I would like to thank all my professors, in particular my thesis supervisor Nenad Kos for the support during the preparation of this paper

my family, and especially my grandmother Tina, who never stopped believing in me

and my friends, with whom I shared highs and lows of the wonderful experience of this Bachelor's

Regret minimization and reinforcement learning to find equilibria in incomplete information games

Roberto Ceraolo
2021

Abstract

Several real-world situations can be modeled through Game Theory as large-scale, multi-agent games of imperfect information. The game of Poker Texas Hold'Em has been on the rise as a game-theoretic modeling challenge, because of its huge game size and the attractiveness of the game per se. In this paper, starting from basic concepts in Game Theory and Machine Learning, we analyze in detail the most common methods used to build bots for Poker games. In such settings, Nash equilibria can be approximated effectively through Counterfactual Regret Minimization, which is an iterative procedure that learns from self-play. It belongs to the sphere of Reinforcement Learning, as it learns repetitively playing against a copy of itself and looks backward after many iterations in order to choose a strategy. In some cases, it is also combined with the use of Neural Networks and other techniques of Machine Learning. We also address the issue of whether Poker can be considered a "solved" game and go through the functioning of some of the best performing implementations.

Table of contents

1. Introduction

“Real life consists of bluffing, of little tactics of deception, of asking yourself what is the other man going to think I mean to do. And that is what games are about in my theory” - John Von Neumann

Game theory is commonly defined as the field that studies the mathematical modeling of rational interacting agents. It has countless applications in several different domains, ranging from social science, logic, computer science, to economics and business competition. Paradoxically, the most studied applications have not much to do with “Games" in the common sense of the word. Nevertheless, many board games and games which require strategic thinking (e.g., Chess or Poker), can be modeled mathematically following the *laws* of Game Theory.

In the last decades, there has been a steep increase in interest in building Artificial Intelligence systems able to play games and win against humans. In particular, the goal of many researchers has been to “solve” games. Commonly speaking, a game is said to be *solved* if its outcome - win, lose, or draw - can be predicted from any position, assuming both players play their optimal strategy. *Perfect information* games, that are sequential games in which every player is always perfectly informed about previous events - including the initialization, for instance the starting cards in hands - are usually easier to solve. In other words, every player at any time in the game has all the information concerning the game state and its possible developments. Common examples are Chess, Checkers and Go. When considering Chess for instance, practically the only 3 pieces of information a player needs in order to completely identify a game state are: the configurations of pieces on the chessboard, the moves played since the start of the game and the rules of the game. On the other hand, Poker is an *imperfect information* game. Each player sees only his/her own cards and the cards that are dealt on the table, he/she does not know her opponents’ ones. In perfect information games optimal strategies are *pure strategies*, hence for each game state there is a single “right” move (the move that leads to the game theoretic value, as we will see). Instead, in imperfect information games, optimal play often comprises mixed

strategies, so a player chooses move *a* with a certain probability p_a (and at least two moves have positive probability) (Von Neumann and Morgenstern, 1944).
When building intelligent systems that learn to play board games, Game Theory is often exploited. Indeed, being able to model board games as rational multi-agent interactions, AIs are programmed to look for Nash Equilibria. The latter is a key concept in Game Theory and is defined as a set of strategies in which no player has incentive to deviate. In other words, none of them gains more utility (e.g., money) by switching to another strategy.

When building AIs aimed at solving games, we often use reinforcement learning. Reinforcement learning (RL) is a very *hot* field these days, given the huge advancements and the increase of its applications. RL is an area of Machine Learning that focuses on the study and the creation of intelligent agents that take actions in a certain environment with the aim of maximizing their cumulative reward. The intuition is the following: a computer player is trained, either playing firstly against humans and then against copies of itself, or directly self-playing from the beginning. It is *rewarded* or *punished* depending on the quality of its moves. At each iteration it learns something more about how to maximize its reward. Alternatively, it can also be "shown" many played games, and it can learn by "looking", that is, finding the game patterns that lead to victory. Recently, several AIs playing Go (Deepmind, 2016), Hide and Seek (OpenAI, 2020), Chess (Deepmind, 2017), and Poker as well have been developed.

In this paper we look at the famous game of Poker and we study the state-of-the-art bots and artificial intelligence systems that play it. Several methods now exist to find approximate Nash Equilibria in some versions of the game. Most of them use a technique called regret minimization, sometimes together with machine learning algorithms, such as neural networks.
Regret is a common human feeling. It is defined as the emotion that we feel wishing we had made a different choice in the past. In decision theory, it is basically the same thing. The regret of a certain action *a* is the difference between the utility that the decision maker gets from the best possible choice he could have made and the utility

that he/she gets from performing action *a*. Logically, we consider the *best* action the one that minimizes regret. Blackwell (1956) provides us with a way (called regret-matching) to build a policy of actions based on regret that allow us to find Nash equilibria. Unfortunately, in extensive games settings such a method is intractable because of the exponential number of game states. We see that Counterfactual Regret Minimization (chapter 3.3) represents a possible solution in that it tries to minimize regret independently at each information set - which are defined as sets of game states among which the decision maker cannot distinguish - and not altogether.

