# Comparing Reinforcement Learning in poker Texas Holdem against others strategies

Léo Wenger
Department of HEC, University of Lausanne, Switzerland leo.wenger@unil.ch

**Abstract.** Abstract: Poker presents a challenge for reinforcement learning (RL) due to it's complexity and incomplete information. However, recent research has made significant strides in addressing these challenges, enabling RL models to compete effectively in poker scenarios. This paper explores the practical application of existing RL algorithms in poker, focusing on users with basic poker knowledge. We investigate the accessibility of 2 algorithms, evaluate their learning progress, and seek to comprehend the strategies developed by the models post-training. Our research question explores into the effectiveness of RL techniques in enabling a poker bot to learn optimal gameplay. Through comparative analysis with other poker-playing agents, we assess the performance and adaptability of RL models in navigating the complexities of multi-agent environments. Overall, our study shows the feasibility and effectiveness of employing RL in poker and offers insights into the future directions of research in this domain. However, we didn't succeed in fully bridging the gap between theory and practice due to limitations in my knowledge.

**Key words:** Poker, Machine Learning, DQN, NFSP, Texas Holdem, Reinforcement Learning

## 1. Introduction

Poker, with its complexity and incomplete information, has historically posed significant challenges for implementing reinforcement learning (RL) due to the numerous possibilities and inherent variance. However, over the past decade, extensive research has addressed these issues. Initial approaches simplified the game to one-on-one scenarios made by Fredrik A. Dahl in 20001 (reference 1) then it was improved by Zhao, E., Yan, R., Li, J., Li, K., & Xing, J. in their paper in 2022 (reference 2). In parallele, advancements enabled RL models to compete against multiple players in the paper by Shi, D., Guo, X., Liu, Y., & Fan, W. (reference 4), for now these models can't adapt their strategies in real-time based on their opponents' behavior but it's only a matter of time.

Most of these pioneering studies have been conducted by PhD students and renowned researchers. The objective of this paper is not to introduce a novel algorithm. Instead, we aim to explore the practical application of existing RL algorithms for poker, particularly for individuals with basic knowledge of the game.

We seek to answer several key questions: 1. How accessible are these existing algorithms for users with fundamental poker knowledge? 2. How effectively can we evaluate the learning progress of an RL algorithm in the context of poker? 3. To what extent can we comprehend the strategies developed by the model after its training?

## 2. Research Question

Building on these foundational questions, the primary research question of this project is: Can a poker bot effectively learn to play the game optimally through reinforcement learning techniques? By exploring this question, we aim to understand the feasibility and effectiveness of using reinforcement learning by comparing the model against other poker-playing agents with different strategies. Through this investigation, we seek to assess not only the performance of the RL model but also its ability to develop and adapt strategies in a complex, multi-agent environment.

## 3. Foundations of Poker Texas Holdem

This chapter provides foundational knowledge essential for understanding the game of poker. While seasoned players may find this information familiar, it serves as a primer for readers new to the game. Feel free to skip this chapter if you are already familiar with the fundamentals of poker especially Texas Holdem.

Poker is a dynamic game comprising multiple rounds, each consisting of distinct states: Preflop, Flop, Turn, and River. These phases dictate the progression of gameplay and influence players' decision-making processes, each state continuing as long as the bets are not equal for all players (all players that didn't fold of course):

**States of Poker Gameplay**

- Preflop: Players receive two cards and must decide whether to fold, check, call, raise, or go all-in (*) based on the strength of their hand.
- Flop: Three community cards are dealt, and players have another opportunity to make decisions based on the new information.
- Turn: A fourth community card is revealed, prompting another round of decision-making.
- River: The final community card is unveiled, leading to the last round of player decisions.

**Determining the Winner**

Players compete to create the best possible combination of five cards from their two private cards and the five community cards. The player with the strongest hand at the end of the final betting round wins the pot. (**references 5**)

Poker offers diverse strategic possibilities, with players having the opportunity to win a hand at any phase if opponents fold. The complexity of the game, coupled with the multitude of decision points, historically posed challenges for reinforcement learning algorithms.

(*) Fold is when you don't want be involved in this round, Check mean a bet of 0, Call mean follow the bet of the previous opponent, Raise mean increase the bet (half-pot, pot...), All in is a form of raise where you bet all your chips. The pot is the amount of all chips bet from the beginning of the round.

## 4. Methodology

### 4.1. What We Use

Now that you know how the game works, we propose to implement a poker bot using reinforcement learning techniques. The bot will play against itself in simulated poker games, gradually improving its strategy through trial and error. We will first implement an environment, then an algorithm where the bot learns to maximize its expected future rewards by interacting with the environment, which in this case is a simulated poker game (Texas Hold'em, no limit). The bot will receive rewards based on its performance in each game, with the goal of maximizing its long-term winnings.

### 4.2. Library Explanation

For this project, we mainly use the library RL Card (reference 7). RLCard is a toolkit for Reinforcement Learning (RL) in card games. It supports multiple card environments with easy-to-use interfaces for implementing various reinforcement learning and searching algorithms. The goal of RLCard is to bridge reinforcement learning and imperfect information games. RLCard is developed by DATA Lab at Rice and Texas A&M University, and community contributors We use two algorithms called DQN and NFSP to make the bot learn by playing against random agents.

