLCard", which lets us create and train reinforcement learning agents, using different algorithms on different traditional card games (for example Blackjack, Poker, Uno, etc. . .). Our work could be divided into 3 major sections. The first, where we added prioritized experience replay to the DQN agent already present in the library. We hope that our agent will be added to the others available in the next release of RLCard. The second, where we explored the behaviour of agents that are specifically trained to exploit only one particular strategy. The last, where we explored the possibility of using a simpler implementation of self-play, that works without all the engineering details explained in most of the papers that use this technique.

2 Background

2.1 RLCard library

RLCard[1] is an open-source toolkit for reinforcement learning research in card games. The entire toolkit was developed adopting three main design principles: Reproducibility (given the same configuration and same seed, all results should be reproducible and completely deterministic); Accessibility (the toolkit should enable experimentation using only a few simple interfaces); Scalability (new environments and agents can be added completely freely). The library has two main components: Environments and Agents. It also includes multiple useful interfaces to be able to efficiently and effectively evaluate the agents' performance.

2.1.1 Environments

Each environment represents one of the multiple card games supported (including Blackjack, Leduc Hold'em, Texas Hold'em, UNO, Dou Dizhu, and Mahjong). The game will proceed using the easy-to-use interfaces of an environment. There are two main way to run a game using an environment: the function `run()`, that return the transitions and payoffs of an entire game without keeping track of the game tree, or the function `step()`, that advance the game only for a single step given the current game state. If the option is enabled during the environment creation, it's also possible to use the function `step_back()` to go to the previous encountered state.

2.1.2 Agents

In this library, each implemented agent has a different algorithm. We can find four different agents: the Deep Q Network agent (DQNAgent), the Neural Fictitious Self-Play agent (NFSPAgent), the Deep Monte Carlo agent (DMCAgent), and the Counterfactual Regret Minimization agent (CFRAgent). The first three can be used in any environment that RLCard offers, while CFRAgent can be only used with one specific environment called "Leduc Hold'em Poker". This is because the CFR algorithm traverses the entire game tree, so it can be used only with games that have a limited search space (like Leduc Hold'em). The library design encourages and facilitate the implementation of custom agent/algorithms. There are only 3 main functions that must be implemented in a custom agent:

- `feed(ts)`: Used to feed a transition (`ts`) to the agent.

- `step(state)`: Used to calculate the next action that the agent will perform, given the current state (`state`)
- `eval_step(state)`: Used to calculate the next action that the agent will perform, given the current state (`state`). This function is used exclusively during evaluation, so it can be implemented differently than `step(state)`.

It's important to notice that there exists another special category of agents in RLCard: Rule Model Agents. This particular type of agent consists of a rule system coded manually by the creators of RLCard. These Rule Models are not available for all games, but they are for the two games that we are interested in. In particular we'll utilize `LeducHoldemRuleAgentV1` and `LimitholdemRuleAgentV1`. These agents only need to implement `step(state)` and `eval_step(state)`, because obviously are not capable of learning from experience. Thus, these models should be used only to evaluate or train other models. We will refer to these models as `BaselineRLCard` agents.

2.2 Algorithms used

Our choice was pretty limited (as written in the *Agents* subsection), so in this section I'm going to explain the basic intuition behind every algorithm that we used (the original paper describing the algorithm will be cited in the title of each subsection).

2.2.1 Deep Q Learning (DQN)[2]

We call a state of the game s, a legal action that can be performed by the agent a , the reward for an action r, and a discount factor γ. We can define the discounted return obtained at time t as $R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}$. We could also define the optimal action-value function $Q^*(s, a)$ as the maximum expected return achievable by following any strategy, after seeing some sequence s and then taking some action a:

$$Q^*(s, a) = max_\pi \mathbb{E}[R_t | s_t = s, a_t = a, \pi]$$

Where π is a policy mapping sequences to actions (or distributions over actions). The Q values obey an important equation, called *Bellman Equation*, that can be expressed as

$$Q^*(s, a) = max_\pi \mathbb{E}_{s'}[r + \gamma max_{a'} Q^*(s', a') | s, a]$$

This equation lets us calculate the Q values in an iterative way (*value iteration*). The problem is that doing it explicitly for all (s, a) is impractical in most real-world situations. Thus, it is common to use a function approximator to estimate the action-value function, $Q(s, a; \theta) \approx Q^*(s, a)$. In our case, this approximator is done using a Neural Network. We refer to a neural network function approximator with weights θ as a Q-network. A Q-network can be trained by minimizing a sequence of loss functions $L_i(\theta_i)$ that changes at each iteration i

$$L_i(\theta_i) = \mathbb{E}_{s,a \approx \rho()}[(y_i - Q(s, a; \theta_i))^2]$$

Where $y_i = max_\pi \mathbb{E}_{s'}[r + \gamma max_{a'} Q^*(s', a'; \theta_{i-1})|s, a]$ is the target for iteration i and $\rho(s, a)$ is a probability distribution over sequences s and actions a that we refer to as the behaviour distribution.

