STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LX**, Number 1, 2020
DOI:

A VIEW ON DEEP REINFORCEMENT LEARNING IN IMPERFECT INFORMATION GAMES

TIDOR-VLAD PRICOPE

Abstract. Many real-world applications can be described as large-scale games of imperfect information. This kind of games is particularly harder than the deterministic one as the search space is even more sizeable. In this paper, we want to explore the power of reinforcement learning in such an environment; that is why I take a look at one of the most popular game of such type, no limit Texas Hold'em Poker, yet unsolved, developing multiple agents with different learning paradigms and techniques and then comparing their respective performances. When applied to no-limit Hold'em Poker, deep reinforcement learning agents clearly outperform agents with a more traditional approach. Moreover, if these last agents rival a human beginner level of play, the ones based on reinforcement learning compare to an amateur human player. The main algorithm uses Fictitious Play in combination with ANNs and some handcrafted metrics. We also applied the main algorithm to another game of imperfect information, less complex than Poker, in order to show the scalability of this solution and the increase in performance when put neck in neck with established classical approaches from the reinforcement learning learning.

1. Introduction

The idea that we learn by interacting with our environment is probably the first to occur to us when we think about the nature of learning [2]. We are thinking about games as simulations of our real world with special, particular features and rules, that is why, lately, this field represented the perfect playground for machine learning research. Solving particular game environments can lead to solutions that scale to more complex, real-word challenges such as airport and network security, financial trading, traffic control, routing ([7], [8], [14]). We witnessed the rapid development of computer AI with the massive

Received by the editors: 27 July 2020.
2010 *Mathematics Subject Classification.* 68T05 .
1998 *CR Categories and Descriptors.* I.2.1 [**Artificial Intelligence**]: Applications and Expert Systems – *Games*.
Key words and phrases. Artificial Intelligence, Computer Poker, Adaptive Learning, Fictitious Play, Deep Reinforcement Learning, Neural Networks.

success in perfect-information games like Chess and Go (AlphaGo Zero, 2014 [12]; LeelaChessZero 2016 [13],), but researchers have yet to reach the same progress in imperfect-information games (AlphaStar, Deepmind, 2019, [1]).

Before the 2000s, poker solving approaches could have been categorized (from an architecture point of view) in 3 main classes: expert system, game-theoretic optimal play and simulations based on enumerations [9]. One of the most successful poker bots at the time was Loki [9], which used those aforementioned methods combined with parametric models for opponent modelling. Although this approach is very far from a Nash-equilibrium, it does find locally optimal solutions to certain situations and its performance can be used as a threshold when comparing our modern ways of solving imperfect information games. Therefore, I will develop my own version of Loki Poker bot, as our first agent, in order to use for testing.

Fictitious play [4] is a popular method for achieving Nash Equilibria in normal-form (single-step) games. Fictitious players choose best responses to their opponents' average behavior. For our deep reinforcement agent, we try to improve on a variant of Fictitious play, normally used in self-play scenarios (NFSP) [6]) and show how this approach can be also re-modelled and applied to a one-player environment. It was proven that NFSP provides poor performance in games with large-scale search space and search depth [18], because of the complexity of opponents' strategy and the fact that a DQN [10] learns in offline mode and it uses only raw, crude data as input. Moreover, NFSP wasn't tested on the more complex variant no-limit. I try to address these issues by considering some high-level hand-crafted heuristics to go alongside raw data from a state of play. My approach uses, in addition, hard coded rankings of card combinations and Monte-Carlo heuristics for assessing an approximate strength of the opponent hand. This will represent the main idea behind the second agent I am going to build.

I empirically evaluate each agent in two-player (heads up) zero-sum computer poker games and explain how each one can work even in a multiple-player scheme with limited performance loss.

2. Background

In this section I provide an overview of reinforcement learning and fictitious play in extensive-form games. I am going to mark some important mathematical elements here as they will be used for reference in the next sections.

2.1. **Reinforcement Learning.**

Reinforcement learning [2] agents typically learn to maximize their expected future rewards from interaction with an environment. The environment is usually modelled as a *Markov decision process (MDP)*. Reinforcement learning

algorithms can learn in many ways, but we are interested in the ones that learn from sequential experience in the form of transition tuples from one state (s) to another taking into account the action (a) necessary to reach the new state and the respective reward of that operation (r): $(s_t, a_t, r_{t+1}, s_{t+1})$. The goal of the agents is to maximize their rewards, this is typically done by learning the *action-value function* Q, defined as the expected gain of taking action a in state s and following the policy π: $Q(s,a) = E^{\pi}[G_t|S_t = s, A_t = a]$. Here, $G_t = \sum_{i=t}^{T} R_{i+1}$is a random variable of the agent's cumulative future rewards starting from time t [2]. From this, it easily follows that we may want to take the action of the highest estimated value Q, that's why *Q-learning* [17] was invented as a way to learn about the greedy policy storing and replaying past experience. To approximate the *action-value function*, a neural network can be used and this approach is one of the most popular when dealing with more complex games and the system is called a *DQN* [15].