### 4.3. DQN (Deep Q-Network)

In theory the objective of DQN is to learn a value function that maximizes the expected cumulative reward. It uses a single neural network to approximate the Q-values for different actions given a state. DQN trains the agent by interacting with the environment and updating the Q-network based on the TD-error (Temporal Difference error) between predicted and target Q-values. Typically, DQN uses epsilon-greedy exploration, where the agent chooses random actions with a small probability epsilon and otherwise selects the action with the highest Q-value. It is suitable for single-agent environments with discrete action spaces, such as board games, Atari games, UNO and more. DQN performs well in environments with relatively simple state-action spaces and clear reward signals.

### 4.4. NFSP (Neural Fictitious Self-Play)

NFSP aims to learn both an optimal strategy for the player (policy network) and an approximate best response strategy (Q-function network) by playing against itself. NFSP utilizes two separate neural networks: one for the policy network (which learns the optimal strategy) and another for the Q-function network (which learns the best response strategy). It employs self-play to train both the policy and Q-function networks. During self-play, the agent alternates between exploiting the current policy and exploring new strategies. The policy and Q-function networks are updated using different loss functions. NFSP balances exploration and exploitation by gradually decreasing the exploration rate during training. Initially, the agent explores more to discover diverse strategies, but as training progresses, it relies more on its learned policy. NFSP is designed specifically for imperfect-information games like poker, where traditional reinforcement learning algorithms may struggle due to the complexity of the game state and the need for strategic thinking. NFSP excels in imperfect-information games like poker, where it can learn complex strategies and adapt to opponents through self-play.

Overall, NFSP is build specifically for imperfect-information games and incorporates techniques to handle the challenges posed by such environments. The previous lines were primarily based on my understanding of the paper by Johannes Heinrich and David Silver (reference 3)
The goal here is not to delve into the specifics of how these algorithms work, but to identify their potential use cases and observe if we can distinguish the differences between them.

### 4.5. Simplifications Made

- Throughout most of the training, the bot plays 1 vs 1. This is an effective way to limit the high variance inherent in poker. In a 2018 paper, it was noted that it took 3 days on a computer to play 100,000 games. That version was simpler than mine, and now 100,000 games can be played on my computer in 25 minutes (reference 2).
- The bet amounts are fixed at half-pot, pot, or all-in. This approach eliminates the need to explore every possible bet size.
- The bot plays against random players who fold, check/call, raise, or go all-in 25% of the time (depending on the state of the game, some actions may be impossible while others are available. The idea is that each time they have multiple options, each option is equally likely).
- Playing against random players is not the same as playing against real people. However, we compare each agent against random players. As stated, the goal is to measure the performance of DQN and NFSP compared to other agents.

## 5. Results

### 5.1. Results of DQN

Before continuing, a few points to note. We attempted to tuned the parameters of the DQN agent, but it did not significantly increase the reward. Increasing the number of episodes did not change the fact that the DQN algorithm plateaus after 20,000 episodes (even with more than 300,000 episodes).

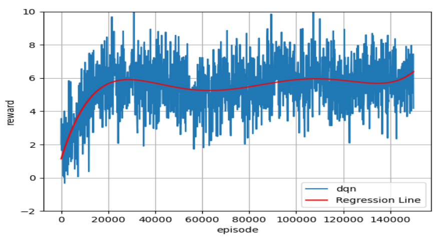


**Figure 1 Trajectories and regression of DQN during the training.**

In the end, we ran 150,000 episodes. The optimal degree of the regression curve is 5. Once it reaches 20,000 episodes, the performance plateaus, and on average, a DQN agent achieves a **reward of 5,61**. We will later compare this reward with those of other models to evaluate its effectiveness.