2.2.2 DQN with Prioritized Experience Replay (PDQN)[3]

This particular algorithm was not present in RLCard. Thus, we decided to implement it from scratch, integrating it into the DQN agent, and then adding everything to the original library. Thankfully the implementation was easier than anticipated, and RLCard design helped in the integration and testing phase. We hope that (after polishing the code and commenting everything) this agent could be added permanently to the library in the next public release.

This new algorithm is a variation of the previous one, so most of the theoretical foundation is equal. The main change is about how the memory for experience replay works. The idea is that different transitions have different importance. Some are extremely useful to the learning process, while others are less important. So, if we could use the most important transitions more frequently, we could speed up the training and reach better results. To do that, we could use as a measure of *priority* the TD error that each transition have, as expressed in this formulation:

$$\delta_j = R_j + \gamma_j Q_{target}(S_j, argmax_a(Q(S_j, a))) - Q(S_{j-1}, A_{j-1})$$

The introduction of priority helps us decide which transition we should choose when we sample a batch of transitions, but we cannot sample only using a greedy policy. Instead, to avoid a lot of problems, we could a stochastic sampling method. We now associate each transition to

it's probability of being sampled, calculated as:

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$$

This correction helps a lot but also introduces some bias. Using this type of transition sampling, we are effectively changing the batch distribution to something that is different than what should originally have been. To mitigate this problem, we can introduce the Importance-Sampling weights:

$$w_i = (\frac{1}{N}\frac{1}{P(i)})^\beta$$

The last things needed to point out about our implementation of this algorithm are the parameters used: α was set to 0.6, while β to 0.4. These values worked fine for all our experiments, so we decided to use them always.

2.2.3 Neural Fictitious Self Play (NFSP)[4]

NFSP is a complex algorithm composed of 2 main components: a network that learns a strategy through reinforcement learning, and a second part that tries to compute the Nash Equilibrium for the game. These two things then are combined to determine the best possible strategy (and move) to perform.

Let's start by talking about Fictitious Self-Play. This algorithm tries to compute the best response to the opponent's average behavior. In general, each past move of the opponents is kept in memory, and when it's time to choose the next move, using that past history, the expected return for each legal move is calculated. The move with the maximum expected return is chosen as the final next move.

An NFSP agent interacts with the other agents and memorizes its experience of game transitions and its own best response behavior in two memories, M_{RL} and M_{SL}. The agent trains a neural network, $Q(s, a|\theta^Q)$, to predict action values from data in M_{RL} using off-policy reinforcement learning. The resulting network defines the agent's approximate best response strategy, $\beta = \epsilon - greedy(Q)$, which selects a random action with probability ϵ and otherwise chooses the action that maximizes the predicted action values. The agent trains a separate neural network, $\Pi(s, a|\theta^\Pi)$, to imitate its own past best response behavior using supervised

classification on the data in M_{SL}. This network maps states to action probabilities and defines the agent's average strategy, $\pi = \Pi$. During play, the agent chooses its actions from a mixture of its two strategies, β, and π. NFSP also makes use of two techniques in order to ensure the stability of the resulting algorithm as well as enable simultaneous self-play learning. First, it uses reservoir sampling to avoid windowing artifacts due to sampling from a finite memory. Second, it uses anticipatory dynamics (Shamma and Arslan, 2005) to enable each agent both to sample its own best response behavior and to more effectively track changes in the opponent's behavior.

2.3 Card Games

The RLCard library has a lot of games that we could have chosen (Blackjack, No-Limit Hold'em Poker, Uno, etc...). We decided to focus only on 2 games: Leduc Hold'em Poker and Limit Hold'em Poker. Unfortunately, the resources at our disposal were extremely limited, thus we could only choose games that have a limited search space, especially if you take into consideration the high amount of work needed to fine-tune the algorithms described above.

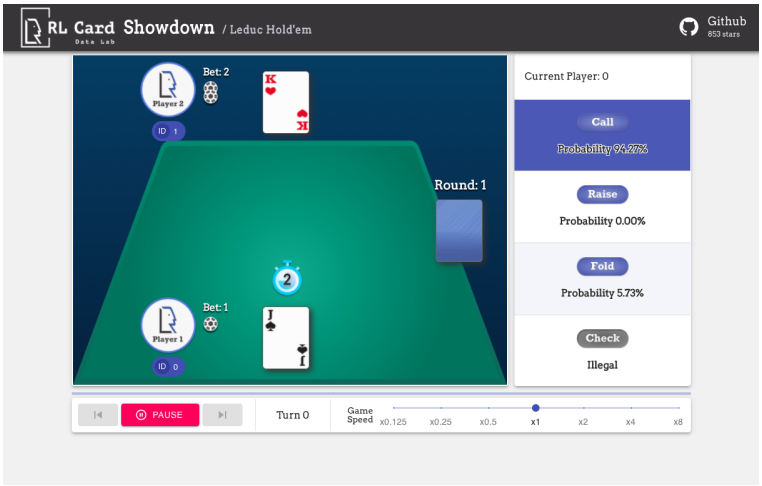


Figure 1: An example of Leduc Hold'em Poker match using the integrated RLCard GUI

2.3.1 Leduc Hold'em Poker

This is an extremely simplified version of poker, generally used in Game Theory research papers[5][6]. The game is played by 2 players, with 3 couple of cards, for a total of 6 cards (generally 2 Kings, 2 Queens, and 2 Jacks). The game is played is this way:

- Each player is dealt one card privately.