2.2. **Fictitious Play.**

Fictitious play [4] is a game-theoretic model of learning from self-play. Fictitious play is commonly defined in normal form, which is exponentially less efficient for extensive-form games. Heinrich et al. (2015) [5] introduced *Full-Width Extensive-Form Fictitious Play (XFP)* that enables fictitious players to update their strategies in behavioral, extensive form, resulting in linear time and space complexity. It was proven that, for a convex combination of normal-form strategies $\hat{\sigma} = \lambda_1\hat{\pi}_1 + \lambda_2\hat{\pi}_2$, we can achieve a realization-equivalent behavioral strategy σ, by setting it to be proportional to the respective convex combination of realization-probabilities: $\sigma(s,a) \propto \lambda_1 x_{\pi_1}(s)\pi_1(s,a) + \lambda_2 x_{\pi_2}(s)\pi_2(s,a)\ \forall s, a$, where $\lambda_1 x_{\pi_1}(s) + \lambda_2 x_{\pi_2}(s)$ is the normalizing constant for the strategy at information state s.

Define $\Delta(n)$ as a simplex in R^n, $v_i \in \Delta(n)$ being the i-th vertex and let $H : \mathrm{Int}(\Delta(n)) \to R$ the entropy function $H(s) = -s^T \log(s)$. In a two-player game, each player chooses its strategy $p_i \in \Delta(m_i)$, $m_i \in N^*$ and collects the associate reward given by the value-function: $V_i(p_i, p_{-i}) = {p_i}^T M_i p_{-i} + \tau \cdot H(p_i)$, where $-i$, $i \in \{1, 2, ..., n\}$ refers to the complementary set $\{1, 2, ..., i-1, i+1, ...n\}$ [11]. Note that we shall use reinforcement learning to approximate this value-function. If follows that we can define player i 's best response as a function $\beta_i : \Delta(m_{-i}) \to \Delta(m_i)$, $\beta_i(p_{-i}) = \arg\max\ V(p_i, p_{-i})$ and player i's average response until step k in the game as empirical frequencies $\pi_i(k) : N \to \Delta(m_i)$ of player P_i [11].

Depending of the game type, there are multiple Fictitious Play (FP) abstractions: in discrete time, continuous and dynamic continuous. For *discrete time FP*, we can define the strategy at step k as the best response to the

empirical frequencies of opponent actions:

$$p_i(k) = \beta_i(\pi_{-i}(k)) \quad (1)$$

In continuous time FP, the following equations are used:

$$\frac{d}{dt}\pi_i = \beta_i(\pi_{-i}(t)) - \pi_i(t), i = \overline{1,2} \quad (2)$$

The difference that comes with the third type of abstraction, in which Poker falls in, is that each player has access to the derivative of his empirical frequency $\frac{d}{dt}\pi_i$, therefore the strategy at moment t can be defined as:

$$p_i(t) = \beta_i\left(\pi_{-i}(t) + \eta\frac{d}{dt}\pi_{-i}(t)\right), \text{ with } \eta \text{ positive parameter} \quad (3)$$

We interpret this formula as a player choosing his best response based on current opponent's average strategy profile combined with a possible change of it that may appear in the future.

The authors of this study, *anticipatory dynamics of continuous-time dynamic fictitious play* [11] show that, depending on the game, for a good choice of η, the stability in Nash equilibrium points can be improved. The challenge that comes with it though is the fact that the derivative cannot be directly measured and needs to be approximated or reconstructed by empirical frequencies measurements.

3. Developing the Agents

In this section, I am going to address the technical details and the main process of building my 2 agents mentioned in the introduction. Therefore, my first agent should be a reinforcement learning free one, that's why I am going to build it as my own mini remake version of Loki [9] featuring betting decisions with card heuristics and opponent-modelling. The second agent will learn Poker training with the first agent trying to consistently beat him, treating the opponent as part of the environment.

3.1. **Agent 1.**

We construct this agent mainly as an expert system at its core with heuristics for betting decisions and opponent-modelling for exploitations. Agent 1 defines his policy π depending on the street of the game, on the hand strength metrics and whether an opponent model was found or not. We got a look-up table for the preflop stage containing the rankings of all 2-card pairs. Starting with the flop, we maliciously evaluate the win rate in a particular situation by enumerating all possible 5-card combinations with the current board and using another look-up table that contains the rankings of all *7462* such distinct combinations of card, with 1 being a *royal flush* and 7462 being *7-5-4-3-2* with at least 2 different *suits*.

This is not really enough as we need to take into account possible future hand strength increase or decrease. That's why we also compute positive hand potential by *Monte-Carlo* simulation of states that can derive from the current one and assessing those with the look-up table mentioned earlier. In order to be completely sure when to place a bet/raise, we need to analyze the limit break points in such a hand strength metric distribution. For that, I simulated 1000 games of Poker and assessed the hand strength of the best 5-card combination that includes the two hole cards. The results can be observed in the figure 1. As expected, most of the hands are really weak, but we can expect great results of our hand strength metric indicates at least *0.8/1.*

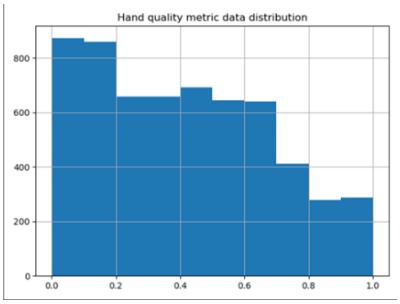


FIGURE 1. Hand strength distribution over 1000 games

Opponent modelling

The goal of this part is to find a good approximation of opponent average strategy π^{-i} with a good accuracy of predicting the *fold* moves. I do that by training two separate supervised classification models which are active during the games, collecting data about the opponent.

