Ecole Polytechnique Fédérale de Lausanne

Laboratory of Artificial Intelligence

Poker Agent : A reinforcement learning approach

Master Semester Project

Author:
Arnaud Duvieusart
Henry Decléty
Matthieu Baud

Supervisor:
Boi Faltings

February - June 2020

Contents

Abstract

This paper shows several approaches of reinforcement learning to optimize the decision making process for the Texas Holden No Limit Poker. Due to the complexity of the original game several simplifications were made to reduce the state space. In this report we describe the game we played, the tricks we used to represent the possible states and actions, 3 distinct techniques of Q-learning and their results against different baselines.

1 Introduction

1.1 Context

Over the past decades, machine learning has shown tremendous results on many games and beated human players on disciplines such as chess and Go. The applications of AI to games became a big part of AI research to develop new models, especially in Reinforcement learning. In general, games require complex strategies. It is only natural that the potential of machine learning was going to be used on one of the most popular game in the world : Poker.

1.2 Motivation

Games can be classified as stochastic or deterministic. For instance, the shuffling of cards introduces randomness into the game which produces a stochastic game. Beside other games like chess or go have deterministic behaviour which makes them more predictables.

Games with incomplete informations provide a good framework to understand and develop new models which can be applied to other real life situations. Poker is one of the most well-known betting game, John Forbes Nash even used it to demonstrate his concept of Nash-Equilibrium.

Furthermore Poker is an inherently psychological game. It is important to understand the opponent behaviour and adapt the strategy consequently.

1.3 Challenges

Poker stands out from other games because it contains a big part of uncertainty. Even the best player of all time would lose a certain number of hands against a complete beginner. This randomness hardens considerably the training and the validation of a policy. Indeed, the same action in the same moment of the game can result to excellent or disastrous outcomes, requiring thus a long training. Furthermore no indicator of reward is available until the end of a turn which makes the learning of early stages very difficult. Finally bluffing is a real aspect of poker and clouds the decision making process.

From these reasons several challenges can be defined for Poker. There are hidden cards on a Poker Game which produce incomplete information and create a stochastic process and probabilistic solutions. The understanding of the opponent can be a good compensation for this lack of information. At the beginning of the game, players' hands constitute a pre-score that strongly impact the result of the game. Since we don't know the opponent's hand, it becomes important to evaluate your own hand and compute its strength. Hand evaluation seems to be very difficult considering the number of combinations.

Antother important challenge is the size of the state space. The simplest version of No- Limit Texas Hold'em has $9.37 * 10^{71}$ decision points which would require about $1.241 * 10^{49}$ yottabytes of memory to store a full strategy [M. Johanson, "Measuring the Size of Large No-Limit Poker Games", University of Alberta, 2013 (Technical report, unpublished)]. For this kind of game tree it is absolutely necessary to use the concept of abstraction. We decided to reduce the version of our poker game and define abstraction variables as states components.

Finally we choose a variable abstraction for betting. Since the amount of bet is continuous we should discretize it in order to reduce the state space. In our game an agent can choose a factor value for bets and raises.

2 Poker game

The game we focused on during this project is a simplified version of the *No Limit Texas Holdem.* Each player is distributed 2 cards and 3, then twice 1 board cards are displayed successively. At each round players are betting according to their hand and visible board cards. There is a forced initial bet called *BLIND* as an incentive to play the round. Possible actions are *CHECK, CALL, RAISE, FOLD.*

- *CHECK* : The player does not open a bet
- *CALL* : The player call the opponent's bet
- *RAISE* : The player raises a bet higher than the previous one
- *FOLD* : The player decides not to follow the opponent and loses the round

If no player has folded the player with the highest hand wins the pot. As explained above, this game is very complex. It was decided to make the following simplifications :

1. The deck of cards is composed of 8 cards of each color, reducing the number of games by a factor $\frac{\binom{52}{9}}{\binom{32}{9}} \simeq 130$
2. The games are opposing 2 players
3. The bets are multiple of *UNIT_BET*
4. The possible raises are with a factor 1,2,3,4