- A betting round occurs.
- A single card is dealt face-up on the table.
- Another betting round occurs.
- Players reveal their cards.
- A player wins if their card has the same rank as the one dealt on the table. If no one has the same rank, wins whoever has the highest valued card.

2.3.2 Limit Hold'em Poker

This is an almost normal version of Hold'em Poker, but with some caveats introduced to reduce the state space of the game. In particular, we have that:

- The amount of money you can bet is limited (in a small range with fixed increments).
- In a single betting round, at most only 1 bet and 3 raises are allowed (after that everyone can only call or fold).

3 Related Work

Our project was focused mainly on understanding the behavior of an agent trained to exploit a single strategy, and on understanding if a "naive" way of implementing self-play could actually work.

3.1 Exploiting an opponent's strategy

This paper[7] was one of the first works in this space. It explains how it can be created an agent that is able to effectively exploit using supervised machine learning. To train the agent a lot of game data is collected, and then it's labeled using a binary label: each game state is a weak state for the opponent, or not. In this way, the model should be able to learn how to recognize and create "weak states" for the opponent. The main limitation of this paper is due to the use of Supervised Machine Learning and the non-adaptive methods use. Furthermore, it doesn't describe the behavior of any agent trained using this method.

Another interesting paper about strategy exploitation is this one[8]. The paper describes a way to train an agent in order to safely exploit a non-stationary opponent. The paper

discusses the trade-off between exploiting an opponent and being exploitable. They came up with a way to model the opponent using multiple strategies instead of a single one (given that the opponent is non-stationary). Unfortunately, even if this paper gave useful insight into what means exploiting a strategy, lacks the proper definition of standard behavior for an exploiting agent.

For this first part of the project, we have found that no other paper discussed in depth the behavior of an agent that exploits a specific strategy, thus we hope that our work can fill (at least partially) the knowledge gap on this topic.

3.2 "Naive" Self-Play

We got the inspiration to do some experiments in this section from this paper[9], even if the system described in the paper is obviously extremely more complex than what we are doing. In particular, we were interested in all the technical details needed to make self-play work. For example, they had to choose the adversary from a pool of old versions of the network, and to do that they had to continuously evaluate what was the best performing old network. We thought that this process was significantly expensive in terms of resource consumption, thus the interest in "simplifying" the process.

An extremely similar method to do self-play was used in this paper[10]. The authors trained an agent using self-play simply continuing to train the agent against itself while updating the network. Unfortunately, they didn't explore how this agent behaved other than looking at its final performance.

4 Experiments

4.1 Baseline and Expert training

At the start of our experiments, we trained an agent against a random player, creating a `baseline`. We ran all the experiments on both Limit Hold'em and Leduc Hold'em environments, even if we only present here the results for Limit Hold'em, as they are more relevant. The agent was trained for 10k episodes, and the evaluation of its performance was made every 10 episodes, in a tournament between the agent and a Random-player, for 200 games, averaging the final reward. We created 3 models, using DQN, PDQN, and NFSP algorithms.

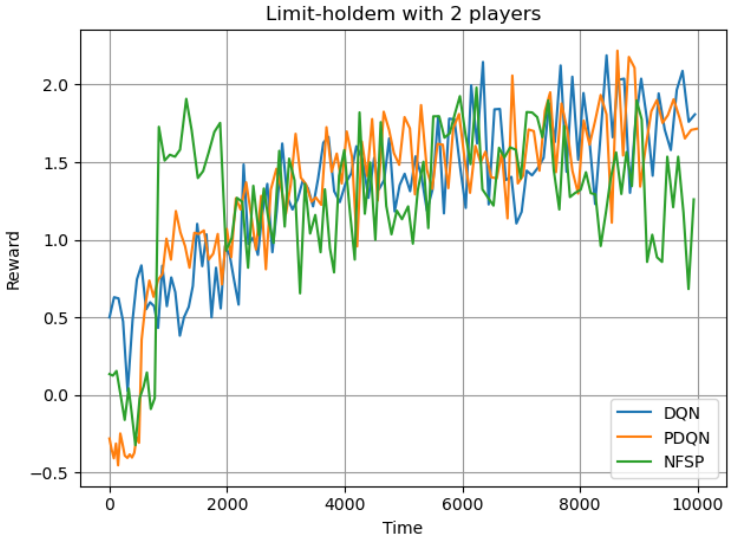


Figure 2: Baseline performance on Limit Hold'em

As shown in 2, the final performance of DQN and PDQN were similar and around 1.8, while NFSP showed more instability with a high pick after 1.5k episodes but decreasing to a final 1.2 average reward.