I use a naïve Bayes classifier (to replicate the Bayesian analysis presented in the Loki paper), after a certain number of actions taken, *min_steps_bayes*, the model will train and try to guess opponent's next move. The input for this classifier consists of a 1D array containing an expected average hand strength of the opponent (obtained from Monte-Carlo simulations), raise demand, the opponent stack, the number of consecutive and same-suits cards on the board and the street number.

I also use a deep neural network as our second classifier. I decided to use a CNN architecture, the input being represented as an image of the current board state alongside some of the scalar features mentioned at the other classifier. This will also have a *min_steps_CNN* parameter set at the beginning, usually at least two times higher than *min_steps_bayes*, after which, the model will be ready to start predicting.

The main reason that I use this configuration is that the Bayes model shines when less data is available, taking into consideration class probabilities but then is really outperformed by a neural network when much more data units are available, so in the long run, we shall keep the neural model active as we deactivate the first one.

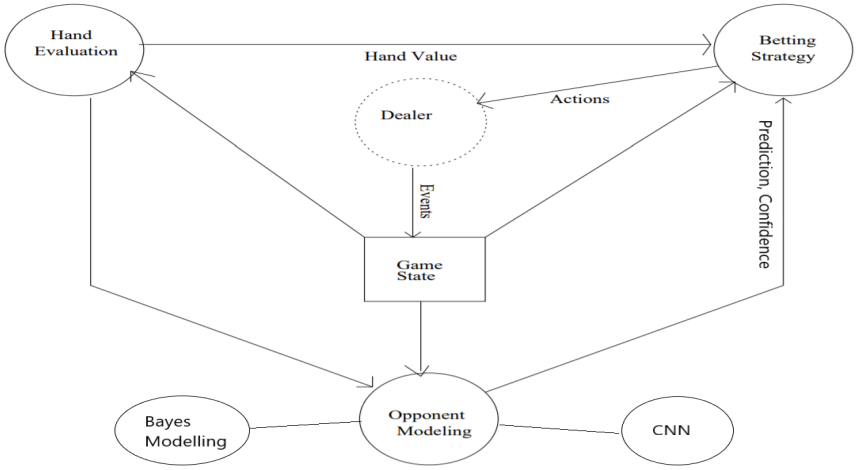


FIGURE 2. Agent 1 architecture (our mini version of Loki.)

Algorithm 1 | Agent 1, expert system with neural opponent-modelling and Bayes classifier method

---

Initialize Bayes memory M_{Bayes}
Initialize CNN memory M_{CNN}
Initialize $acc \leftarrow 0$
for 1 : nogames do
  Initialize new game G and execute agent via RUNAGENT for each player in the game
function RUNAGENT(G)
  wr $\leftarrow$ getwinrate (currentposition)
  if $a \neq \hat{a}$ then
    $nomistakes \leftarrow nomistakes + 1$
  Set policy $\sigma \leftarrow \begin{cases} acc - greedy\,(\text{exploiting} - \text{expert} - \text{system}\,(wr)\ \text{policy}), & \text{with probability } acc \\ \text{simple} - \text{expert} - \text{system}\,(wr)\ \text{policy}, & \text{with probability } 1 - acc \end{cases}$
  Observe initial information state s and opponent action a
  Store behaviour tuple (s, a) in supervized learning memory M_{Bayes} and M_{CNN}
  if $\mathrm{M}_{Bayes}.size$ % minstepsbayes then :
    $acc \leftarrow$ train Naive $-$ Bayes $-$ Classifier
  if $\mathrm{M}_{CNN}.size$ % minstepscnn then :
    $acc \leftarrow$ train Neural $-$ Network $-$ Classifier
  if agent follows targeted response policy σ=$acc - greedy$ then
    Save our prediction $\hat{a}$
end function

The agent functions as an expert system with betting decisions based on mentioned hand strength heuristics at first and then, after it collects enough data to start the Bayes classifier it changes its policy (in an accuracy greedy

way) to another expert system for exploitation. This system will be deactivated as we gather enough information to start the neural model and use this one for opponent's moves prediction. We shall choose to use the opponent modelling part based on the number of mistakes the classifies make during a few games.

This agent will be important in testing by adversarial reasoning, as it offers a measurement to opponent's exploitability through the opponent modelling part.

3.2. **Agent 2.**

This deep reinforcement learning agent will continuously learn to play Poker by training with **Agent 1** from scratch. Its strategy of play combines the greedy strategy β offered by the action-value function with the average strategy π obtained though supervised classification.

Recall the equation (3), subtracting π_i from both sides and using (1) yields:

$$\frac{d}{dt}\pi_i = \beta_i \left(\pi_{-i}(t) + \eta \frac{d}{dt}\pi_{-i}(t) \right) - \pi_i(t) \qquad (4)$$

In *NSFP* [6], the authors chose a discrete time approximation of the derivative: ${\beta^{t+1}}_i - {\pi_i}^t \approx \frac{d}{dt}{\pi_i}^t$ which, if substituted in (4) yields:

$$\begin{array}{rcl} p_i(t) & \approx & \beta_i(\pi_{-i}(t) + \eta(\beta_i(\pi_{-i}(t+1)) - \pi_{-i}(t))) \quad \Leftrightarrow \\ p_i(t) & \approx & \beta_i((1-\eta)\pi_{-i}(t) + \eta\beta_i(\pi_{-i}(t+1))) \end{array}$$

and this is how we arrive at the combined policy approach $\sigma \equiv (1-\eta)\hat{\pi} + \eta\hat{\beta}$ which was proven to be really good in practice.

