Deep Reinforcement Learning in Imperfect Information Games:
No-limit Texas Hold’em Poker

Pricipe Tidor-Vlad
Student
Babes-Bolyai University
Cluj-Napoca, Romania
ptidorl@gmail.com

Abstract

Many real-world applications can be described as large-scale games of imperfect information. This particular set of problems is challenging due to the random factor that makes even adaptive methods fail to correctly model the problem and find the best solution. Neural Fictitious Self Play (NFSP) is a powerful algorithm for learning approximate Nash equilibrium of imperfect-information games from self-play and it showed real success in limit Texas Hold’em Poker. However, it uses only crude data as input and its core aspect is represented by a Deep Q-Network which is offline and is very hard or impossible to converge in more complex online games with large search scale and deep search depth. In this paper, I develop a new variant of NFSP that combines the established fictitious self-play with neural gradient play to improve the performance on large-scale zero-sum imperfect-information games. I use this algorithm to solve the more complex no-limit version of Texas Hold’em Poker using powerful handcrafted metrics and heuristics alongside crude, raw data. When applied to no-limit hold’em Poker, the agents trained through self-play outperformed the ones that used fictitious play with a normal-form single-step approach to the game. Moreover, I showed that my algorithm converges close to a Nash equilibrium within the limited training process of my agents with very limited hardware. Finally, my best self-play-based agent learnt a strategy that rivals expert human level.

Keywords—AI, Reinforcement Learning, Poker, self-play, gradient play.

I. 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 [1]. 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 ([12], [3], [4]). Most of these real-world games involve decision making with imperfect information and high-dimensional information state spaces.

We witnessed the rapid development of computer AI with super-human performance in perfect-information games like Chess and Go (AlphaGo Zero, [5]; LeelaChessZero [6]), but researchers have yet to reach the same progress in imperfect-information games (AlphaStar, [7]). An optimal theoretical solution to these games would be a Nash equilibrium i.e. a strategy no one can gain extra profit by deviating from it.

Fictitious play [8] is a popular method for achieving Nash Equilibria in normal-form (single-step) games. Fictitious Self-Play (FSP) [9] extends this method to extensive-form (multi-step) games. Neural fictitious Self-Play (NFSP, [10]) combines FSP with neural network function approximation. It is an effective algorithm and the first end-to-end reinforcement learning system that learns approximate Nash Equilibrium in imperfect information games without prior knowledge. It uses anticipatory dynamics; the agents choose their strategies from a mixture of average (supervised learning network) and greedy responses (Q-learning network).

With all of that said, it was proven that NFSP provides poor performance in games with large-scale search space and search depth [11], because of the complexity of opponents’ strategy and the fact that a DQN learns in offline mode, not real-time during the game. Solutions to these problems were proposed (MC-NFSP, [11]) that use Monte Carlo Tree Search instead. This, indeed, provides better and more stable performance but I am interested in a pure neural approach not using any brute force search methods. As I am going to apply this algorithm mainly to Poker, a game where intuition is key in winning, exhaustive search is not always necessary. In this paper, I solve this issue by combining anticipatory dynamics with neural gradient play which yields an incremental better response search for our strategies.

Many AI bots have proven themselves to be above any human in no-limit Hold’em (Libratus [12], Pluribus [13]) but this does not mean that the game is completely solved. For that, we need a mathematical way of showing that the agent will definitely win money, given a certain interval of time or games, which was actually done with Cepheus [14] for the in-limit version. The main agent I develop is an attempt to mathematically solve the no-limit variant of Texas Hold’em which is still considered unsolved in different formats to this day.

Furthermore, this paper also highlights a direct comparison to some of my previously developed agents. For this, I refer to my article A View on Deep Reinforcement Learning in Imperfect Information Games [15].

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. As input, I use raw data, as an image of cards from the current visible board combined with two hand-crafted scalar inputs: hard coded rankings of card
combinations and Monte-Carlo heuristics for assessing an approximate strength of the opponent hand. The best agent built (with my modest hardware) learnt a strategy close to human expert play.