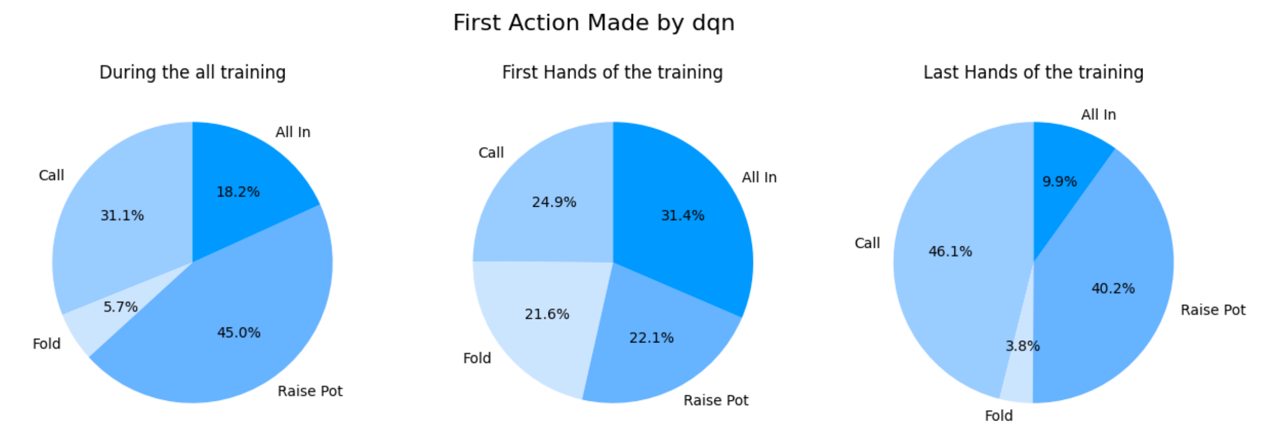


**Figure 2** **DQN Ratio of actions preflop for all datas, beginning and end**

The most interesting observation from the training process is the change in the agent's decision-making behavior. Initially, the algorithm chooses between calling, folding, raising, and going all-in approximately 25% of the time for each action. However, by the end of the training, its behavior shifts significantly:
**Folding**: The agent folds only 3-4% of the time.
**Raising**: The agent raises 40% of the time.
**Checking/Calling**: The agent checks or calls 35% of the time.
**All In**: The agent goes all in 10% of the time.

These changes indicate that the DQN agent adapts its strategy over time, moving away from random action selection towards more strategic decision-making.
**Notes**: "First hands" refer to the first 10% of the hands played during training, which shows that the algorithm starts to learn from this initial subset, hence the percentages are not exactly 25%. And "Last hands" refer to the last 10% of the hands played during training.

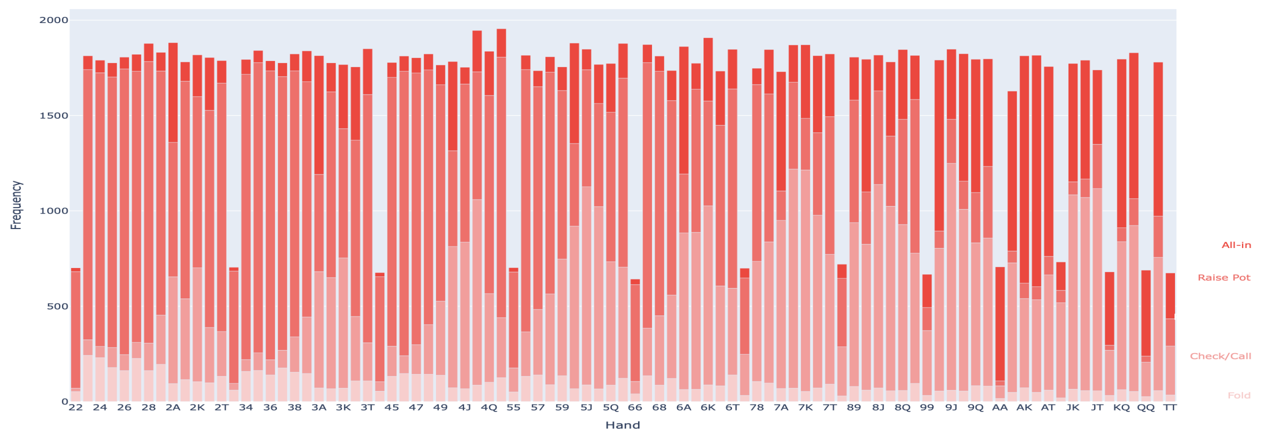


**Figure 3** **Frequency of each hand postflop and Actions for all data in DQN**

The graph offers several intriguing insights. While the hands aren't precisely classified by strength, they're close enough. The general trend is that weaker hands are on the left and stronger hands are on the right, with pairs and Aces being particularly noteworthy. Note that you can find 2 others similar graph in the References for the beginning and the end of the training

**1. Frequency of Pairs:** It's expected that pairs like AA appear less frequently than combinations like AK. There are only 6 possible pairs for AA (AsAh, AsAd, AhAd, AsAc, AdAc, AhAc) compared to 16 combinations for AK. This means pairs show up 16/6 = 2.66 times less often than other hands.

**2. All-In Decisions:** The better the hand, the more likely the bot is to go all-in on the first decision. The darker red sections on the right side of the graph, where the better hands are listed, illustrate this tendency.

**3. Rarely Folding:** The bot rarely folds, which was initially surprising. One might think folding bad hands would be optimal. This behavior might be due to the reward system favoring not folding immediately, especially since the opponent plays randomly.

**4. Raising with Worse Hands and Checking/Calling with Good Hands:** The bot tends to raise with weaker hands and check/call with average or good hands. There are two possible explanations for this:
1. The bot might understand the concept of bluffing. With a good hand, there's no need to scare the opponent away. Instead, it can wait for the opponent to bluff and then capitalize.
2. The bot realizes that since the opponent plays randomly, raising with a weak hand could be advantageous. By raising, the bot gets another chance as the opponent is likely to fold 25% of the time. If the bot raises twice, the probability of the opponent not folding both times is 56.25%. With good or average hands, it's better to see the flop and evaluate the strength of the hand, as the opponent could put the bot in a tough spot by going all-in 25% of the time.

**5. All Datas:** The graph containing all the data confirms that the initial action for strong hands is predominantly all-in (dark red), while weaker hands are mainly subjected to raises (medium dark red). - It's puzzling why the bot chooses to go all-in with AA and QQ more frequently than with KK. Typically, one would expect it to prioritize AA, KK, or QQ, or the reverse order, but here it's a mix of the two. Indeed, it's possible that the bot understands the necessity of mixing its strategy with very strong hands to avoid predictability and maintain a level of strategic complexity. It's also possible that during the training due to randomness, QQ was thought to be better than KK, and the error propagate.