Therefore, Agent 2 uses 3 neural networks. First, a *DDQN* system [15] with a *value network* $Q\left(s, a \left| \theta^Q \right.\right)$ for predicting the Q values for each action based on data from M_{RL}. It trains through backpropagation using the *Bellman equation* with future Q values obtained through a *target network* $Q'\left(s, a|\theta^{Q'}\right)$Secondly, we use a *policy network* $\Pi\left(s, a \left| \theta^{\Pi} \right.\right)$ to define our agent's average response based on data from M_{SL}. We choose our main policy σ from a mixture of strategies: $\beta = \varepsilon - greedy(Q)$ and $\pi = \Pi$: $\sigma \equiv (1-\eta)\hat{\pi} + \eta\hat{\beta}$, $\eta \in (0, 1]$.

Observe that the algorithm used has general scope and may be used in other games, MDPs with imperfect information or to practical real-life problems.

Note that we can also set the main policy σevery step t in the game for a more stochastic approach. I will actually do that in the experiments to test this small change to the NFSP algorithm.

The neural networks for the two strategies are implemented as CNNs and will have mainly the same architecture, the only difference appearing at the last layer

Algorithm 2 | Agent 2, reinforcement learning agent with fitted Q-learning

for 1 : nogames do
Initialize new game G and execute agent via RUNAGENT for each player in the game
function RUNAGENT(G)
Initialize replay memories M_{RL} (*circular* buffer) and M_{SL} (own behaviour dataset)
Initialize average − policy network $\Pi\left(s,a|\theta^{\Pi}\right)$ with random weights θ^{Π}
Initialize action − value network $\mathrm{Q}\left(s,a|\theta^{Q}\right)$ with random weights θ^{Q}
Initialize target network with weights $\theta^{Q'} \leftarrow \theta^{Q}$
Initialize $\pi-\beta$ parameter η
for each episode do
Set policy $\sigma \leftarrow \begin{cases} \varepsilon - greedy\left(Q\right), \text{ with probability } \eta \\ \Pi, \text{ with probability } 1-\eta \end{cases}$
Observe initial information state s_1 and reward r_1
for $\mathrm{t} = 1, \mathrm{minreplaymemorysize}$ do
Sample action a_t from policy σ
Execute action a_t in emulator and observe reward r_{t+1} and next information state s_{t+1}
Store transition $\left(s_t, a_t, r_{t+1}, s_{t+1}\right)$ in reinforcement learning memeory M_{RL}
if agent follows best response policy $\sigma = \beta\left(= \varepsilon - greedy\left(Q\right)\right)$ then :
Store behaviour tuple $\left(s_t, a_t\right)$ in supervised learning memory M_{SL}
Update θ^{Π} with gradient descent on loss
$L\left(\theta^{\Pi}\right) = \mathrm{E}_{(s,a)\sim M_{SL}}\left[KL\ Divergence\ \Pi\left(s,a|\theta^{\Pi}\right)\right]$
Update θ^{Q} with gradient descent on loss
$L\left(\theta^{Q}\right) = \mathrm{E}_{(s,a,r,s')\sim M_{RL}}\left[\left(r + \max_{a'}\ Q\left(s',a'|\theta^{Q'}\right) - Q\left(s,a|\theta^{Q}\right)\right)^2\right]$
Periodically update target network parameters $\theta^{Q'} \leftarrow \theta^{Q}$
end function

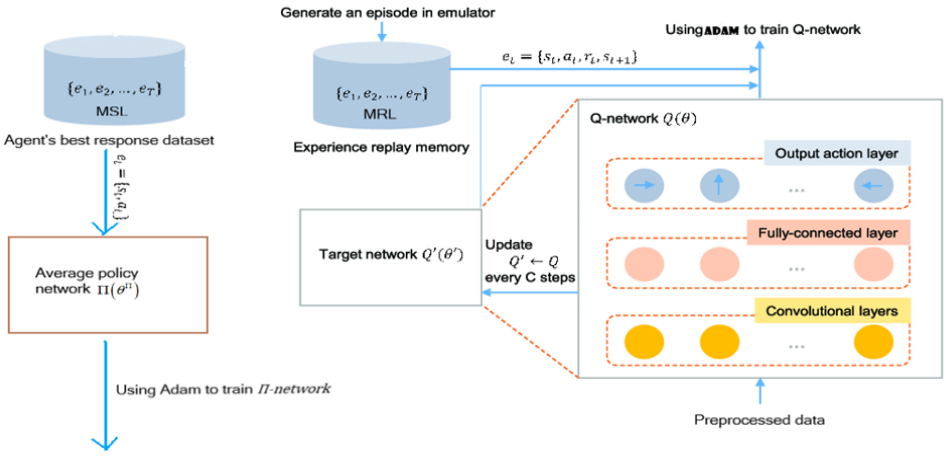


FIGURE 3. Overview of Agent 2 architecture.