After this, we wanted to understand if a new agent trained against the baseline, could lead to better performance. We will refer to this agent as `expert`. Here we conducted all the experiments as described above, but the newer agent was trained against the previously trained baseline. Note that all the evaluations were made still against a random player, to have comparable results.

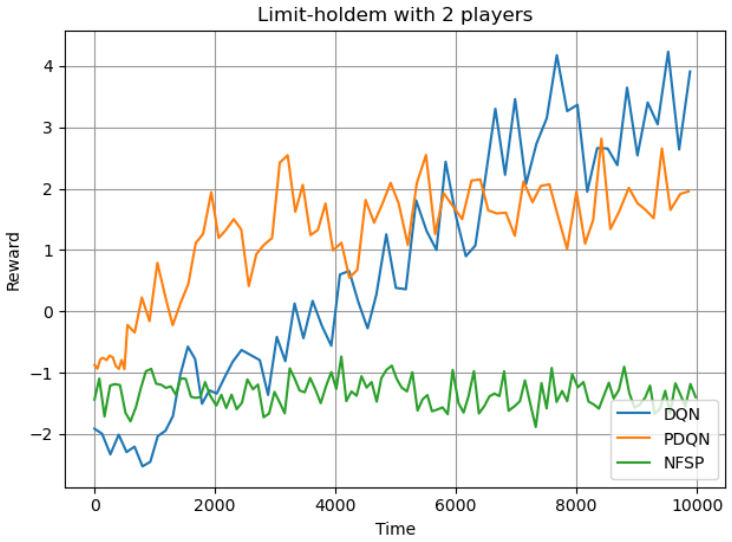


Figure 3: Expert performance on Limit Hold'em

The results are shown in 3. Here three main aspects need to be highlighted:

- NFSP was not able to learn anything, being constant on a negative final reward. We believe that this is due to the strength of the opponent but mainly because NFSP has a large number of hyperparameters to be tuned, so conducting more tuning could lead to better results.
- PDQN was faster in reaching convergence than DQN. It achieved something between 1.5 and 2, as final reward, just after 2k episodes, while DQN took 5k episodes, to achieve the same result. On the opposite, DQN was able to achieve a final reward a lot higher than PDQN, with a score between 3 and 4.
- Maybe the most important result here was that both DQN and PDQN were able to achieve final performances higher than the baseline (2). Especially DQN achieved a final average reward of 3.5, meanwhile, the same algorithm, in the baseline version, was stuck at 1.8.

4.2 Agents' performance in Multiplayer Games

We were interested in understanding if the same intuition that works for agents that are mastering a game works also for the agents that are exploiting a specific strategy. Thus, we execute 3 types of tournaments:

- Baseline vs RandomAgent
- Expert vs Baseline
- Expert vs RandomAgent

The first two types represent an agent against a strategy that is trained to exploit, while the 3 is the example of an agent vs a strategy that isn't trained to exploit. Each of these tournament types was executed having 1 agent of the type mentioned first, and then an increasing number of agents having the type mentioned second. For example, we have a tournament with a Baseline agent and 1 RandomAgent, then another with a Baseline and 2 RandomAgent, etc...

To obtain the numerical results, we executed 10000 games in each tournament, to be sure to have reached approximately the expected return. We've obtained data for games from 2 players to 6 players, with both Leduc Hold'em Poker and Limit Hold'em Poker.

4.2.1 Baseline vs RandomAgent

These are the the numerical results:

	2P	3P	4P	5P	6P
DQN	1.189	1.191	1.186	1.159	1.197
PDQN	1.133	1.129	1.139	1.135	1.129
NFSP	0.832	0.831	0.815	0.805	0.819

Table 1: Expected return for Baseline in Leduc

	2P	3P	4P	5P	6P
DQN	1.844	1.817	1.786	1.789	1.788
PDQN	1.890	1.926	1.874	1.895	1.885
NFSP	1.335	1.351	1.379	1.276	1.321

Table 2: Expected return for Baseline in Limit

	3P	4P	5P	6P
DQN	+0.08	-0.35	-2.61	0.59
PDQN	-0.31	+0.39	+0.15	-0.44
NFSP	-0.16	-2.07	-3.25	-1.44

	3P	4P	5P	6P
DQN	-1.49	-3.15	-2.97	-3.07
PDQN	1.88	-0.88	0.24	-0.29
NFSP	1.17	3.22	-4.48	-1.09

Table 3: +/- performance with respect to 2P game for Leduc (left) and Limit (right)

The data that helps the most to comprehend this experiment is in Table 3. We can see that there are very minor changes in performance (always within ±5%) for both games. DQN was the most consistent, while PDQN and NFSP showed slightly more erratic performance. The only difference between Leduc and Limit Hold'em Poker is that the return variance is slightly higher for Limit.