3 State/Action representation

State abstraction is key in *Q-learning* since the state is solely responsible in the determination of the action to take. We should thus identify the different aspects of the game that are correlated to the decison making process. But at the same time, we must keep our state space "simple" enough to allow exploration. We chose to keep in our state :

- The player's position : representing the *strength* of a hand. This quantity is explained in the next subsection
- The total pot, i.e. the gain possible
- The number of cards on the board representing the step of the game
- The amount to call, i.e. the money the agent has to bet to stay in the game

Hand	Probability
Straight Flush	0.0311%
4 of a kind	0.168%
Full House	2.6%
Flush	3.03%
Straight	4.62%
3 of a kind	4.83%
2 pairs	23.5%
Pair	43.8%
High Cards	17.4%

Figure 1: Hands probabilities

- The last move made by the opponent, i.e. the move type and its value if applicable
- The player and the opponent's pot

This representation implies a simplification of the decision making process because we pruned some important information from the state space such as the values of the cards on the board or on our hand but this would require an enormous increase in the state space or card abstraction which didn't seem like a good approach for a game as poker.

3.1 Position simulation

As indicated in section 3 a component of our state abstraction represents the hand's position or strengh. A human player as a very natural sense of this value (though it is not always obvious to tell which hand as a higher probability of winning). Still we need a scalar indicator of how likely a particular hand is to win given the visible elements of the game. The exact value of this probability is : given my hand and the visible board cards what is the ratio of possible games won. However for a deck with N cards, there are $\binom{N}{9}$ possible games (2 cards per players and 5 board cards). To compute the above probabilities one would need to check the hand's strength of both players for each of those possible games. Then for each $\binom{N}{2}$ possible hands, there are $\binom{N-2}{3}$ flops and for each of those there are $N-5$ truns and $N-6$ rivers. For every of those situations, we would need to compute the ratio of won games. Even for our simplified verion of the game it is impossible to compute all those probabilities.

Thus we decided to use a stochastic approach and to only have an estimation of this probability. Thus for a given hand we simulate 100 games. On 20 such simulations, the standard deviation of the estimated probability of victory was 0.05 before the flop to 0.2 after the river. It was decided that 100 games was a good trade-off between efficience and precision.

Another issue was to detrmine, given a hand and the board cards what was the best *combination*. The first approach was to check all possible combinations from the best to worst and stop when a match was found. This revealed to be quiet inefficient since we are systematically checking for excellent hands which are very rare. As shown in Figure 1 most of the time a player will have either a pair, 2 pairs or a high card. It is important to minimize the number of step needed to check for those possibilities (**note :** Figure 1's probabilities are approximations for 52 cards deck poker).

Furtermore some checks can be combined, if a player doesn't have a pair it is impossible that he has either a double pairs, three of a kind, a fullhouse or four of a kind. Figure 2 shows the process to determine the strength of a hand.