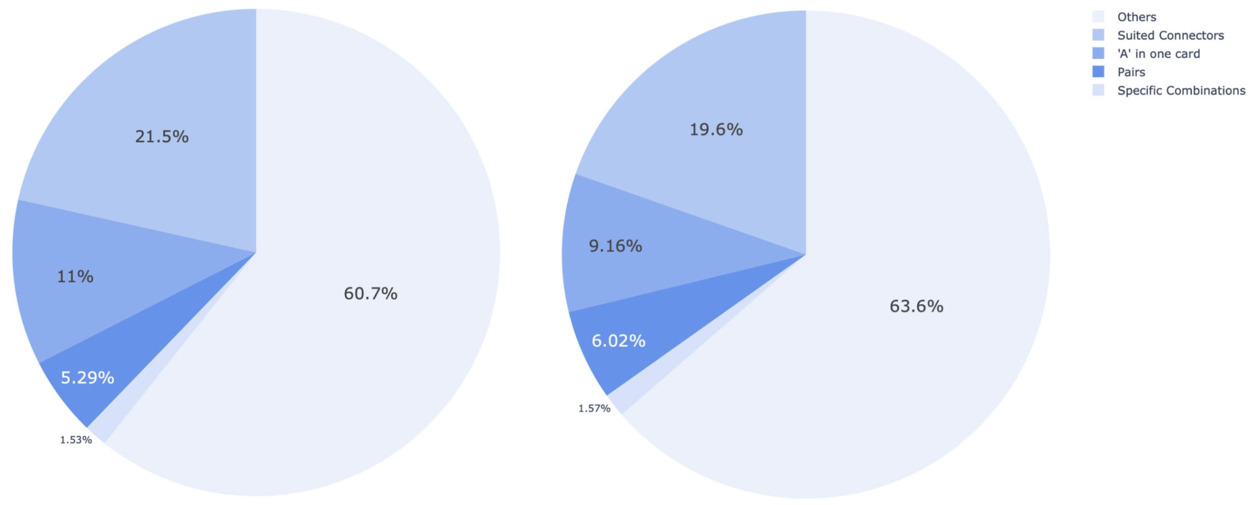


**Figure 4** Fold ratio for the beginning and end of training

**For Fold:** We see that the bot folds more garbage hands than other types, which is a good sign. However, there is no significant difference between the beginning and the end of the training, except the other hands, that are folded 63.6% at the end compared to 60.7%. Note that we only see this in the interactive graph but we have 2157 observations in the beginning but only 382 in the end, so it confirmed that DQN fold less frequently overall.

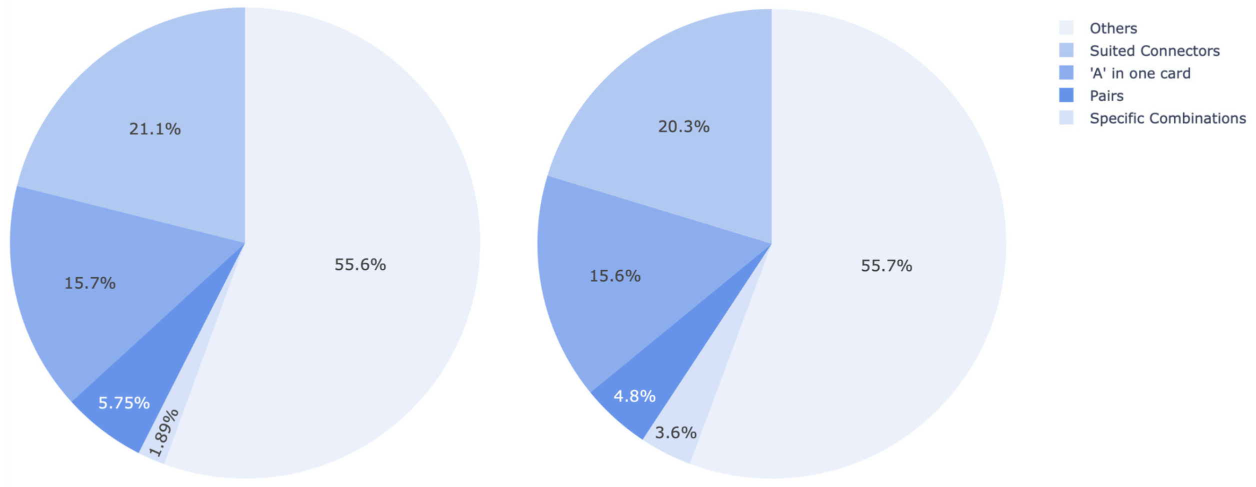


**Figure 5** Check/Call ratio for the beginning and end of training

**For Call:** The bot calls more often with good hands compared to other hands. The only slight difference between the beginning and the end of the training is that the bot calls less often with a pair at the end of the training.

