# Artificial Intelligence Techniques for Poker Playing Agents

ECSE 526 – Final Project

Edoardo Holzl, Marin Thiercelin and Sébastien Gaspoz
*Department of Electrical and Computer Engineering*
*McGill University, Montreal, QC, Canada*
*{edoardo.holzl, marin.thiercelin, sebastien.gaspoz}@mail.mcgill.ca*

## Abstract

Over the years, poker has become a very popular game, obtaining media coverage comparable to sports and reaching more and more players worldwide. In the field of AI research, poker has become an important topic after Deep Blue completed the effort of chess research. This project will explore the different aspects of poker playing agents, including methods for reducing the state-space, Bayesian inference, game and utility theory as well as reinforcement learning.

## I. Introduction

The game of poker present two very interesting features from an artificial intelligence perspective. First, the game contains imperfect information, as each player only knows about its own cards and the one on the table. This affects greatly the strategy of the agent, who has to try to guess the unknown information. And even in case of a strong intuition of what the other players might hold, they might as well be bluffing. Secondly, the game involves stochastic outcomes, as the cards are shuffled, one cannot predict what is about to occur. These two properties along with the popularity of the game made it a very interesting issue for AI researcher.

## II. Texas Hold'em Poker

### *A. Poker Terminology and actions*

In Texas Hold'em Poker, every player can act in four different ways. The actions are : Check, Call, Raise, Fold. The Check is a "pass" action that does not involve betting any money, the Call is when the player places a required amount into the pot, but not more, the Raise is when the player bets a certain amount plus the amount he is required to "Call", and the Fold is when a player decides not to take part in the round anymore .

There are 10 possible hands that a player can obtain, ranked from lowest to highest: High-Card, Pair, Two-Pair, Three-of-a-Kind, Straight, Flush, Full-House, Four-of-a-kind, Straight Flush, Royal Flush.

### *B. No-Limit version*

The no-limit version of Texas Hold'em Poker involves many players playing against each other and a continuous betting amount that is only limited by the opponent's pot. This version provides much more freedom to the player and thus much more uncertainty to the game itself, making it much harder and much more complex to implement an agent that plays "correctly" .

### *C. Limited Head's up version*

The limited Head's up version of Texas Hold'em Poker involves only two players against each other, with a fixed bet amount. We've decided to implement our agents to play only this version because of time constraints and the complexity involved in implementing AI algorithms for Poker. In the version we implemented, every player can only bet once to make things simpler (the dealer can bet again if the small-blind bet, but then the small blind has to either call or fold)

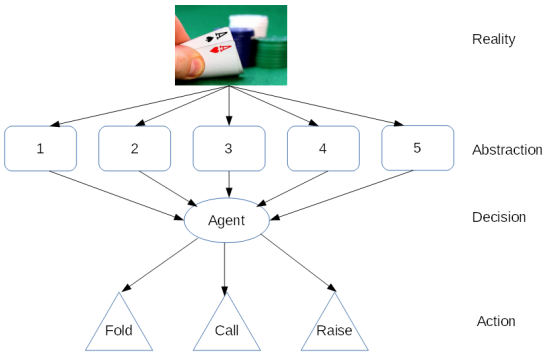
### *D. The Bucketing Abstraction*

In Texas Hold'em poker there are in the order of $10^{14}$ different observable states. Hence, to have programs that played efficiently, we had to use a game abstraction.

We used an abstraction, named Bucket abstraction. It works by computing a value called Expected Hand-Strength, based on the private cards and community cards. The hand is then classified in buckets based on that value. The assumption for our abstraction is that cards that are in the same bucket should be played with similar strategies. All our computer programs use the bucketing abstraction to play , rather than the real

Identify applicable sponsor/s here. If no sponsors, delete this text box (*sponsors*).

game. That allows efficient algorithm but also limits the performance, since an optimal player in the abstraction could be suboptimal in the real game.




## III. Naive Agents

We first created 3 basic agents:

- The Random agent that takes an action (fold, call, raise) with probability 1/3
- The All-in agent, that always raise, regardless of the situation.
- The Bucket-Agent, that maps directly from the bucket to action distribution.

These 3 agents served mainly to benchmark the other agents. The Bucket-Agent had already a reasonable behaviour, but they have a common weakness, they don't consider all the information they have: Random and All-in are playing blindly, and Bucket-Agent is unconscious of his opponent's moves. We then focused on more sophisticated agents, that didn't show such big weaknesses.

## IV. Bayesian-Inference Based Agent

The Bayesian agent works in two main steps. First a phase of «learning», during which it tries to find a model that fits the opponent strategy best. Then it tries to find a counter strategy to the model it learned.