The input is represented as a *17x17x9* 3D array containing the images of the last two board states joined by the scalar features we mentioned at **Agent 1** where we add the opponent last action. The fact that we add the last board state and the opponent last action is due to wanting to test an *attention mechanism* similar to the one used for *AlphaGo Zero* [12]. The CNN is composed of 5 hidden layers: 4 layers of convolution, 2 MaxPooling and 1 fully-connected. The loss for the value network remains the classic *MSE* and for the policy network, we use *KL divergence*. M_{SL} will be updated using *reservoir sampling* [16] and M_{RL} will function as a *circular buffer*. Above (figure 3), we can see the architecture of this agent.

4. Experiments

I am mainly focused on no-limit variant of Poker for experiments, but I am also going to test the algorithm on another imperfect information game to solidify our claim of general scalability and applicability. I devised a less complex game than Poker and verify the necessity of the essential components by rigorously evaluating the respective performances. In case of Poker, we are going to measure each agent's performance against some generic players and against each other.

4.1. **Applying on another game.**

Introducing *Blop game* (figure 4), originally a perfect information game that consists of a quadratic matrix/image where a pixel is colored as blue (our player), another as green (the exit) and the third one as red (the enemy). The player can move in all 8 directions associated with the grid, or may choose to stay still. The player receives a negative 1 reward for moving in any direction and a positive 20 for reaching the exit, but it gains negative 300 if it hits the enemy moment when also the game ends. The objective of this game is to reach the destination as efficient as possible.

Without any additional rules, this forms a deterministic game that can be solved quite easily by popular search methods such as A^*. Because we want to observe the functionality of the algorithm in the field of games with imperfect information, we will make a **fundamental change** in terms of the base rules used. Thus, we will ***poison*** 5 out of the 9 basic moves: when the player tries to stand still or move vertically or horizontally, he will make a random move instead. Thus, only the **4 movements** (diagonally) would work as intended.

For this experiment, the initial configuration uses a 10x10 image and is always the same: the player in the upper left corner, the exit in the center and with enemy steady near the exit on the segment formed by the initial points of the player and the exit. This way, the max reward that we are aiming for is **20**, obtained only through diagonal moves.

We use *algorithm 2*, all we feed the algorithm is the **RGB** images of states, we therefore use convolutional neural networks with 3 hidden layers (2 convolutions and 1 fully-connected). The parameter η was set to **0.1**, ε to **0.12**, max size of $M_{SL}$to **2m** and for M_{RL} to **20k**. We updated the parameters of Q and Π networks once every **4** steps (for each one) and the *target network* parameters were reset once every **5** episodes.

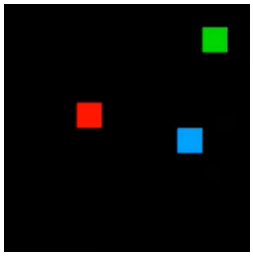
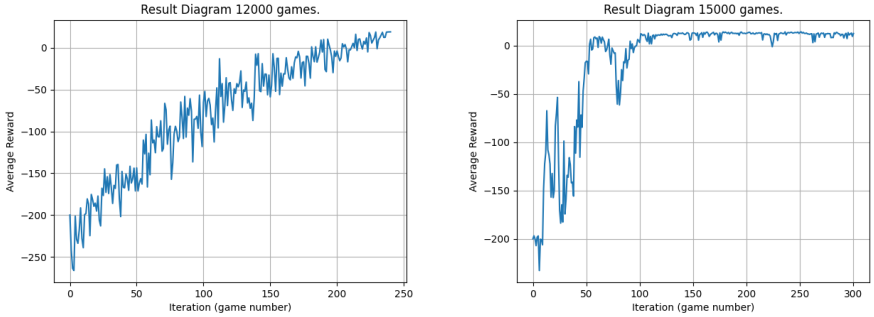


FIGURE 4. Blop Game.

In order to study the performance and the speed of convergence, I compare the results with the results obtained by a standard *Double DQN* [10], a very popular system for solving games. This was implemented through setting the parameter η to **1** (always selecting the greedy strategy).

Figure 5 shows a crushing victory for our implementation that uses a combination of greedy and average strategies.



(a) Standard DDQN training process. Each stat point represents the aggregate performance in last 50 episodes

(b) Algorithm 2 training process. Each stat point represents the aggregate performance in last 50 episodes

FIGURE 5. Comparison between algorithm 2 performance in Blop Game and a standard method of solving games from literature.

It learns much quicker that it should not rely on anything apart from diagonal moves for reaching the goal.

Both greedy and average strategies of Agent 2 converge to the same optimal solution of reaching the exit in 5 moves with only diagonal moves. However, while at episode 5000, the algorithm 2 was pretty much done, the DDQN aggregate reward was still in the negatives 50.

I then modified the game to allow the enemy to randomly move, as the player moves, in order to add even more imperfect information to the game. This did not pose a challenge to our agent though, as it solved the game almost as fast as the previous version with quicker convergence speeds than DDQN's.

4.2. **General specifications for the Poker games.**

The format I am using for the games is heads-up, no-limit with **100** chips as starting stack and **5** chips small blind. For performance evaluations I am using two metrics: ***average stack*** over a fixed number of games and ***mbb/h*** (milli big blinds per hand = 1/1000 of a big blind). To provide some intuition, the values for a mbb/h metric will usually stay in the interval [-750, 750] and a human professional player would aim for winnings of **40-50 mbb/h**. An average stack of over 100 guarantees, most of the time, a match **win rate** of at least 50%.