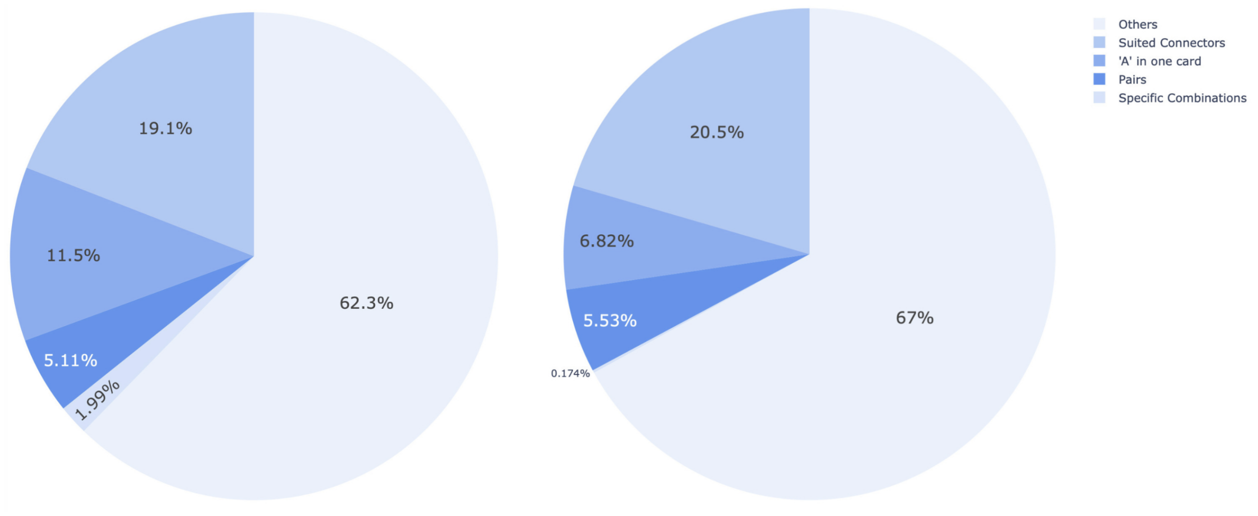


**Figure 6 Raise ratio for the beginning and end of training**

**For Raise:** We clearly see that the bot raises more with other hands than with good hands. He also decide to raise less at the end of the training when he has a A in his hand or a specific combination. This confirms our earlier observation that the bot tends to raise with average or garbage hands.

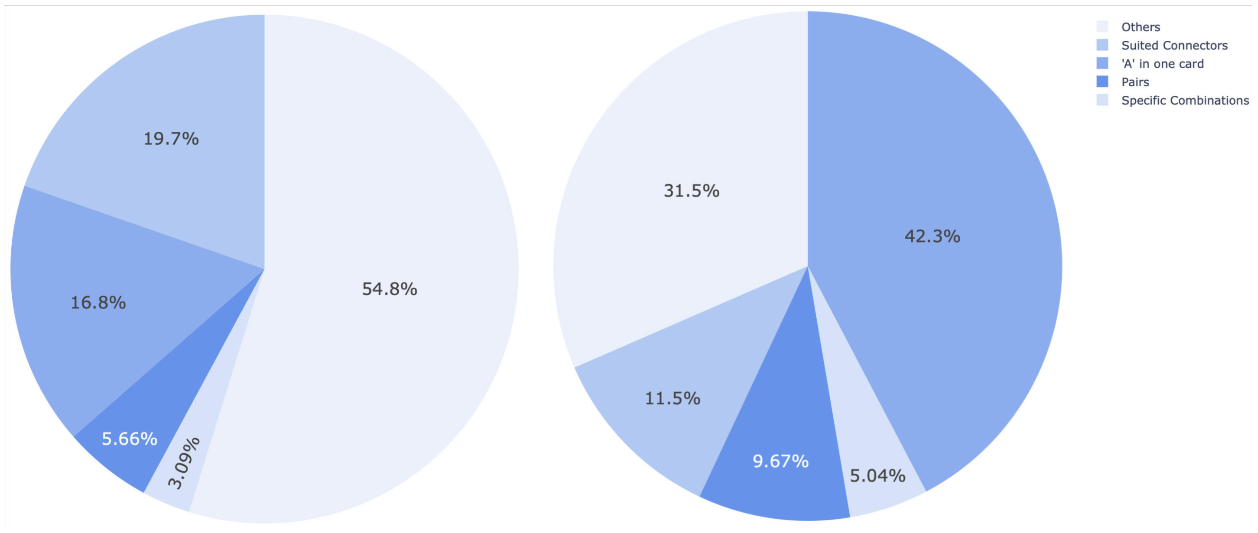


**Figure 7 All In ratio for the beginning and end of training**

**For All-In:** There is a major difference between the beginning and the end of the training. Pairs, specific combinations and A in the hand are more likely to go all-in compared to other combinations. Here DQN fold 3 times less overall when we compare the beginning and the end of the training.

## 5.2. Results of a Simple Agent

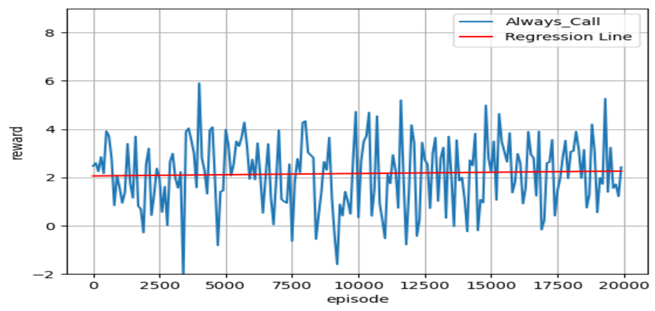


**Figure 8** **Trajectories and regression of Always Call Agent during the training.**

We only ran 20,000 episodes because we know the agent won't improve due to its deterministic play. The goal was primarily to observe the average reward before comparing it to the DQN agent. This agent has a **reward of only 2,16** compared to 5,61 with the DQN agent, demonstrating that the DQN agent performs significantly better than an algorithm with a simple strategy.