The main takeaway here is that for Baseline is almost equal to playing again 1, 2, 3, 4, or 5 players. Our intuition is that, if an agent knows exactly how to exploit a fixed strategy, then a multiplayer is reduced almost to a `n` games 1 vs 1. Thus, the performance doesn't change.

4.2.2 Expert vs Baseline

These are the the numerical results:

	2P	3P	4P	5P	6P
DQN	+0.267	+0.266	+0.271	+0.267	+0.266
PDQN	-0.369	-0.359	-0.370	-0.361	-0.368
NFSP	+0.299	+0.291	+0.287	+0.308	+0.298

Table 4: Expected return for Expert in Leduc

	2P	3P	4P	5P	6P
DQN	2.717	2.666	2.717	2.707	2.717
PDQN	2.395	2.3851	2.4117	2.3876	2.3933
NFSP	-1.368	-1.335	-1.418	-1.358	-1.332

Table 5: Expected return for Expert in Limit

	3P	4P	5P	6P
DQN	-0.15	+1.69	-0.04	-0.22
PDQN	-2.71	+0.27	-2.17	-0.27
NFSP	-2.79	-3.98	+2.99	-0.49

	3P	4P	5P	6P
DQN	-1.88	-0.02	-0.37	-0.01
PDQN	-0.41	+0.69	-0.31	-0.07
NFSP	-2.41	+3.72	-0.71	-2.58

Table 6: +/- performance with respect to 2P game for Leduc (left) and Limit (right)

The actual results for this category are practically the same as for Baseline vs RandomAgent. Thus, all considerations also work here.

4.2.3 Expert vs RandomAgent

These are the the numerical results:

	2P	3P	4P	5P	6P
DQN	1.192	1.151	1.077	1.179	1.266
PDQN	1.033	1.121	0.913	1.036	1.134
NFSP	0.682	0.546	0.749	0.632	0.688

Table 7: Expected return for Expert in Leduc

	2P	3P	4P	5P	6P
DQN	+1.557	+1.501	+1.393	+1.754	+1.562
PDQN	+1.596	+1.605	+1.760	+1.488	+1.652
NFSP	-0.051	-0.064	-0.041	-0.047	-0.043

Table 8: Expected return for Expert in Limit

	3P	4P	5P	6P
DQN	-3.49	-9.71	-1.08	+6.14
PDQN	+8.52	-11.55	+0.34	+9.84
NFSP	-19.94	+9.79	-7.33	+0.88

	3P	4P	5P	6P
DQN	-3.65	-10.54	+12.66	+0.28
PDQN	+0.55	+10.31	-6.76	+3.49
NFSP	+26.96	-19.3	-6.96	-14.03

Table 9: +/- performance with respect to 2P game for Leduc (left) and Limit (right)

Now the results are very different. As we can see, the performance becomes rather unpredictable. We can no longer see any pattern in the performance, in fact, we can see that a high variance in performance is present in both games. This effect is definitely more present in Limit Hold'em Poker because the game itself is more complex and can show more interesting properties. Leduc Hold'em Poker also shows a similar behavior, but given the limited state-space search the effect is reduced.

Another interesting fact can be observed if we compare the absolute returns in Table 1 and Table 2 with Table 7 and Table 8. Both of those agents are against RandomAgent, but the one with the highest return is Baseline. This goes against the common intuition that "if Expert is stronger than Baseline, then Expert should be even stronger against RandomAgent". But this is not the case, instead, Baseline has higher returns that Expert. This can be explained by thinking about the nature of these agents: they are not mastering a game, but learning how to exploit a specific strategy. Thus, given that Baseline is trained specifically to exploit RandomAgent while Expert is not, the results become clear.

4.3 "Naive" self-play

At this point, we wanted to explore a different type of training. Until now all the experiments were run against a Random Agent or against a Baseline, previously trained against random. Here, we wanted to explore a new type of training: `"naive" self-play`.

For self-play, we consider a scenario where an agent is trained against itself. With "naive" we mean a pure game where an agent is facing itself, without any particular engineering strategies, useful to maximize the convergence or avoid local-maximum. All the experiments were run using the DQN algorithm. We will refer to the DQN agent that we are trying to learn as `agent`; to the agent playing in position 1 as `player1`, and to the agent playing in position 2 as `player2`. The agent is trained for 10k episodes, the evaluations are made

every 10 episodes, in a tournament between the agent and a Random-player for 200 games, averaging the final reward.

First, we ran this experiment in the Leduc Hold'em environment. As described in the introductory version above, this is an extremely simple version of Poker, with a limited action and observation space.