The generic players used are the following: *Randomplayer* (a player that chooses call 3 times out of 5 and the other actions 2 times out of 5 with equal probable chance), a *Callplayer* (a players that always calls) and *HeuristicMCplayer* (a player that chooses its actions based only on Monte-Carlo simulations and not look-up tables).

4.3. **Agent 1.**

I am going to refer to the *Agent 1* without opponent modelling as BaseAgent1 player. We can clearly see an improvement in the performance of Agent 1 (Table 1), using the opponent modelling part compared to when we don't use it. Although I expected a higher gain in winnings, we should not forget that we are limited by how good and exploitable the expert systems behind Agent 1 are. After 250 games against *HeuristicMC*, we got **85.71%** test set accuracy for predicting moves which provided a **2%** increase in performance during this length of play.

Results after 250 games Texas Hold'em Poker per player-match, statistical error +/- 2			
Player 1	**Player 2**	Avg. Stack (Player 1)	**Result (winrate P1)**
HeuristicMCplayer	**Randomplayer**	107	52%
BaseAgent1player	**Randomplayer**	141	72%
BaseAgent1player	**Callplayer**	163	81.6%
Agent 1	**Callplayer**	167	84.8%
BaseAgent1player	**HeuristicMCplayer**	111	57.6%
Agent 1	**HeuristicMCplayer**	119	59.2%

Table 1 Results of some experiments with artificial players

Experiment with a human player

I've invited a friend, Catalin, to take on this first agent. The test subject has an **advanced beginner** to **low intermediate level** at Poker, he knows

the rules of the game very well and can make educated decisions during most of the situations, but lacks the experience of more advanced players. Catalin accepted to play a total of **29** games against **Agent 1** in which he adopted an anti-computer strategy, constantly changing his style of play and testing for bluffs.

With all of that said, **Agent 1** managed to beat him both in the first 22 games where an *opponent model* wasn't available and in the next 7 games at full power. Even on such a small sample size of games, the neural network signaled a **60%** accuracy in predicting the opponent's next move.

Player 1	Score	Player 2	No games	Win-rate	
Base Agent 1	+13 -9	**Catalin**	22	59.09%	
Agent 1	+6 -1	**Catalin**	7	85.71%	
Agent 1 total	+19 -10	**Catalin**	29	65.51%	Total

Table 2 Agent 1 and BaseAgent1 performance against a human (low intermediate) player

The majority of losses came from *all-ins* in the preflop stage of the game, but as the model learnt more about Catalin's playing style, it became more resilient in calling bluffs and started to aim for a turn-river finish. The mbb/h winnings were over **150 mbb/h** for the artificial player. One other thing that Catalin told us is that he became very surprised of the playing style in the last 7 games, during which the agent tried to exploit him.

4.4. **Agent 2.**

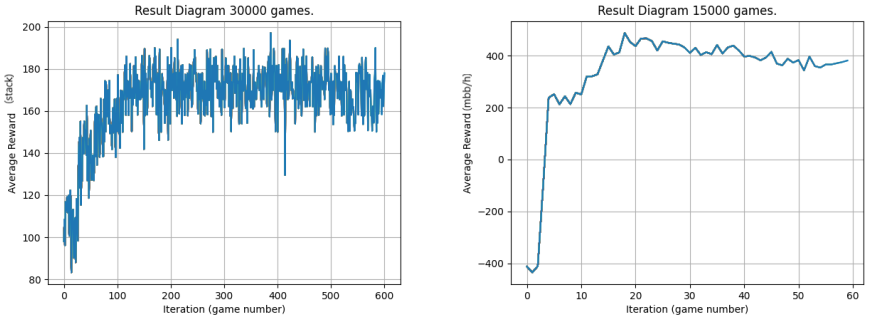
We are ready to apply the algorithm in no-limit Texas Hold'em Poker, by considering the opponent as part of the environment and trying to consistently beat him. η and ε were both set to 0.1, max length of M_{RL} to 300k and for M_{SL} to 1.2 m, the *learning rate* fir reinforcement learning and supervised learning were set to 0.05, 0.005, respectively. The exploration rate ε decays to 0 proportionally to the inverse square root of the number of games in the training process. The agent performs 2 stochastic gradient updates of mini-batch size 256 per network for every game. The *target network-ul* parameters were reset once every 128 hands of play.

The training process was performed several times from scratch to confirm that the results are indeed consistent. The ultimate goal of this agent is to beat Agent 1 in at least 250 games match, for this we first trained an agent to beat *Random*_player (to first get a small sense of how Poker works) and then, we saved that version to next train with BaseAgent1_player.

Finally, this version of Agent 2 will be the starting point to train against Agent 1.

Below *(figure 6)* we can see the performance of algorithm 2 training against the Random_player, it quickly crushes him. In the testing phase afterwards, the

average strategy of this agent recorded an average stack of **173.02** (84.8%-win rate), while the combined strategies approach recorded a close **166.23**.



(a) Agent 2 training process. Each stat point = the aggregate performance (stack) in last 50 episodes

(b) Agent 2 training process. Each stat point = the aggregate performance (mbb/h) in last 250

FIGURE 6. Measuring Agent 2 training performance in stack and mbb metrics.