### *A. Inferring a model for the opponent*

The Model the agent use to represent the strategy of his opponent is based on 2 main assumptions.

1: The opponent's strategy is based on the bucket abstraction
2: The opponent chooses its actions based only on his cards and the community cards, and is blind to our agent's actions.

We can represent the model we trying to fit as the following pseudo code:

```
Choose_action(cards,community_cards) = {
        i <- find_Bucket(cards,community_cards)
        Choose to Raise with probability Teta1(i)
        If did not raise, then{
                Choose to Call with probability Teta2(i)
                If did not call, then{
                        Fold
                }
```

This model has 10 parameters:

Teta1(i), Teta2(i) for i = 1, … ,5. Basically, we consider the opponent's strategy as 2 Bernoulli experiments.

### *B. The Inference Algorithm*

Prior distributions:

We choose independent beta distributions for all parameters:
P(Teta) ~ Beta (a,b) with initial parameters a=b= 5.

Algorithm to get the posterior distributions:

At each end of a round the agent make 2 observations:

- The cards the opponent had during this round.
- The list of action

The agent then use this information to obtain the posterior distributions, using Bayes' rule:

$$P(A \mid B) = \frac{P(B \mid A)\,P(A)}{P(B)},$$

Which gives us the following update rule:

```
i = bucket of the opponent this round
a(i,1) <- a(i,1) + # of raise
b(i,1) <- b(i,1) + # of call + # of fold
a(i,2) <- a(i,2) + # of call
b(i,2) <- b(i,2) + # of fold
```

The A-posteriori distribution is still a beta distribution with the updated parameters.

Estimation of the parameters:

We used a MAP (maximum a posteriori) estimate to get an approximation of the model parameters

P( Θ | actions and cards ) ~ Beta( a', b' ) -> MAP(Θ) = (a' - 1) / ( a' + b' - 2)

### *C. Results*

Here are some results of the model estimation part : On the right the true model, on the left the model guessed after 50 games.

| Bucket | P(Fold) | P(Call) | P(Raise) |
|---|---|---|---|
| 1 | 0.5 | 0.45 | 0.05 |
| 2 | 0.3 | 0.6 | 0.1 |
| 3 | 0.1 | 0.6 | 0.3 |
| 4 | 0.02 | 0.5 | 0.48 |
| 5 | 0 | 0.4 | 0.6 |

| Bucket | P(Fold) | P(Call) | P(Raise) |
|---|---|---|---|
| 1 | 0.1635 | 0.7581 | 0.0783 |
| 2 | 0.1473 | 0.6817 | 0.1709 |
| 3 | 0.0575 | 0.6125 | 0.3299 |
| 4 | 0.0156 | 0.6181 | 0.3661 |
| 5 | 0.0063 | 0.5671 | 0.4264 |

| Bucket | P(Fold) | P(Call) | P(Raise) |
|---|---|---|---|
| 1 | 0.4 | 0.4 | 0.2 |
| 2 | 0.25 | 0.5 | 0.25 |
| 3 | 0.1 | 0.5 | 0.4 |
| 4 | 0.02 | 0.4 | 0.58 |
| 5 | 0 | 0.2 | 0.8 |

| Bucket | P(Fold) | P(Call) | P(Raise) |
|---|---|---|---|
| 1 | 0.1493 | 0.6346 | 0.2160 |
| 2 | 0.0975 | 0.5798 | 0.3225 |
| 3 | 0.0576 | 0.5570 | 0.3852 |
| 4 | 0.0199 | 0.4739 | 0.5060 |
| 5 | 0.0070 | 0.3803 | 0.6126 |

We can see here that the agent is able to recognize that an agent is a Careful player ( model 1 bluffs 5% of the time) and a Bluffer (model 2 bluffs 20% of the time). The agent can then use this information to its own profit.

### *D. Playing a counter-strategy based on the model*

Estimating the Hand-Strength of the opponent:

The first step of the counter strategy is to use the model, built during the "learning phase", to get an estimation of the strength of the hand the opponent holds. Based on the past actions, we compute the likelihood of this sequence of action, for each possible bucket the opponent could be in.
We use the maximum likelihood estimation for the bucket, and measure the correctness of this approximation by the ratio of the likelihoods of the maximal and second maximal bucket ( 1 meaning that we have no clue).

Example:

We trained one Bayesian agent against the All-in agent and the other one with a careful Agent. We then asked the agent to estimate the card of the opponent based on the action sequence : {bet, bet, bet}

The first agent made the guess bucket 1 with evidence : 1.01
The second agent made the guess bucket 5 with evidence : 5.1

This shows that the agent knows to ignore the actions of the All-in agent (because the actions of All-in agent are independent of his cards ) , but is much more alarmed when the careful player raises 3 time in a row.