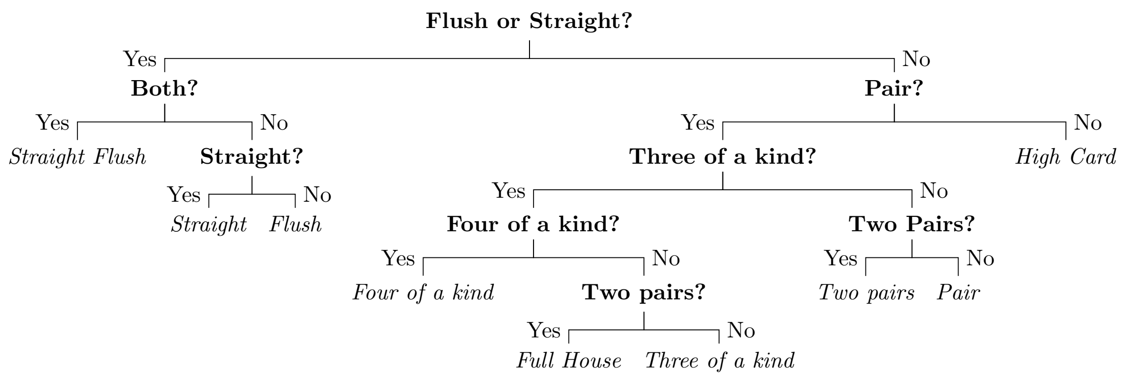

Figure 2: Hand check process

4 Approaches

4.1 Monte Carlo

4.1.1 Model-free learning

In the game we are trying to learn, and more generally in any poker game, players don't have prior information of the environment and all the information is essentially collected by experience. In practice, that means that the transition probabilities from one state to another (or the so called model of the environment) are not known beforehand.

One of the simplest method to learn such model-free games is probably Monte Carlo. Similar to dynamic programming approaches, there is a policy evaluation (finding the value function for the epsilon-random current policy) and policy improvement step (finding the optimum policy).

4.1.2 Rewards and exploration

With this method, the exploration is mostly guided by the current policy and therefore by the rewards encountered. Of course, to ensure an unbiased exploration, we use an epsilon-greedy approach. But epsilon being small, rewards still play a big role in the exploration, and when well designed they allow a smart exploration, which is key in such games where the size of the state-space is the issue.

In our Monte Carlo implementation, an episode correspond to a full game of poker with 3 rounds. Therefore the final reward gets much discounted when it gets to the first actions of the episode. In order to create more frequents rewards, we designed a simple reward shaping function representing the expected reward for each state, given the player's position and the current pot.

4.1.3 Limits of the discrete approach

Our Monte Carlo approach gives decent results, but some things can be improved, starting with the scattered structure of actions and states. Indeed, this approach doesn't ever account for the distance/similarity betweens states or actions, and therefore each sample can only be used teach something about the exact states and actions traversed, no interpolation or extrapolation can be made.

4.2 Deep Q-learning

In this section we study the possibilities offered by deep Q-learning. In this approach, we try to approximate the Q values using Neural Networks. The Networks used take as input the hand's position simulation, the total pot, the number of board cards visible, the agent's stack, the opponent's stack and the amount to call. We try to predict the best action to take for every input using 3 hidden layers of 16 neurons. This network is our *policy network*. Two approaches were studied :

- A continuous output to predict the amount that should be bet at this point
- A discrete output to predict the exact action (*FOLD, CHECK, CALL, RAISE*) to take at this point

Both approaches work the following way :
The goal is to approximate the reward $R_{s,a}$ our agent can obtain doing action a in state s and use those as sample weights for our Neural Network. This estimation is a linear combination of two intermediate estimations : E_1, E_2
E_1 is theoretical and is computed at each action our agent takes the following way.

$$E_1 = p_{call} * (p_{win} * P_1 - (1 - p_{win}) * P_1) + (1 - p_{call}) * P_2$$

With

$$p_{call} = \begin{cases} 1, & \text{if my action is CALL or FOLD} \\ \sim 1/bet, & \text{otherwise} \end{cases}$$

$$p_{win} = \begin{cases} 0, & \text{if my action is FOLD} \\ \sim 1/position, & \text{otherwise} \end{cases}$$

And P_1, P_2 are respectively the pots with and without my bet.
On the other hand E_2 is practical and is the actual gain (or loss) of the turn discounted by a factor γ at each action.
Thus, our final estimate is

$$R_{s,a} = \alpha E_1 + (1 - \alpha) E_2$$

In parallel, we use a *value network.* This network as the same architecture than the previous but try to predict the amount that we should win from a given state. It is trained on past episodes. At each step, the agent observes the state, the action it chooses and the expected reward $E1$. Every G games, the agent predicts a reward for every state using its *value network* and updates it using past state,reward pairs. This predicted reward is substracted from $R_{s,a}$ before training the *policy network* using the states, actions and final rewards computation as sample weights. This substraction allows our Network to tell if in a past similar situation it made a better choice.