The greedy strategy (Q-network strength) is a little lower at **137.82**. It is nice to see that already **Agent 2** became more successful in defeating *Randomplayer* than **Agent 1** ever was. Below (figure 7), we can clearly see how the agent won, by analyzing his play style.

It seems that in general, the agent is aggressive, always trying to increase the pot and earn more. This is indeed the **right strategy** against a player who does not rely on any relevant game information. But obviously, *call* or *fold* decisions must be made at least occasionally, when the game hand / current situation of the board is unlucky for us (in order to stop the opponent from winning through luck).

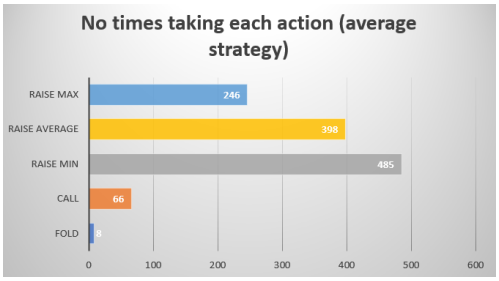
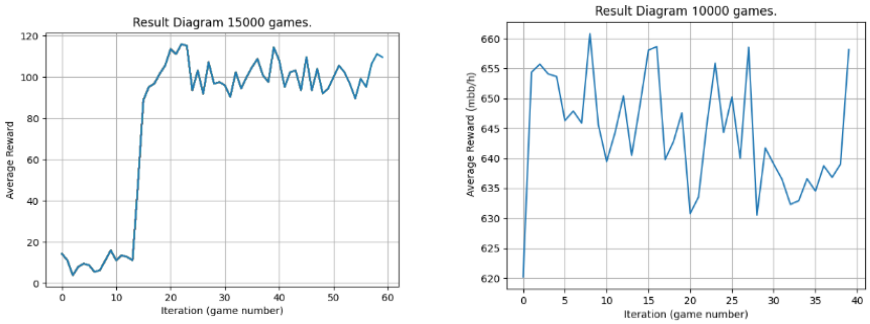


FIGURE 7. Play style in 250 games vs Randomplayer

Going after Agent 1 now *(figure 8)*, we can see that in the first 15k games, the performance is pretty much similar to BaseAgent1player, but after another 15k games of training Agent 2 completely outshines him. He wins, apparently by finding a way to exploit the expert systems that both BaseAgent1 and Agent 1 are based on. This effect

seems more severe when training against Agent 1 where the winnings cross over 600 mbb/h. We can deduct that from this agent playing style, using only raise average, calls and folds. On the other hand, the version of the Agent 2 that won against BaseAgent1 has a more balanced playing style and it is more destined to do well against human players. The strategies do not converge to the same locally optimal one, which means there is still space for improvement by increasing the number of iterations.



(a) Training vs BaseAgent1, first 15k games

(b) Training vs Agent 1 after 10k games

FIGURE 8. Training evolution against the two versions of Agent 1.

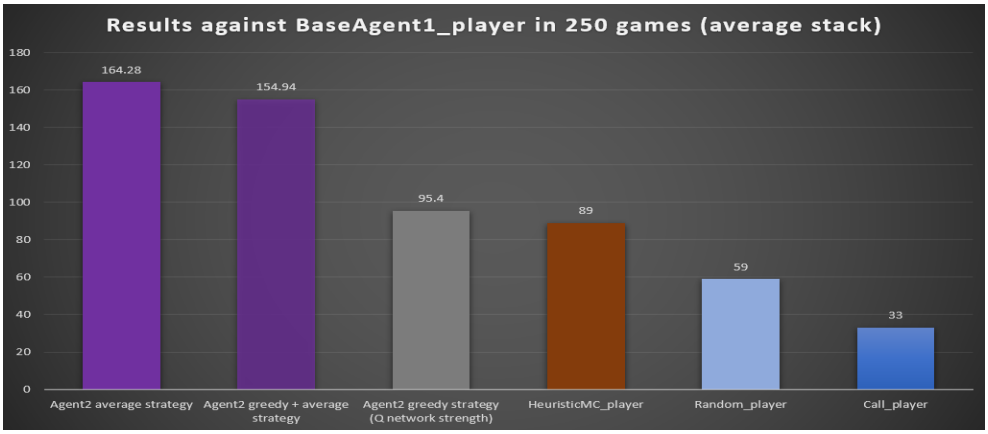


FIGURE 9. Results of some previous players against BaseAgent1player compared with Agent 2; statistical error +/-10.

However, due to the nature of the study and limited resources, the current results are good enough to call a victory for reinforcement learning.

It is interesting to visualize the overall performance of all the agents against one of the best build until now *(figure 9)*. The version used in that chart was the one trained for 30k games against Agent 1.

As we can see, the one agent developed through reinforcement learning completely outperforms the other ones in head-up no-limit Texas Hold'em Poker.

Experiment with a human player

I've invited another friend who has a much higher level than Catalin at Poker. The test subject has an **amateur** level of play, advanced intermediate to advanced. He is experienced, but lacks the real money high stakes experience of play that professional players possess. Expectations are not high, because in the end, our artificial player knows the game of poker only by training with other artificial players. Do note that I am using the version of Agent 2 whose performance can be visualized in figure 9.