Playing based on the hand strength estimation:

The agent can then play not only based on its cards but also using the information it gets from the opponent's actions.
The Bayesian agent basically maps from his bucket, the opponent's bucket (estimation) and strength of estimation, to action probabilities.
If we have enough evidence we take a decision based on our cards and on our estimation, otherwise we focus only on our cards.

## V. Reinforcement Learning Agent

The first reinforcement learning algorithm that came to our minds was the Q-learning algorithm which at first seemed suitable to tackle Texas Hold'em playing.

The first and perhaps most important aspect of any Machine learning algorithm is the abstraction of states. As mentioned earlier, the game of poker is a non-deterministic game where many parts of the entire game state are hidden and thus it is impossible for the agent to be omniscient and know the actual state. As a first part of our development, we will present the state abstraction used for the learning algorithms.

### *A. State Abstraction*

Since the actual game state is invisible for the agent, our representation of states is limited by the information the agent can perceive at a given point of the game. The states we used are a combination of the current game stage, the bucket the agent is currently in and the opponent's previous action (if any); that is, a state is a tuple [Stage, bucket, opponent's previous action], and only "Checks" and "Bets" are considered for the opponent's action. By using this scheme, we end up having 4*6*2 = 48 states which is considerably less than the humongous number of states poker has.

After a few training episodes, we've noticed that the agent needed more information about his cards for the stages of "Pre-flop" and "Flop", due to the high probability of changing buckets from Pre-flop to Flop or from Flop to Turn (e.g. having a 6 of Spades and a 7 of Spades at "Pre-flop" will end up in a bucket 0 or 1 while it could be, at later stages, in a bucket 4 or 5 with a straight or a flush). To tackle this problem, we added two new buckets at the "Pre-flop" stage and 3 new buckets at the flop stage, which are described as follows:

- Bucket 6 : for cards that are separated by at most 2 cards and are suited

Bucket 7: for cards that are separated by at most 2 cards and are off-suited

- Bucket 6 : Flush-draw (meaning he still needs one card to obtain a flush)

  Bucket 7: Open-Ended Straight-Draw (meaning he has 4 consecutive cards and still needs 1 to get a straight)

  Bucket 8: Inside Straight-Draw (meaning he is missing one middle card to get a straight)

With these added buckets, the number of states increases to 2*(8 + 9 + 6*2) = 58 states

### *B. The Learning Algorithm*

1) What is the agent Learning ?:

At first we tried implementing an agent that learns (using q-learning) the expected winnings of a given state action pair, but it didn't really work well; It seemed to beat the Bucketing based agent, but after roughly 1000 training episodes, it learned that the best action was always to fold and we thus decided to implement an agent that learns the probabilities of having a positive reward for a state action pair.

The final agent learns probabilities of winning if he chooses action **A** in state **S** (recall that a state is a combination of the current stage, the current bucket and the opponent's previous action).

The probabilities are learned based on a voting system. At the end of each round, the sign of the result is back-propagated to all the state action pairs and the utilities are updated accordingly :

- If the reward is positive, one vote is given to all the state-action pairs' utility involved in the outcome
- If the reward is negative, one vote is deducted from all the state-action pairs' utility

The best action given a state **S** is then chosen in the following way: the utilities of all actions are all scaled up to have the lowest one set to 1 (we don't want any action to have a 0 probability) and then normalized in order to interpret them as probabilities.

For example: Suppose the votes associated to state **S** are [417.0, -32.0, 1] (call, bet, fold) then they will be scaled up to become [450, 1, 34] and then the corresponding probabilities will be [0.92,0.002, 0.078].

2) Learning From Folds:

This part of the learning was a bit trickier than others: how to let the agent learn when a fold is a good one or a bad one ? That is if it ends up in the same situation should it fold or not ? A simple way to determine if a fold was good or bad is to compare the agent's hand with the opponent's and decide who would've won if he kept playing. If the agent would've won the game, then one vote will be deducted from the fold action in the state where it occurred, otherwise, one vote will be given to the fold action in the given state.

3) Check/Check, making the difference:

This section is just to describe one exception in the voting mechanism that enhances the final performance. Suppose for now that the reward was negative, and the update reached a state where the opponent's action was either nothing (because player is the first to play) or a Check, and the player's action was a Check. In that case, we only deduct half a vote from the state because a Check/Check situation does not affect the outcome as much as a Bet/Call situation and thus less attention is given to it.