4.2.1 Continuous approach

For this approach, the prediction of our *policy network* is a scalar representing the amount that should be bet at this point in the game. This network uses relu activations and Adam optimizer using MSE on accuracy. The output value represents more of a threshold since our game is simplefied and allow only bet of a multiple of *UNIT_BET*.

Let x be the output of the *policy network* and $x = q * UNIT\_BET + r$ be the euclidian division. For an amount to call of y ($y = 0$ if it is my turn to call) the betting rule for this agent is the following (assuming the agent has a sufficient pot):

$$Action = \begin{cases} \text{CHECK/FOLD}, & \text{if } x \leq y/2 \\ \text{CALL}, & \text{if } y/2 \leq x \leq 2.y \\ \text{RAISE}(q), & \text{otherwise} \end{cases}$$

In parallel, with probability ϵ the action is randomly sampled for all possible actions to enhance exploration

4.2.2 Discrete approach

For this approach, the prediction of our *policy network* is an encoding of an action to take. For this approach, action had to be discretized and it was decided to keep 6 possible actions : *CHECK, CALL, RAISE(1), RAISE(2), RAISE(3)* and *FOLD*. Since this is now a classification problem our output layer has now a softmax activation and catergorical crossentropy loss was used. The *value network* was similar to the continuous approach. Additionnaly to equilibrate the exploration/exploitation tradeoff, entropy regularization was introduced the following way : we define an entropy cost E_c and introduce an additionnal loss

$$-E_c * H(A) = -E_c * \sum_{a \in A} p_a * log(1/p_a)$$

where $H(A)$ is the entropy of the actions distribution probability. This way we penalize high entropy because high entropies mean large differences in actions probability and thus less exploration.

4.3 Clipped double DQN

4.3.1 Model

In reinforcement learning with discrete action space, using generalizing function approximation methods such as artificial neural networks tend to produce overestimations. This systematic overestimation introduces a maximization bias in learning. And since Q-learning involves bootstrapping when learning estimates from estimates, this overestimation become an issue to fix. The double Q learning try to fix this issue by updating the Q values of one table by another one.

For each environment step we select the state and select the action. A epsilon-greedy for chosing actions and favorise the exploration. Then we execute the action to observe the next state and retrieve the reward.

On this batch of episode keeping the information if we are in the final state we can update the Q values as follow:

$$Q^*(s_t, a_t) = r_t + \gamma min_{i=1,2} Q_{\Theta i}(s_{t+1}, argmax_{a'} Q_{\Theta i}(s_{t+1}, a'))$$

At this tep we update the network by computing the loss thanks to the MSE over the Q values then apply the optimizer backward procedure.

4.3.2 Opponent modelling metrics

One interesting feature of poker is to determine the adversary behavior. This can be done by creating an opponent modeling classification. One of the most used player classification is the Sklansky classification [4].

From the differents metrics proposed by Sklansky the two most important used first evaluate whether the adversary is aggressive or passive and the second determine the tight or loose behavior.

- **Pre-Flop Raise** : a percentage measure of how often a player raises pre-flops:

$$PFR = \frac{N_{flops} * 100}{N_{games}}$$

- **Aggression Factor**: the ratio between raises and call actions without the checks:

$$AF = \frac{N_{bets} + N_{raises}}{N_{calls}}$$

4.3.3 State specification

A first implementation tried to keep the 3 last moves of the games but in this state format the number of states combinations become too high. Therefore a more convenient number of variables To reduce the space of the states we decided to keep a simpler representation:

- Position of player cards
- Total pot value
- The boards cards (in a categorical format)
- The stacks of each players (2 variables)
- The value to be call
- Opponent metrics: pre-flop raise
- Opponent metric: aggression factor

4.3.4 Network Architecture

The architecture of the two networks consist of :

- **Layers** :
  4 layers with size (10, 16), (16, 16), (16, 16) and (16, 6). An uniform distribution is used to initialise the weights of the two networks. With 10 state variables and 6 discretized actions.
- **Cost function** :
  The well-known MSE cost function is to minimized the squared distance between optimal values:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

- **Optimizer** :
  Adam Optimizer update the network to minimize the MSE along the weights. Adam was designed to combine the advantages of Adagrad, which works well with sparse gradients, and RMSprop, which works well in on-line settings. Having both of these enables us to use Adam for broader range of tasks.