In **10** games arranged for this match, Agent 2 won **7** and lost **3**, with an estimated winnings of **120** mbb/h. First of all, analyzing the games, Agent 2 really taught himself the basic, trivial strategies of the game:

- never fold if the opponent does not raise after preflop.
- mostly raise a good hand but also bluff from time to time if you have good potential for the next streets.
- usually all-in when the hand is very good and the pot is significant.
- never fold in the next round after a big raise of your own, if the opponent does not put pressure.

Secondly, I noticed a very aggressive tendency in *preflop*, which is also present in other artificial players such as **Cepheus** [3], but what is even more interesting, although the *AI* prefers to *raise* in this part of the game, in most cases he does not accept this exact behavior from his opponent, folding to a raise greater than 20-30 in preflop (but not all the time).

Third, it's pretty tricky to accurately report whether the AI really does intentionally bluff or not, but from what I've noticed, even when going all-in on the flop or turn, it always has at least one pair, probably fail-safe. There were, however, a few isolated cases, when he put a lot of pressure but had nothing in his hand, probably estimating that the opponent, most likely, has nothing.

This agent can actually play a **multi-player Poker game**, although not as well as in heads-up, by making a small change in our inputs when we use the

predict function to get a move. The only input components that we use, relevant to a multi-player game, is the average estimated opponent strength, which can be recomputed with respect to the number of players through Monte-Carlo simulations and the opponent's stack which can be substituted with the average stack of all the opponents.

Note that for these experiments, I used a *NVIDIA Tesla T4 Workstation* with *32*GB of RAM and a *NVIDIA GTX 1050ti w*ith *16*GB of RAM, but the resulting artificial players can be run on a less impressive machine even without a GPU, with 8GB of RAM.

5. Conclusion and Further Research

I have successfully showed the power and utility of deep reinforcement learning in imperfect information games, compared to other methods. When applied to no-limit hold'em Poker, deep reinforcement learning agents clearly outperform agents with a more traditional approach.

Further research on this matter may consists in developing a Poker agent trained completely through self-play. It would be interesting to see how an artificial player that learns only by playing with a decent opponent to get better at a game stands against a player trained by playing only against itself.

Acknowledgment

I want to thank Professor Laura Diosan and Lecturer Gabriel Mircea (Babes-Bolyai University) for supervising this project. As an inexperienced undergraduate, I received massive advice from both of them to complete this research project.

References

[1] Arulkumaran, K., Cully, A., and Togelius, J. Alphastar: An evolutionary computation perspective. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion* (2019), pp. 314–315.

[2] Barto, A. G. *Reinforcement learning: An introduction*. MIT press, 1998.

[3] Bowling, M., Burch, N., Johanson, M., and Tammelin, O. Heads-up limit hold'em poker is solved. *Science 347*, 6218 (2015), 145–149.

[4] Brown, G. W. Iterative solution of games by fictitious play. *Activity analysis of production and allocation 13*, 1 (1951), 374–376.

[5] Heinrich, J., Lanctot, M., and Silver, D. Fictitious self-play in extensive-form games. In *International Conference on Machine Learning* (2015), pp. 805–813.

[6] Heinrich, J., and Silver, D. Deep reinforcement learning from self-play in imperfect-information games. *arXiv preprint arXiv:1603.01121* (2016).

[7] Lambert Iii, T. J., Epelman, M. A., and Smith, R. L. A fictitious play approach to large-scale optimization. *Operations Research 53*, 3 (2005), 477–489.

[8] Nevmyvaka, Y., Feng, Y., and Kearns, M. Reinforcement learning for optimized trade execution. In *Proceedings of the 23rd international conference on Machine learning* (2006), pp. 673–680.
[9] Papp, D. R. Dealing with imperfect information in poker.
[10] Sewak, M. Deep q network (dqn), double dqn, and dueling dqn. In *Deep Reinforcement Learning*. Springer, 2019, pp. 95–108.
[11] Shamma, J. S., and Arslan, G. Dynamic fictitious play, dynamic gradient play, and distributed convergence to nash equilibria. *IEEE Transactions on Automatic Control 50*, 3 (2005), 312–327.
[12] Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., et al. Mastering the game of go without human knowledge. *nature 550*, 7676 (2017), 354–359.
[13] Stipić, A., Bronzin, T., Prole, B., and Pap, K. Deep learning advancements: closing the gap. In *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)* (2019), IEEE, pp. 1087–1092.
[14] Urieli, D., and Stone, P. Tactex'13: a champion adaptive power trading agent. In *Twenty-Eighth AAAI Conference on Artificial Intelligence* (2014).
[15] Van Hasselt, H., Guez, A., and Silver, D. Deep reinforcement learning with double q-learning. In *Thirtieth AAAI conference on artificial intelligence* (2016).
[16] Vitter, J. S. Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS) 11*, 1 (1985), 37–57.
[17] Watkins, C. J., and Dayan, P. Q-learning. *Machine learning 8*, 3-4 (1992), 279–292.
[18] Zhang, L., Wang, W., Li, S., and Pan, G. Monte carlo neural fictitious self-play: Approach to approximate nash equilibrium of imperfect-information games. *arXiv preprint arXiv:1903.09569* (2019).

Babeș-Bolyai University, Department of Computer Science, 1 M. Kogalniceanu Street, 400084 Cluj-Napoca, Romania
Email address: ptidor1@gmail.com