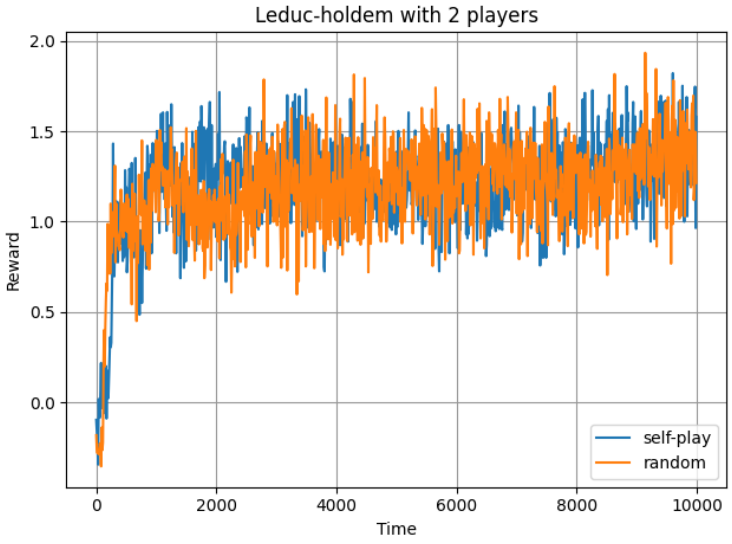


Figure 4: Self-play vs random on Leduc Hold'em

As shown in the 4 the self-play version and the random version, achieved the same performance. We believe that this is due to the simple environment, where there is no space for a particular improvement, as all the possible policies are very limited and easy to learn.

Then, we moved to the Limit Hold'em environment. With a bigger state and action space, almost similar to real Poker, we could finally achieve different performances between the two training.

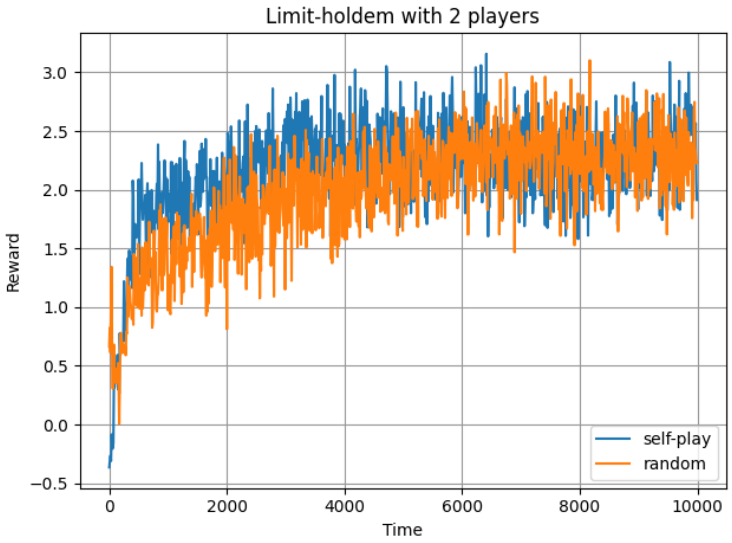
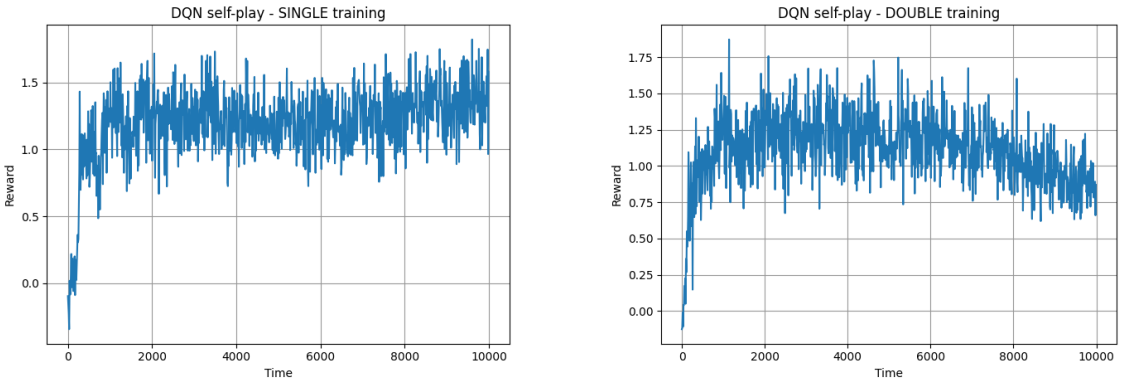


Figure 5: Self-play vs random on Limit Hold'em

As shown in the 5 the final reward are pretty much the same. What changes is the velocity of convergence. We can see that the self-play training was slightly faster in reaching convergence than the simple random version. The self-play uses 4k episodes to reach the final average of 2.5, meanwhile, the random version had to wait until 6k episodes.