II. BACKGROUND

In this section I provide an overview of reinforcement learning and fictitious self-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.

A. Reinforcement learning

First, Reinforcement learning [1] agents typically learn to maximize their expected future rewards from interaction with an environment that 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 \((r)\) \(s,a,r,s'\) . 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 \(\pi: Q(s,a) = E[G_t | S_t = s, A_t = a] \). Here, \(G_t = \sum_{i=0}^{T} R_{s,a,i}\) is a random variable of the agent’s cumulative future rewards starting from time \(t\) [1]. From this, it easily follows that we may want to take the action of the highest estimated value \(Q\), that’s why \(Q\)-learning [21] 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 [16].

B. Neural Fictitious Self-Play

Neural Fictitious Self-Play [10] is a model of learning approximate Nash Equilibrium in imperfect-information games using deep learning. At each iteration, the agents choose their best response (greedy strategy) with a DQN and update their average strategy by supervised learning through a policy network. That is done by storing datasets of each agent’s experience in self-play as transition tuples \((s,a,r,s')\) in a memory \(M_{sl}\) (designed for RL) and by storing agent’s own behavior \((s,a)\) in a memory \(M_{ul}\) (designed for supervised learning). If we set the self-play sampling in a way that an agent’s reinforcement learning memory approximates data of an MDP defined by the other players’ average strategy profile, then we can be sure that we find an approximate best response from an approximate solution of the MDP by reinforcement learning. As we can see, the respective data necessary to train the neural networks through backpropagation is collected within the simulated games during the training process which is offline so it naturally has problems in on-policy games where we need to sample opponents’ changing strategy while we play. To see how we can improve on this and take more into consideration the opponents’ ever-changing strategies, we need to look deeper at how NFSP uses anticipatory dynamics [17] to stabilize the convergence around Nash Equilibrium points.

Define \(\Delta(n)\) as a simplex in \(R^n\), \(v_i \in \Delta(n)\) being the \(i\)-th vertex and let \(H: \text{int}(\Delta(n)) \rightarrow R\) the entropy function \(H(s) = - s^T \log(s)\). In a two-player game, each player chooses its strategy \(p_i \in \Delta(m_i)\), \(m_i \in N^\prime\) and collects the associate reward given by the value-function: \(V(s_i | p_i, p_\text{opp}) = p_i^T M(p_i, p_\text{opp}) + \tau \cdot H(p_i)\), where \(i \in \{1,2,..,n\}\) refers to the complementary set \(\{1,2,..,i-1,i+1,...,n\}\) [17].

Note that we shall use reinforcement learning to approximate this value-function. It follows that we can define player \(i\)’s best response as a function \(\beta_i: \Delta(m_i) \rightarrow \Delta(m_i)\), \(\beta_i(p_i, p_\text{opp}) = \text{arg max } V(s_i | p_i, p_\text{opp})\) and player \(i\)’s average response until step \(k\) in the game as empirical frequencies \(\pi_i(k)\) \(N \rightarrow \Delta(m_i)\) of player \(P_i\) [17].

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:

\[
p_i(k) = \beta_i(\pi_\text{opp}(k))
\]

In continuous time FP, the following equations are used:

\[
\frac{d}{dt} \pi_i = \beta_i(\pi_\text{opp}(t)) - \pi_i(t), i = 1,2
\]

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

\[
p_i(t) = \beta_i(\pi_\text{opp}(t) + \eta \frac{d}{dt} \pi_\text{opp}(t)), \eta \text{ positive parameter}
\]

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 [17] show that, depending on the game, for a good choice of \(\eta\), 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.