For computing the loss, we compute the current-state-Q-values and the next-state-Q-values of both models, but use the minimum of the next-state-Q-values to compute the expected Q value. Finally we update both models using the expected Q value.

5 Training

For each agent, the training is broken down in several steps :

1. Train our agents against *dummies*. These *dummies* are agents acting according to a simple static policy
2. Train our agents against copies of themselves

5.1 Dummies poll

Our first implementation was the following poll :

- **Aggressive** : If the win probability simulation is above 0.5 the agent raises. Otherwise it checks/folds
- **ALL IN** : This agent raises to ALL IN
- **CALL** : This agent checks or call
- **CHECK/FOLD** : This agent only checks/folds, never calls or raises
- **RAISE** : This agent always raises with a factor 2
- **Mirror** : This agent acts like a mirror, i.e. it replicates the actions of the opponent
- **Random** : This agent samples uniformly his action from the set of possible moves

However these *dummies* were too *dumb* to look like human players, even beginners. They were first used to test the implementation of our environment then it was decided to keep less *dummies* but to enhance their policies. The final *dummies* rely on the position simulation. We kept the following :

- **Aggressive**
- **Moderate** : This agent calls if the position is in $[0.4; 0.6]$ and raises proportionally to the position if it is above 0.6. It folds otherwise
- **Bluffer Aggressive** : Aggressive agent but with bluffs with probability 0.5, i.e. *RAISE(1)* on flop and *RAISE(2)* on turn
- **Bluffer Moderate** : Moderate agent with bluff

5.2 Mirror training

Once the training against the dummies was completed and simple baselines policies were learnt, the agents started the second step of training : mirror training. The agents played several games and updated several time their policy against a static copy of themselves. Then the copy was updated and the process would start again.

6 Evaluation and metrics

6.1 Evaluation against agents

As explained above, evaluation of a game based on randomness is very difficult. Each agent played 40 games of 10 rounds each against all others agents and all four *dummies*. This number was chosen considering time and complexity constraints.
Similarly, it is not obvious what metrics are to be used since several are of interest. Our agents try to maximize their gains so it was obvious that we would give a special care to the average game gain. And since randomness plays a key role, this amount will be put in perpective by the standard deviation of the different gains. Finally another key metric is the win rate as it gives a general idea of performance. Human evaluation was also considered as it is the most natural evaluation. To this end, a player graphical interface was developped. For our evaluation we used starting pot of players of 10000, *UNIT_BET* was set to 1000 and *blinds* to 500

6.2 Evaluation against humans

Another way to evaluate our agents is to play against human players. To do so we created a User Interface with the QT library for python. This interface implements our simplify version of poker. A human can play against each agent and dummies. The human player cannot see the agent's hand except for the case where the game reach the end of the river and each player should show their cards to play. Figures 3 and 4 are some game examples screenshots.