4) Exploration Function:

During the first 100 training episodes, one action out of 5 was chosen at random, giving the agent the possibility to try more actions even though the utility value doesn't recommend it. By giving this degree of freedom in the beginning, the agent is forced to try new actions and learn their utility from the outcome.

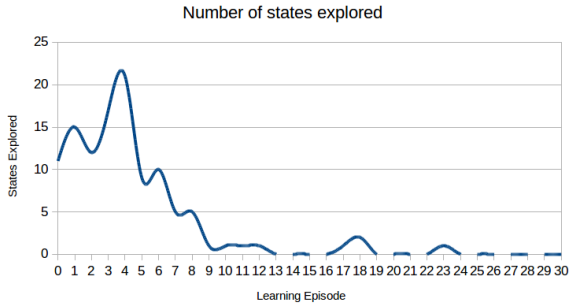
5) Learning from a teacher:

To accelerate the learning process, we implemented a Teacher agent based on the Bucket-Agent (i.e. acts as if the Bucket-Agent was playing) which transmits (if it ends up having a positive reward) its bucket and action history to the Learning agent for it to update the utilities accordingly and learn form the teacher. The learning process with the teacher was roughly 130% faster than without the teacher, saving some precious time.

### *C. Results*

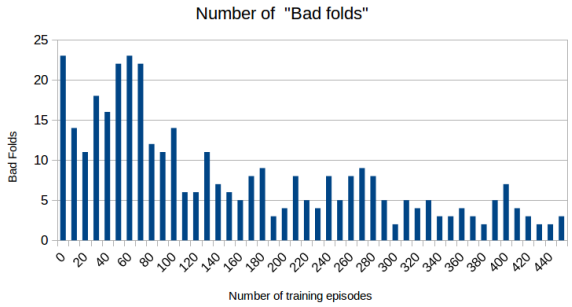
In this section we will present some results related to the agent's performance and learning process.

First let's take a look at how the exploration function helped exploring the state space:



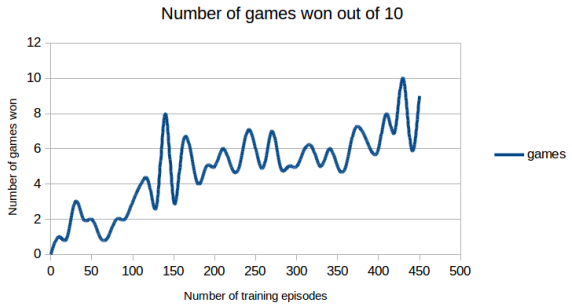

We can clearly notice how the number of explored states quickly decreases and reaches 0 after roughly 25 games.

Second, we look at the effectiveness of learning from folds:




It is also noticeable that the number of folds decreases as the number of training episodes increases, meaning that the agent now knows better how and when to fold.

Finally, we present the agent's actual performance against the Bucket-Agent.




One can also notice the improved performance with the increase of learning episodes.

# VI. Counter-Factual regret minimization agent

Instead of considering how to make our agent win, we'll consider now how to make an agent not lose. We will then ensure that in the worst case, the agent will tie against its opponent. Over time and with its opponent mistakes, the agent will end up taking the lead. The design of this agent is based on two concepts of game theory which will be explored in more depth in the next section.

The algorithm is based on what M. Johanson proposed in his thesis [1].

## *A. Theoretical Foundation*

The first crucial notion is Nash equilibrium. In non-cooperative games, this represents the situation where each player has no way to improve its gain by changing only its own strategy. Consequently, in zero sum games, when playing along an action that is a Nash equilibrium, the opponent can never decrease your gains by unilaterally modifying its actions. In a game as complex as poker, it is not reasonably feasible to reach a Nash equilibrium, so we will restrict ourselves to a ε-Nash equilibrium, that is an approximation of the former by a margin ε.

The second notion is regret, which as expected represents the amount lost as compared to the optimal play. If performing the optimal action a* and actual action a grant respectively utilities u(a*) and u(a), the regret r(a) is u(a*) – u(a). Instead of considering the overall regret over a whole hand, one can break it down into smaller counter-factual regrets, which are the regrets generated by each individual action.

Now that these notions are introduced, one might still wonder how those are linked. The two following theorems put everything together:

- In a zero-sum game at time T, if both players' average overall regret is less than ε, then, $\sigma^T$ is a 2ε equilibrium.
- The overall regret is bounded by the sum of the counter-factual regrets.

Hence, by combining these two theorems, we find that minimizing the counter-factual regrets will minimize the overall regret and hence reach an equilibrium.