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

\[
\frac{d}{dt} \pi_i = \beta_i(\pi_i(t) + \eta \frac{d}{dt} \pi_i(t)) - \pi_i(t).
\]
In NSFP [10], the authors chose a discrete time approximation of the derivative: \( \beta^{t+1} - \pi_t \approx \frac{d}{dt} \pi'_t \) which, if substituted in (4) yields:

\[
p_r(t) \approx \beta_t \left( \pi_{t-1} + \eta \beta_t \left( \pi_{t-1} + \pi_t \right) \right)
\]

This actually represents the action-value function with the average strategy \( \pi \) obtained though supervised classification.

Therefore, it uses 3 neural networks. First, a DDQN system [19] with a value network \( Q(s,a|\theta^D) \) for predicting the \( Q \) values for each action based on data from \( M_{SL} \). It trains through backpropagation using the Bellman equation with future \( Q \) values obtained through a target network \( Q'(s,a|\theta^T) \). Secondly, I use a policy network \( \Pi(s,a|\theta^P) \) to define our agent’s average response based on data from \( M_{SL} \). We choose our main policy \( \sigma \) from a mixture of strategies: \( \beta = \epsilon - \text{greedy}(Q) \) and \( \pi = \Pi \) to define our agent’s average response based on data from \( M_{SL} \). This actually represents the same approximation of anticipatory dynamics in discrete time fictitious play used in NFSP [10], but here I am using it to define my agent in a one-player game. I am not trying to approximate a Nash Equilibrium in this context. The other differences come from the model architectures, inputs and from how often I use each strategy of play to sample games. Moreover, unlike NFSP, I mainly considered a Poker game iteration to be just a hand of play here and reset the main policy accordingly.

The neural networks for the two strategies were also implemented as CNNs and have mainly the same architecture, the only difference appearing at the last layer. The input is represented as a 17x17x9 3D array containing the images of the last two board states joined by the scalar features I mentioned at Agent 1 where I add the opponent last action.

C. Agent 3 (my proposed approach now)

To clarify, this agent will be based on self-play only, using neural nets, without any external help from other players for training and without brute force, real-time exhaustive search. This agent will learn by playing with itself, from scratch, both constantly trying to achieve better rewards. Below (figure 1), we can see the architecture of this self-play system and how the strategies are generated.

**Figure 1 Agent 3, self-play system architecture**
Like the Agent 2, I am devising the greedy and average strategies, this time through self-play, though I also have a reference to the opponent’s average strategy to construct a better response search. To understand how this is mathematically done, take the gradient of the value function:

$$V \left( \frac{\partial}{\partial \tau} \right) = M \left. \right|_{\tau \rightarrow i} \left( \frac{\partial}{\partial \tau} \right)$$

We are interested in the differential equations system that defines the dynamic gradient play:

$$\frac{d y}{dt} = P_{\Delta} \left[ \right]$$

where $P_{\Delta} : R^2 \rightarrow \Delta(n)$ is the projection on the simplex $\Delta(n)$. Therefore, we can obtain a parametrized approximation of $\frac{d y}{dt}$, using two forms of behavioral evolution of strategy of play in FP (DT – discrete time FP, GP – gradient play). Using the definition, we get:

$$\frac{d y}{dt} = P_{\Delta} \left[ \right]$$

Let $S(t) \in \Delta(n)$ such that $S(t) = P_{\Delta} \left( \pi_i (t) + M \pi_r (t) \right)$. Then it follows that:

$$
\frac{d y}{dt} = P_{\Delta} \left( \pi_i (t) + M \pi_r (t) \right) = S(t),
$$

combining this with (5) yields that for every $\rho \in [0,1]$ we have:

$$
\frac{d y}{dt} \approx \rho \left( \hat{\beta}_i \hat{\pi}_i (t) + (1 - \rho) \left( \pi_i (t) \right) \right),
$$

Substituting now $\frac{d y}{dt}$ in (3), we get the final formula:

$$p_i (t) = \beta \left( \left( \pi_i (t \right) + \eta \left( \rho \hat{\beta}_i \hat{\pi}_i (t) + (1 - \rho) \left( \pi_i (t) \right) \right),
$$

which means my agent can choose their actions from a mixture of strategies:

$$\sigma = \left( \frac{\hat{\beta}_i \hat{\pi}_i (t) + (1 - \rho) \left( \pi_i (t) \right)}{\rho \hat{\beta}_i \hat{\pi}_i (t) + \left( \pi_i (t) \right) \right)}.$$