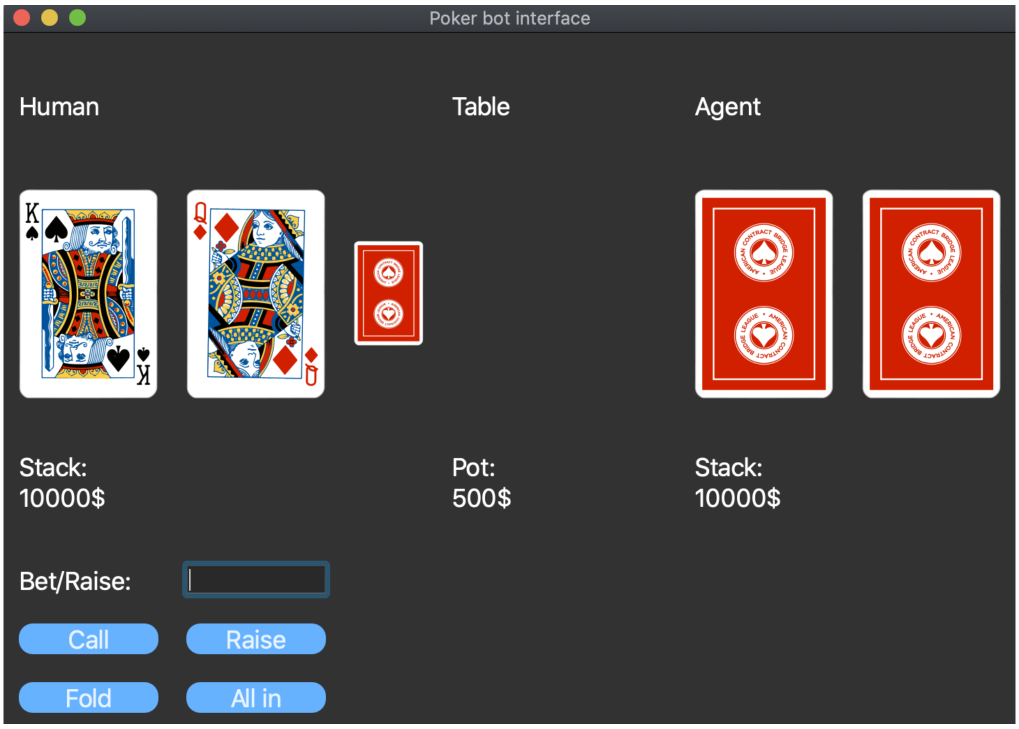

Figure 3: Initial game setup example

Unfortunately we did not have the time to play enough games with the interface to

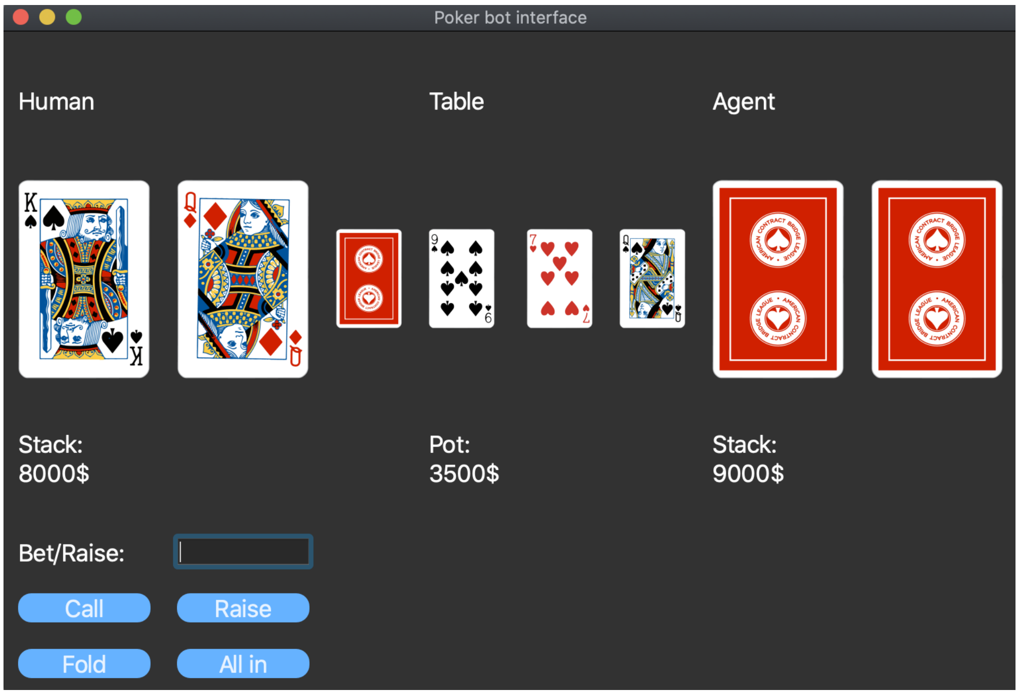

Figure 4: Post flop game example

evaluate well our different agents against human player. This step could be done as future work.