At this point, we wanted to explore more the area of self-play training. When training the agent in an environment 1v1, two trajectories (s,a,r,s') will be created: the first for player1 (trajectories1) and the second for player2 (trajectories2). Until this moment, all the previous experiments were conducted using only trajectories1, we will call this `single training`. Now we wanted to explore the effect of using both the trajectories during the training, we will refer to this as `double training`. We ran the training for 10k episodes, with the same set-up used before.



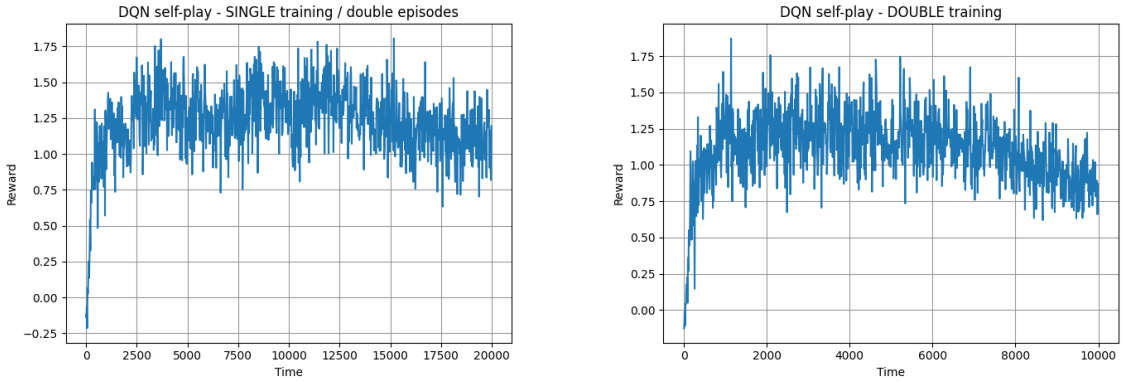
(a) single training

(b) double training

Figure 6: Single training vs Double training

As shown in 6, the single and the double training achieved the same results. This was not expected and our belief is that, as we are using a "naive" approach to self-play, there is no space for particular improvements, using both the trajectories. With some engineering strategies focused on improving the self-play training, we believe that using the double training approach could lead to faster convergence. Note also that in the double training we have slightly more instability of the reward during the simulations, and this could be related to some overfitting of the strategy.

In the end, we wanted to compare the single and double training approaches with the same number of trajectories. We ran an experiment with 10k episodes for the double training (20k trajectories) and 20k episodes for the single training.



(a) single training / double episodes

(b) double training

Figure 7: Single training with double episodes vs Double training

As shown in 7 there is no particular improvements in using one approach or the other, as already described above. Considering the slight instability of the double training, we could say that, in this "naive" version of the experiment, the single training is preferred.

4.4 ExpertRLCard: Training against BaselineRLCard

As described in the introductory section above, the RLCard suite provides also hard-coded agents (BaselineRLCard). At this point, we were interested to explore the performances of an agent trained against those agents. We trained the agents in the same set-up used in section 4.1, for the previous experiments, and also, as before, the testing was done against a random agent. We will refer to these new agents, trained against the BaselineRLCard, as `ExpertRLCard`. The results are shown in 8.

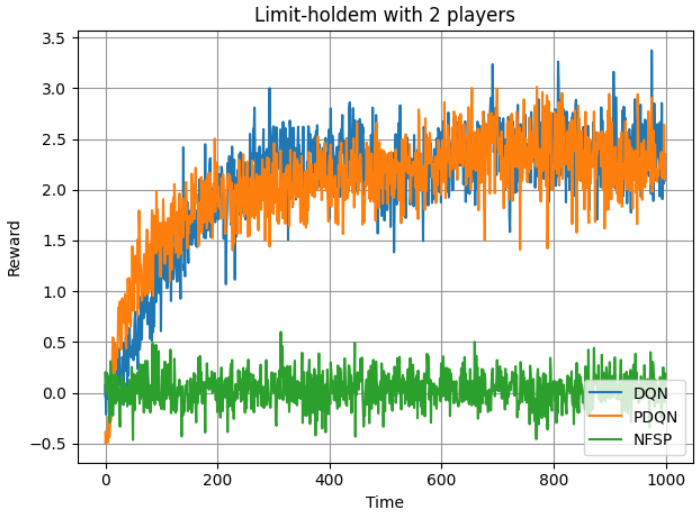


Figure 8: ExpertRLCard on Limit Hold'em

Here two main aspects need to be highlighted:

- NFSP was not able to learn anything, being constant on a neutral reward. As already talked about before, we believe that this is due to the strength of the opponent but mainly because NFSP has a large number of hyperparameters to be tuned, so conducting more tuning could lead to better results.

- DQN and PDQN achieved pretty much the same results, with a final average reward of around 2.3. Compared to the baseline (agent trained against a random agent with a 1.8 final average reward) and described in 2, we can see that both the algorithms were able to perform better. This is an important result because it showed that training against an agent with more knowledge of the game, rather than a RandomAgent, leads to an improvement in performance.