## 5.3. Results of a "Special" Agent

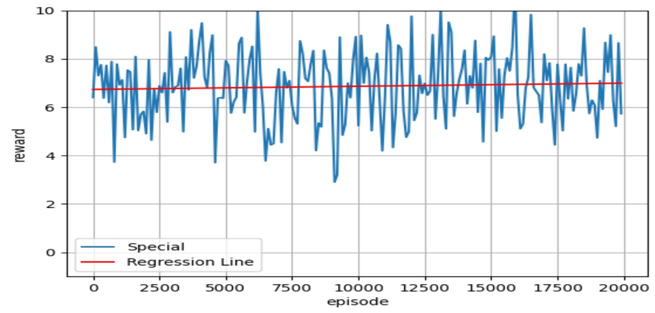


**Figure 9** **Trajectories and regression of Special Agent during the training.**

We also ran 20000 observations and we get a **reward of 6.86**. Surprinsingly the special agent that don't even play perfectly, outperformed the DQN agent. Not by a lot but with a significant margin. So why can't the DQN agent improve further? We'll delve into this discussion later on. Maybe NFSP can do better ?

## 5.4. Results of NFSP

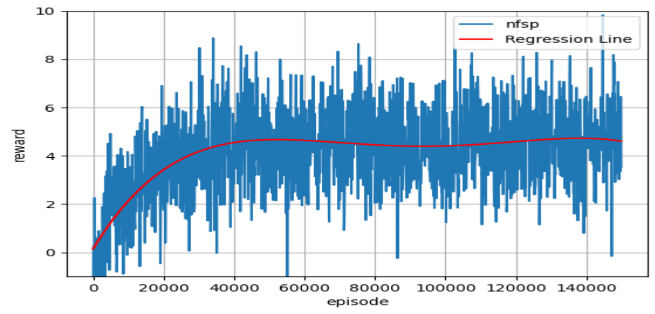


**Figure 10** **Trajectories and regression of NFSP during the training.**

After 40000 episodes, a nfsp Agent as a **reward in average of 4.56**. For some reason, the reward is lower than that of DQN, but we're interested in examining how different the strategy of this algorithm is.

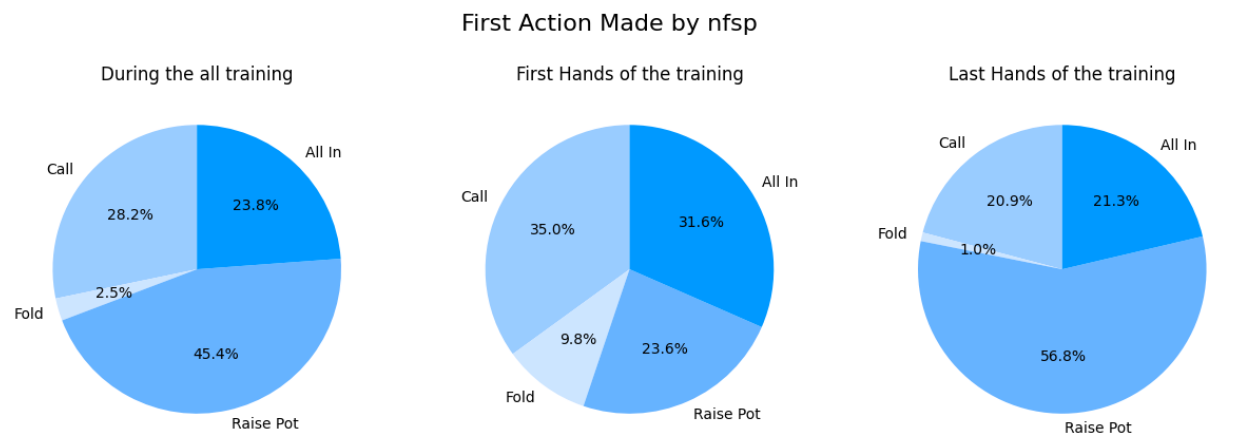


**Figure 11 NFSP Ratio of actions preflop for all datas, beginning and end**

We immediatly see the difference with DQN if we compare the end of the training, NFSP goes all in twice more than DQN, call 30% compare to 46% for DQN, raise 49% compare to 40% for DQN, and fold 3 times less