The motivation behind this choice is that the evolution of the GP strategy follows a better response search, adjusting the strategy of play in the direction of the gradient from the empirical frequencies of the opponent. Thus, using this form, especially in a game with imperfect information, where the best answer is harder to find, it is important that we don’t stagnate and we always try to find a better solution than the current one (and if we have already found the best solution then the gradient should suggest so).

I want to favor finding the best response though, that is why are going to set the $\rho$ parameter to be:

$$\rho \approx 1 - \eta + \epsilon \quad \text{with} \quad 0 < \epsilon < 2 / 100.$$

Below, I present Algorithm 1, the main algorithm that agent 3 uses to get learn Poker from self-play.

**Algorithm 1 | Agent 3, reinforcement learning (self-play) agent with fitted Q-learning**

```plaintext
for 1:no_games do
    Initialize new game G and execute agent via RUN_AGENT for each player in the game
    function RUN_AGENT(G)
        Initialize memory $M_{\epsilon} \left( \right)$ and $M_{\eta} \left( \right)$ own behaviour reservoir
        Initialize average-policy network $\Pi \left( s, a \left| \theta^\Pi \right. \right)$ with random weights $\theta^\Pi$
        Initialize opponent average-policy network $\Pi \left( s, a \left| \theta^\Pi \right. \right)$ with random weights $\theta^\Pi$
        Initialize action-value network $Q \left( s, a \left| \theta^Q \right. \right)$ with random weights $\theta^Q$
        Initialize target network with weights $\theta^Q \left. \right| \theta^Q$
        Initialize anticipatory parameters $\eta, \rho$
        Observe initial information state $s_i$ and reward $r_i$
        for $t=1$ to $\min$ replay memory, size do
            Sample action $a_i$ from policy $\sigma$
            Execute action $a_i$ in emulator and observe reward $r_i$ and next information state $s'_i$
            Store transition $(s_i, a_i, r_i, s_i')$ in reinforcement learning memory $M_\epsilon$
            if agent follows best response policy $\sigma = \beta \left( \epsilon - \text{greedy} \left( Q \right) \right)$ then
                Store behaviour tuple $(s_i, a_i)$ in supervised learning memory $M_\eta$
            end if
            Update $\theta^Q$ with gradient descent on loss
            $L \left( \theta^Q \right) = E_{s_i, a_i} \left( KL \text{ Divergence} \Pi \left( s, a \left| \theta^\Pi \right. \right) \right)$
            Update $\theta^\Pi$ with gradient descent on loss
            $L \left( \theta^\Pi \right) = E_{r_i, \epsilon, \eta, s_i} \left( \left[ r \right. \max_e \left( Q \left( s, a \left| \theta^Q \right. \right) \right) \right)$
            Periodically update target network parameters $\theta^Q \left. \right| \theta^Q$
        end for
    end for
end function
```

**IV. EXPERIMENTS**

I am mainly focused on no-limit variant of Poker for experiments, but I also tested a similar algorithm on another imperfect information game to solidify my claim of general scalability and applicability in the other paper I referenced previously [15]. In case of Poker, I am going to measure each agent’s performance against some generic players and against each other. I also paired the final agent against a human player.