7 Results

In this section we examine the results of our different agents. We refer to them with the following code : Monte Carlo (MC), Deep Q-learning Continuous (DQC), Deep Q learning Discrete (DQD), Clipped double DQN (CDQN), Aggressive Dummy (AD), Moderate Dummy (MD), Bluffer Aggressive Dummy (BAD) and Bluffer Moderate Dummy (BMD)

Figures 5 and 6 allows us to make the following observations :

- First of all, we remark that no agents achieve unanimity, i.e. has positive average gain on all opponents
- Monte Carlo approach seems to be the more consistent and stable against a range of different strategies.
- Deep Q-learning continuous approach learnt to counter the agents it was training against but performs poorly against new strategies
- Deep Q-learning discrete approach seems to give the worst results on average but to resist to Monte Carlo
- Clipped double DQN approach performs well against deep learning approachs and have high gains variance suggesting a to be more of a gambler

	MC	DQC	DQD	CDQN	AD	MD	BAD	BMD
MC	×	500/560	0/268	3037.5/1442	337.5/466	312.5/730	-62.5/672	-187.5/730
DQC	-500/560	×	0/328	-2625/1421	125/243	37.5/342	-37.5/303	-62.5/604
DQD	0/268	0/328	×	-2612.5/1571	-225/248	-462.5/466	-487.5/236	-587.5/333
CDQN	-3037.5/1442	2625/1421	2612.5/1571	×	-875/2024	-1100/2077	-50/1278	-737.5/2024

Figure 5: Average gains/standard deviation

	MC	DQC	DQD	CDQN	AD	MD	BAD	BMD
MC	×	1	0.5	0.975	0.725	0.65	0.35	0.3
DQC	0	×	0.5	0.075	0.625	0.5625	0.4625	0.4825
DQD	0.5	0.5	×	0	0.275	0.225	0.0625	0.0675
CDQN	0.025	0.925	1	×	0.4125	0.3875	0.525	0.4625

Figure 6: Win rates

8 Discussions

The results above confirm the intuition that poker being a strategic game, some strategies will be efficient against some will weak against others. For instance Monte Carlo and Clipped double DQN approaches are extremly efficient against Deep Q-learning continuous approach, but the latter is on average better than them against the baselines. It is thus very complicated to estimate the strength of an agent without considering a particular opponent. However we have been able to implement agents with strong stable gains and low variation.

8.1 Insights

By observing the agents play and playing against them ourselves, we got some insight on the strategy it learned. On top of the intuitive move of raising when it has a good hand and folding when it doesn't, we observed the following behaviors with the Monte Carlo agent:

- The agent rarely calls, instead it mostly folds or raises.
- The agent will often take big risks and bet big portions of its stack.

8.2 Potential Improvements

Potential source of improvements include:

- Longer trainings: The state space is so big that our strategies did not completely converge.
- Combining agents to get a wider strategy.
- Perform human evaluation.
- Verify the insights with robust data.

9 Conclusion

This project gave us a great insight into the challenges of implementing agents that learn real-world games, more specifically poker. Today we have implemented a few offline learning algorithms, but it is clear and has be shown by the Libratus project, that in order to compete with real-world players, offline learning is not enough, and other modules like online subgame learning and oponent strategy learning can help get to smarter strategies.

References

[1] Thrun, Schwartz . *Issues in Using Function Approximation for Reinforcement Learning.* 1993

[2] van Hasselt, Guez, Silver. *Deep Reinforcement Learning with Double Q-learning.* December 2015

[3] Fujimoto, van Hoof, Meger *Addressing Function Approximation Error in Actor-Critic Methods.* October 2018

[4] Sklansky *he Theory of Poker: A Professional Poker Player Teaches You How to Think Like One.* 2007