### *B. Practical Steps*

The algorithm used returns a strategy for each possible situation the agent will have to face by computing the probabilities of folding, calling and raising. The algorithm works by splitting the strategy in two, a dealer part and a small-blind part. It initializes the strategies such that the agent always fold, call and bets with equal probability. It also initializes the overall counter-factual regrets as the zero array. Then, using its perfect information of the two half strategies, it updates the probabilities for each information set as follows:

1. Compute the expected value of each of the three actions $e_a$.
2. Compute the average expected value e as the sum of the $e_a * p_a$, and use it to compute each counter-factual regrets values $r_a$ by $r_a = p_{opp}*(e_a-e)$, where $p_{opp}$ is the probability that the opponent took the previous action, according to its strategy.
3. Update the overall counter-factual regret by adding the values obtained in 2
4. Update the probabilities following the following rule:
   a) If the overall regret for action a $R_a$ is negative, set the probability $p_a$ to 0
   b) Else, set $p_a$ to $R_a/R^+$, where $R^+$ is the sum of all the positives overall regrets.
5. Repeat steps 1 to 4 until convergence.

figures. Table titles are to be centered *above* the tables.

### *C. Conclusion*

The agent produced by this technique is not optimal. Indeed, because of the many abstractions used due to time constraints, all the theoretical results on which this agent was based are not necessarily met, and the convergence might be off in some cases. As we wanted to minimize the regrets to minimize the overall regret which in turn leads to a strategy that approximates an estimated optimal play, we add a lot of uncertainties. A surprising thing to note is the very aggressive play-style towards which the agent converged. Nevertheless, the agent showed great results in the competition, which is to be discussed in part VII.

## VII. Results

In order to evaluate the performance of each agent, we let the every pair of agents play 20 games against each other, each game being played twice (first and second leg) in order to be reduce as much as possible the variance induced by the factor chance. For the first and second leg, we used the same shuffled deck, the first time starting with player1 as dealer and the second time with player2 as the starting dealer. In this way, each agent gets the same possibilities, and a win cannot be considered to be the result of luck only. A win is awarded to an agent if it wins both reversed games. A draw occurs if one of the agent wins the first one and loses the return match. Here are the results of the tournament, presented in a table, with results (wins_player1,wins_player2,draws).
The competition lead to the regret agent to win undefeated (3 wins and a draw), followed by the all-in, the Bayesian, the bucket and finally the leaning agent. The low performance of the learning agent can be attributed to the fact that it was trained with the bucket agent and wasn’t able to react appropriately to the very aggressive play-style of the top agents.

## References


[1] M. Johanson, "Robust strategies and counter-strategies: building a champion level computer poker player", Master thesis, University of Alberta, 2007.

[2] D. Billings and al., "Approximating game-theoretic optimal strategies for full-scale poker", University of Alberta, In International Joint Conference on Artificial Intelligence, pp 661–668, 2003.

[3] F. Southey, M. Bowling and al., "Bayes' bluff: opponent modeling in poker", University of Alberta, In Proceedings of the 21st Annual Conference on Uncertainty in Arti?cial Intelligence 2005.

[4] N. da Silva Passos, "Poker learner: reinforcement learning applied to Texas hold'em poker", Master thesis, Universidade do Porto, 2011.

[5] E. Lingg, A. Go and B. Srinivasan, "Machine learning applied to Texas hold'em poker", 2012, Stanford University, unpublished.

[6] A. Chopra, "Knowledge and strategy-based computer player for Texas hold'em poker", Master thesis, University of Edinburgh, 2006.

[7] S. Russell and P. Norvig, "Artificial Intelligence, A Modern Approach", Third Edition, Prentice Hall, pp. 667-679, 2010.

[8] A. C. Davison, "Probability and Statistics for SIC", Ecole Polytechnique Federale de Lausanne, 2012.

[9] Wikipedia. Texas hold ’em — wikipedia, the free encyclopedia.https://en.wikipedia.org/wiki/Texas_hold_'em#Rules, 2016. [Online;accessed 04-Dec-2016].

| P1\P2 | All-in | Bucket | Bayesian | Learning | Regret |
|---|---|---|---|---|---|
| **All-In** | X | (5,4,11) | (8,3,9) | (14,2,4) | (3,11,6) |
| **Bucket** | | X | (4,6,10) | (7,7,6) | (5,5,10) |
| **Bayesian** | | | X | (7,4,9) | (3,5,12) |
| **Learning** | | | | X | (0,14,6) |
| **Regret** | | | | | X |

Table: Competition results. Results are shown as (P1 wins, P2 wins, draws) triplets