**A. General specifications**

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 $\text{mbb/h}$ (milli big blinds per hand = 1/1000 of a big blind). To provide some intuition, the values for a $\text{mbb/h}$ metric will usually stay in the interval $[-750, 750]$ and a human professional player would aim for winnings of $40-50 \text{ mbb/h}$. 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: Random_player (a player that chooses call 3 times out of 5 and the other actions 2 times out of 5 with equal probable chance), a Call_player (a players that always calls) and Heruristic_MC_player (a player that chooses its actions based only on Monte-Carlo simulations and not look-up tables).
B. No-limit Texas Hold’em Poker

We want our self-play agent to be unbeatable in the long run, so now an episode will be represented by a game (which can have several hands) and not an only hand of play as I considered in Agent 2. Also, Agent 3 will receive an immediate reward of 0 for each move and only at the end of a hand / end of a game, he will receive a non-zero reward depending on how many chips he won. Thus, Agent 3 will not be penalized immediately for a raise of 100 (all-in), for example, but if he loses that hand, then he will receive a reward of negative 100 at the end of it, which is very high. In this way, we tell the AI that it doesn’t matter what moves he chooses as long as the reward at the end of the game is maximum.

Since we want to test the effect of that better response search through gradient play proposed in the theoretical part, we will analyze the behavior / performance of an Agent 3 trained against a copy of itself taking into account the policy \( \hat{\pi} \), \( (\rho < 1) \) and the behavior performance of an Agent 3 trained against a copy of itself without regard to the policy \( \hat{\pi} \), \( (\rho = 1) \), as of Algorithm 3. I will therefore call these two agents: Agent3_GP, Agent3_DP, from gradient play, discrete play respectively (which refers to the method used to approximate the CDP derivative). The final Agent 3 will be a version of Agent3_GP.

As a first remark, Agent3_GP needs almost half the time (6h) to finish the training of 50k games, compared to Agent3_DP (11h). We can observe a much smoother decrease to 0 and less noisy in the case of Agent3_DP, but at the same time, the order of magnitude is much larger in the case of Agent3_GP. In the end, the mbb/h value stops at 12.55 for Agent3_DP and for Agent3_GP it hits 4.07. In practice, however, we will see that the performance of the agent using better response search through gradient play is better than the other one.

What is very impressive here (figure 5) is the fact that I used the version of Agent 2 that trained with Random_player, having as sole objective to defeat it. Although Agent 3 had no interaction with Random_player, learning the game of poker only through self-play, he achieves a performance almost identical to that of Agent 2, even surpassing the performance of all the other deep reinforcement learning agents, after just 17 hours of training!

This match-up was also an opportunity to study the differences between Agent3_GP’s style of play and Agent3_DP’s. Agent3_DP plays much safer and is much more reserved about a raise, mainly choosing to wait through calls, very rarely choosing to go all-in (figure 2). Instead, Agent3_GP is much more aggressive, bouncing back between calls (predominant action) and raises.

I decided to first test the performance when we train less than 1 day. In the figures 3 and 4 we can see Agent 3’ the descendancy to Nash Equilibrium. Parameters \( \eta \) and \( \varepsilon \) were set to 0.1, 0.9, respectively, \( \rho \) was set to 0.92 (for Agent3_GP), max length for \( M_{56} \) to 200k and for \( M_{52} \) at 1m. We make one stochastic gradient update of mini-batch size of 256 per network for every 64 steps and the target network parameters were reset every 1000 updates.