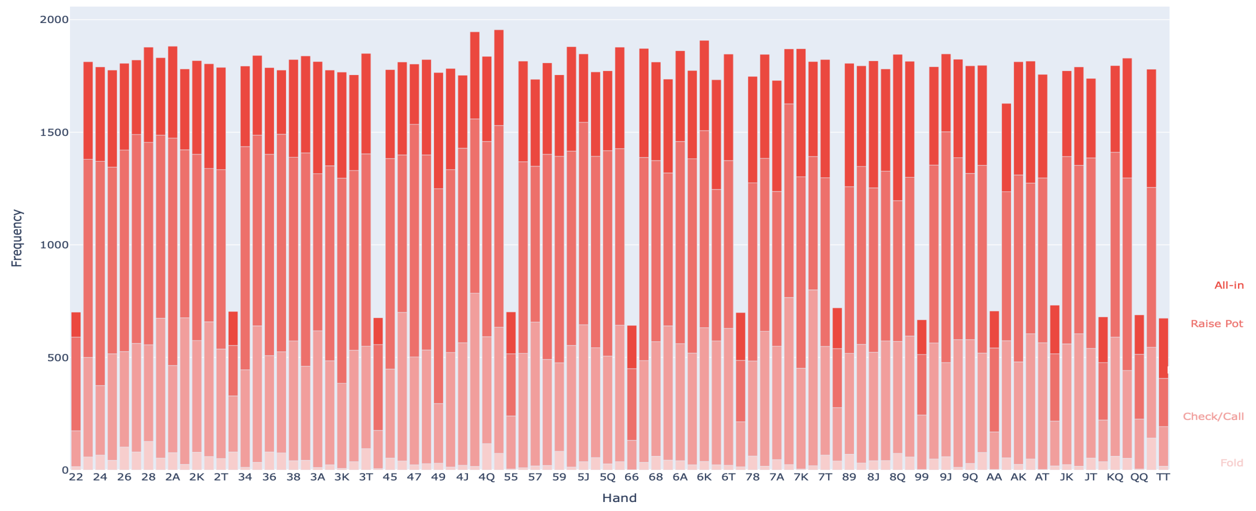


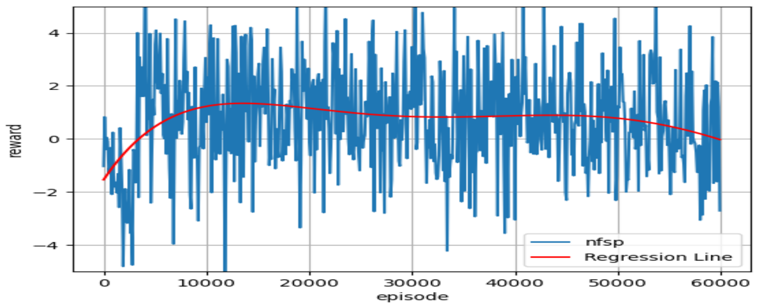
**Figure 12 Frequency of each hand postflop and Actions for all data in NFSP**

Contrary to DQN we don't see a difference between the good hand and the bad ones, it's like NFSP did learn the different strength of each hand. The 2 things learn seems to be that he should almost never fold for the first action, and raise more often than the beginning of the training.

## 5.5. Results Against Multiple Opponents

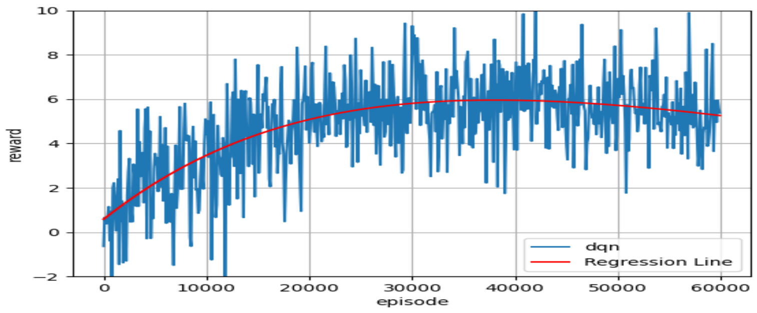
We attempted to optimize NFSP and DQN to play against two opponents. Previously, the agent only played against a randomly acting opponent. Now, it will face both a random agent and an agent that always calls.

Theoretically, NFSP is expected to perform better when playing against multiple opponents. We only run 60000 observations cause we saw earlier than they both hit a plateau around 20000 observations.



After 10000 observations a nfsp Agent as in average a **reward of 0.78**. That's even worse than against 1 random agent!

**Figure 13 Trajectories and regression of NFSP against 2 agent during the training.**



After 20000 observations, a DQN Agent as in average a **reward of 5.7**. It's almost the same as when he was playing against a random agent.

**Figure 14 Trajectories and regression of DQN against 2 agent during the training.**

Additionally, we could present various graphs to analyze the distribution of hands played against two different players. However, the primary objective was to determine whether NFSP could surpass DQN in performance, which it did not.

## 5.6. Comparative Analysis of Results

Upon reviewing the performance of various agents, including DQN, a simple agent, "special" agent, NFSP, and their performance against multiple opponents, several noteworthy observations emerge.

**DQN vs. Simple Agent :** The DQN agent significantly outperformed the simple agent, with an average reward of 5.61 compared to 2.16 achieved by the simple agent. This substantial performance gap underscores the effectiveness of more complex learning algorithms over deterministic strategies.

**DQN vs. "Special" Agent :** Surprisingly, the "special" agent, while not optimized for optimal play, achieved a marginally higher reward of 6.86 compared to DQN's 5.61. This raises questions about the limitations of the DQN agent's learning capabilities and suggests avenues for further investigation into alternative learning approaches.