- Instead, comparing these performances to the expert (agent trained against the baseline) described in 3, both the algorithms, but especially DQN, performed worse. We believe that this is because all the performance evaluations were made against a random agent. The ExpertRLCard was trained against a hard-coded agent, with no random behavior exploration. Instead, the expert was trained against a baseline, previously trained vs random, so it was able to exploit random behavior better.

4.5 Tournament between agents

In the end, we conducted some 1v1 tournaments between all the models that we have: baseline, BaselineRLCard, Expert, and ExpertRLCard, on both Leduc-Hold'em and Limit-Hold'em, with the three algorithms DQN, PDQN, and NFSP.

Making the combinations of all the 4 models, we ended up with 6 tournaments. The data for all the tournaments are available in our GitHub repository.

We will present here the result for the tournament between the strongest models: Expert and ExpertRLCard, using DQN and PDQN (as NFSP was not able to learn anything in both cases). We ran 10000 tournaments between the two agents, averaging the final reward. A positive final score means that player1 won, and a negative final score saw player2 winning. The Expert was player1 and ExpertRLCard was player2

	Leduc-Hold'em	Limit-Hold'em
DQN	-0.2692	-0.3888
PDQN	-0.7999	-1.9265

Table 10: Expert vs ExpertRLCard

Table 10 shows an important result. We saw in the previous section that the expert achieved better performance than ExpertRLCard because it was trained to exploit random behavior. But, in a tournament between the two, the ExpertRLcard won every time, highlighting how, in a real world scenario, training an agent against a stronger agent with knowledge of the game, leads to a stronger model with higher capabilities.

5 Conclusions

We showed that an expert achieved better results than a baseline. Training against a stronger opponent (baseline) instead of a random player, led to a stronger model able to score higher performance.

Interestingly, we also discovered that when an agent is directly trained to exploit another strategy, it doesn't matter how many other agents (that hold the exploited strategy) it is playing against, the performance will be practically the same.

Furthermore, we explored the effect of a "naive" self-play training approach. We showed that self-play training achieved convergence faster than the training vs a random agent. Moreover, we showed that, in this "naive" version of the experiment, there is no improvement in using a double training approach.

In the end, we explored training against hard-coded agents already included in the library. We saw that these new agents (ExpertRLCard) performed worse, against random, than the experts, but won all the tournaments against them. That highlighted the difference between exploiting a strategy and learning a better strategy, to win the game.

6 Future Works

In general, we would like to expand our data using more games and more algorithms. Furthermore, we'd like to be able to properly tune NFSP, because it is one of the best algorithms in the field.

Also, we would like to explore how adding to our "naive" self-play some of the technical details mentioned previously impacts the learning and the performance. In particular, given that all experiments for this section were completed in a `1 vs 1` setting, we want to explore how a multiplayer game modifies the behavior of the agents and the final performances.

References

[1] Daochen Zha et al. *RLCard: A Toolkit for Reinforcement Learning in Card Games*. 2019. DOI: 10.48550/ARXIV.1910.04376. URL: https://arxiv.org/abs/1910.04376.

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

[3] Tom Schaul et al. *Prioritized Experience Replay*. 2015. DOI: 10.48550/ARXIV.1511.05952. URL: https://arxiv.org/abs/1511.05952.

[4] Johannes Heinrich and David Silver. *Deep Reinforcement Learning from Self-Play in Imperfect-Information Games*. 2016. DOI: 10.48550/ARXIV.1603.01121. URL: https://arxiv.org/abs/1603.01121.

[5] Terence Schauenberg. "Opponent Modelling and Search in Poker". In: 2006.

[6] Finnegan Southey et al. *Bayes' Bluff: Opponent Modelling in Poker*. 2012. DOI: 10.48550/ARXIV.1207.1411. URL: https://arxiv.org/abs/1207.1411.

[7] Shaul Markovitch and Ronit Reger. "Learning and Exploiting Relative Weaknesses of Opponent Agents". In: *Autonomous Agents and Multi-Agent Systems* 10 (Mar. 2005), pp. 103–130. DOI: 10.1007/s10458-004-6977-7.

[8] Zheng Tian et al. *Learning to Safely Exploit a Non-Stationary Opponent*. 2021. URL: https://openreview.net/forum?id=zoQJBVrhnn3.

[9] Noam Brown and Tuomas Sandholm. "Superhuman AI for multiplayer poker". In: *Science* 365.6456 (2019), pp. 885–890. DOI: 10.1126/science.aay2400. URL: https://www.science.org/doi/abs/10.1126/science.aay2400.

[10] Michiel van der Ree and Marco Wiering. "Reinforcement learning in the game of Othello: Learning against a fixed opponent and learning from self-play". In: (2013). DOI: 10.1109/ADPRL.2013.6614996.