<table>
<thead>
<tr>
<th>Player</th>
<th>No. hours trained</th>
<th>No. Games Played</th>
<th>Final Average Stack</th>
<th>Winnings (mbb/h)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agent3_GP</td>
<td>6</td>
<td>250</td>
<td>117.83</td>
<td>263.37</td>
</tr>
<tr>
<td>Agent3_DP</td>
<td>11</td>
<td>250</td>
<td>120.71</td>
<td>318.93</td>
</tr>
<tr>
<td>Agent3_GP</td>
<td>45</td>
<td>250</td>
<td>99.2</td>
<td>192.55</td>
</tr>
<tr>
<td>Agent3_GP</td>
<td>17</td>
<td>250</td>
<td>117.1</td>
<td>338.82</td>
</tr>
<tr>
<td>Agent3_GP</td>
<td>30</td>
<td>250</td>
<td>110</td>
<td>340.18</td>
</tr>
<tr>
<td>Agent3_GP</td>
<td>6</td>
<td>1000</td>
<td>118.45</td>
<td>248.35</td>
</tr>
<tr>
<td>Agent3_GP</td>
<td>11</td>
<td>1000</td>
<td>116.42</td>
<td>299.03</td>
</tr>
<tr>
<td>Agent3_GP</td>
<td>30</td>
<td>1000</td>
<td>111.9</td>
<td>361.08</td>
</tr>
<tr>
<td>Agent3_GP</td>
<td>17</td>
<td>1000</td>
<td>116.54</td>
<td>356.17</td>
</tr>
</tbody>
</table>
Agent3_GP consistently beats Agent 2, in both experiments (250 and 1000 games, respectively), this makes Agent 3 take the status of the best agent developed so far, after only 6 hours of self-play! Out of curiosity, I paired up Agent3_GP after 30 hours of training against Agent 1. The results are not surprising at all, getting a win rate of 88.46% and an average stack of 175 after 130 games against the expert system with neuronal opponent modeling.

To obtain the final version of Agent 3, I let it train through self-play 3 days straight. After it was over, it became the best agent I developed. The training evolution can be observed in figure 6.

All of my agents can actually play a multi-player Poker game, although not as well as in heads-up, by making a small change in the inputs when I use the predict function to get a move. The only input components that I 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.

I also trained the algorithm with raw data, without hand-crafted input metrics, just like in NSFP [10], to see if the algorithm still converges without any prior knowledge of the domain. And if so, how does this compare to the version above in which we are actually using solid prior knowledge of the game.

The results can be seen in figure 7. It seems the algorithm still converges to approximate Nash-Equilibrium, but slower than our main proposed version.

For my final experiment, I’ve invited a semi-professional human Poker player, Serban. He is very experienced with the game, playing constantly on real high money stakes but lacks the tournament play.
He played 56 hands against my agent, from figure 6, (during a 10-game match) and the results were crushing. My agent recorded winnings of 241.07 mbb/h with the final score 7-3.

The human player said it was very impressed with the style of play of my agent but he recognized some mistakes during the match regarding the preflop stage of the game, which can be very costly during a professional match. Mainly, the agent does not recognize very weak cards in the preflop, such as 7-3, at which point he should not call for a raise.

A temporary solution could be a Monte-Carlo search, which immediately draws attention to very weak combinations of cards at any stage of the game. Indeed, this version is still not perfect, or close to perfect, but training on more iterations should strengthen my AI bot considerably. It is an important victory, though, all things considered.

We can sum up all the information in the graph below (figure 8), which illustrates the full evolution of my final agent.

![Graph](image)

**Figure 8 Agent 3 training and performance evolution**

V. FURTHER RESEARCH

Although the results looked pretty successful, it is very hard to correctly assess the level of play of my best agents. Until I test them against a professional player or top computer programs like *Hyperborean*, I can’t know for sure that they are indeed at top human level. Furthermore, due to time and hardware constrains, I couldn’t experiment on more iterations, we can maybe descend even more closer to a Nash equilibrium in optimal conditions. Improvements can also be made regarding the format of the game. All the agents were trained in heads-up, no-limit, 100-100 starting stack with 5 small blind formats, but for more general play, it is recommended to consider the small blind as percentage of the starting stack.

VI. CONCLUSION

I have successfully showed the power and utility of deep reinforcement learning in imperfect information games, compared to other methods and I have developed a new approach to learning approximate Nash equilibria from self-play that does not use any brute force search and only relies on the intuition provided by deep neural networks. When applied to no-limit hold’em Poker, training through self-play drastically increased the performance compared to fictitious play training with a normal-form singe-step approach to the game. My experiments have shown the self-play agent to converge reliably to approximate Nash equilibria with crude data and limited hand-crafted metrics as input. I have also proved that my algorithm is very flexible and comes with a great measure of scalability, being an option for other imperfect information games with any number of players.

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