**DQN vs. NFSP :** While the DQN agent exhibited a higher average reward of 5.61 compared to NFSP's 4.56, NFSP displayed distinct strategic differences. NFSP demonstrated a higher propensity for aggressive actions, such as raising and going all-in, suggesting a different approach to strategic decision-making.

**NFSP vs. Multiple Opponents :** In the context of playing against multiple opponents, NFSP unexpectedly exhibited poorer performance, with an average reward of 0.78 compared to DQN's 5.7. This divergence from theoretical expectations underscores the complex dynamics at play in multi-agent scenarios and highlights the need for further exploration into effective training strategies.

These comparative analyses highlight the nuanced differences in performance and strategic decision-making among different agents. Further research into the underlying mechanisms of each algorithm and their adaptability to diverse gaming environments is essential to fully harness their potential in real-world applications.

### 5.7. Improve the models and play against the models

In the python notebook you have the possibility to tuned some parameters of the different agent such as hidden layers, Q-MLP layers, discount factor, target estimator update frequency, learning rates for reinforcement and supervised learning, anticipatory parameter, batch size, training frequency, minimum buffer size for learning, replay memory sizes, epsilon values for exploration, batch size for Q-learning, training frequency for Q-learning, save frequency...

For now, we may not have found the perfect model, but we have the capability to play against any model created, to see if you can defeat them. While the models won't learn from you, this allows us to observe if the models make erratic moves or if their behavior makes sense in real-life scenarios. If you're interested, please refer to the References.

## 6. Limitations

One of the main limitations encountered in this study is the understanding of the underlying algorithms. Due to this, it is challenging to identify the precise modifications needed to enhance the learning performance of DQN or NFSP. If we manage to fine-tune NFSP or DQN correctly, the results could vary significantly. According to the literature, both have the potential to outperform the majority of human players in poker. However, the time constraints imposed by the extensive effort required to develop a bug-free program prevented a deeper exploration of these algorithms. We opted to exclusively display the first decision of each round because the initial decision preflop holds more significance than the final decision in the river stage, especially considering that many games do not progress to the river. However, this approach can be extended to analyze decisions at any stage of the game for a more comprehensive analysis if needed.

## 7. Conclusion

In conclusion, our comparative analysis of various agents, including DQN, a simple agent, a ”special” agent, and NFSP, provided valuable insights into their respective performances in poker gameplay.

Firstly, the superiority of DQN over a simple agent was evident, highlighting the effectiveness of complex learning algorithms compared to deterministic strategies. However, the surprising performance of the ”special” agent, despite not being optimized for optimal play, raised questions about the limitations of DQN’s learning capabilities and suggested the need for further investigation into alternative approaches.

While DQN exhibited a higher average reward compared to NFSP, NFSP demonstrated distinct strategic differences, particularly in its propensity for aggressive actions. However, NFSP unexpectedly exhibited poorer performance against multiple opponents, underscoring the complexities of multi-agent scenarios and the necessity for exploring effective training strategies further.

Additionally, our study revealed the potential for playing against models to understand their decision-making processes at different stages of the game. This insight offers opportunities to enhance our own gameplay strategies by learning from model behaviors, without encountering the black box problem associated with other algorithms in data science.

Overall, while reinforcement learning for poker presents challenges in tuning and understanding underlying algorithms, our findings underscore the importance of continued research and exploration to unlock the full potential of these models in real-world applications.

### 7.1. Proposed Future Directions

1. Understand the deeper mechanisms of these algorithms to build a better model.
2. Investigate the possibility of training the model using a database of poker players who played in tournaments between 1995 and 2000.
3. Develop a small website where players can challenge the model, thereby creating a database with real player interactions instead of random opponents. The model will be updated every X months and will continue to challenge the players.
4. Utilize the developed model as a real-time assistant during a poker game. By showing the current state of the game, it can suggest the best possible actions.

## Acknowledgments

The author gratefully acknowledges the support of the University of Lausanne, the professor Simon Sheidegger, the teaching assistants Mariapia Lombardo and Anna Smirnova.

## 8. Appendix

ChatGPT was utilized to debug and improve some functions during the Python programming phase; however, it often struggled with resolving critical bugs.

## 9. References

1. Dahl, F. A. (2001). *A Reinforcement Learning Algorithm Applied to Simplified Two-Player Texas Hold'em Poker*. DOI: 10.1007/3-540-44795-4_8
2. Zhao, E., Yan, R., Li, J., Li, K., & Xing, J. (2022). *AlphaHoldem: High-Performance Artificial Intelligence for Heads-Up No-Limit Poker via End-to-End Reinforcement Learning*. DOI: 10.1609
3. Heinrich, J., & Silver, D. (2016). *Deep Reinforcement Learning from Self-Play in Imperfect-Information Games*. DOI: 10.24193/subbi.2020.2.03
4. Shi, D., Guo, X., Liu, Y., & Fan, W. (2022). *Optimal Policy of Multiplayer Poker via Actor-Critic Reinforcement Learning*. DOI: 10.3390/e24060774
5. *Hand Strength*. World Series of Poker. Retrieved from https://www.wsop.com/poker-hands/
6. *Our GitHub Page with the Python Notebook*. Retrieved from https://github.com/Supertux007/Poker-AD-AP
7. *Library RLCard Used*. Retrieved from https://github.com/datamllab/rlcard/tree/master