Being an imperfect information game with several agents, Poker is quite hard to model and to solve. Therefore, every study relies on some degree of abstraction, some simplification of the game. The stronger the abstraction, the further away the simulation is from a real game, and the less the AI will be able to generalize to a real Poker setting. In game-theoretical terms, the number of information sets for each player must be reduced to an acceptable number such that the abstract game can be tackled. Some common poker abstractions include limiting the available sequences of bets, replacing some decisions with fixed policies, reducing the number of rounds or the cards in the deck.

Despite its complexity, in the last years great advancement has been made, from the first computer program beating expert players in 2008 to the point that it was also claimed that Heads-up limit Hold'em Poker is essentially "weakly solved".

The aim of this paper is to analyze the current panorama of computer players able to play poker, their methods of learning, and their link with Game Theory. The reader will be introduced to concepts of Game theory and Machine Learning needed to understand the functioning of such systems. The mathematical framework of regret minimization will be analyzed in detail. The concept of solving a game together with the claim of Poker being weakly solved is also studied. Finally, we will look at some successful implementations of AIs playing poker.

2. Framework - Notation and abstractions

In the past decades, many researchers focused on building intelligent systems able to play popular perfect information games, such as Chess, Checkers and Go. For instance, Connect Four and Checkers were "solved". Basically, before the weak-solution claim of Poker was announced (Bowling, Burch et al., 2015)[1], every nontrivial[2] game solved was a perfect-information game. Imperfect information games are games in which players do not have full knowledge of past events. Examples may be sealed-bid auctions, or also every card game in which players do not know which cards have been dealt to other players. Von Neumann, one of the founders of modern game theory, mentioned and studied poker tactics, as it is the quintessential imperfect information game. Von Neumann's and Borel's are among the first computational studies on multi-agent environments and paved the way for a whole area of research.

2.1. Limit and no limit games

Poker is the most common card game in the world and its most popular variant is Texas Hold'em. Several simplified versions of it are particularly relevant here, as they have been modeled and studied by the researchers working on computational game theory in imperfect information games. Let's keep in mind that their aim is to find approximate optimal strategies for the games. Mainly 3 versions of Poker are approached for these kinds of studies. One of them is a "toy" version, *extremely* simplified, developed in order to study precisely the possible strategies and equilibria. Such version is commonly called "Kuhn" Poker (Kuhn, 1950). The deck includes only three cards, for instance a King, a Queen and a Jack. Each player pays a chip ex-ante, is dealt a card, and then there are two rounds of betting or passing. Two bets or passes in a row end the game and the player with the higher card wins. If a player passes after his/her opponent bets, the former loses. Clearly it is very different from the common game of Poker but some of the essential features of the game are retained. Moreover, the 30

[1] Most of the contributions to the literature on this topic is developed by researchers from the University of Alberta, as they founded the Computer Poker Research Group. They work specifically towards the creation of AIs able to outperform humans in Poker, as a testbed for solving real-world complex problems.

[2] "Trivial" games are the ones that can be solved without the use of a computer

(6 possible card deals times 5 possible developments of the game) possible configurations of the game make it very simple to model.

The other two versions studied by computer scientists and game theorists are much more complex and are commonly played. They are Heads-up Limit Texas Hold'em (HULHE) and Heads-up No Limit Texas Hold'em (HUNL). When considering these variants, we usually take as a reference the modalities played at the Annual Computer Poker Competition (ACPC). This competition fostered and amplified the efforts toward the generation of algorithms capable of finding better and better strategy approximations for Poker. "Heads-up" means that in these variants there are only 2 players in games. "Limit" means that betting is limited to certain values. In limit Texas Hold'em, bets and raises in the first two rounds of betting must necessarily be equal to the big blind[3] and in the following two rounds, bets must be equal to twice the big blind. "No-limit" means instead that players have almost complete freedom in betting strategies. In no-limit Texas Hold'em (which is the one you see on television and in the main tournaments) players can bet any amount over the minimum raise up to the total number of chips the player has (all-in). The minimum raise is usually equal to the size of the previous bet. Clearly this much larger freedom with respect to the limit version implies a larger number of game states, information sets, and larger complexity in modeling. In particular, there are 2 main obstacles: first of all, the set of possible bets at each round depend on the betting sequence taken in earlier rounds. Secondly, the possibility of betting any number of chips in an interval means that there is a huge number of possible betting sequences leading to the final round.

To simplify both HULHE and HUNL, often some so-called "abstractions" are employed. The two main types of abstractions are action abstractions and card abstractions. One example of action abstraction is clustering similar bet sizes in *buckets*. Since the difference between a bet of $300 with one of $302 is not significant at the end of the game, we can restrict the number of allowed bets grouping them and allowing step increases instead of a continuum of bets. Considering every possible bet is not

[3] The big blind is a compulsory bet in some poker games, such as Texas Hold 'Em. The player two seats to the left of the dealer or the dealer button pays for it. Generally, the big blind is the minimum bet allowed in a round.

necessary. Card abstractions instead are about grouping together hands that are strategically similar, such as (A ♣, 4 ♡) and (A ♡, 4 ♣) (Ganzfried, Sandholm, 2014).

2.2. Game states

Having measures for the size of a game helps in understanding its complexity. Generally, the *larger* a game is, the harder it is to find optimal strategies for it. But what does it mean to measure the size of a game? There are many ways in which this can be done, for instance we can consider the number of unique game states, the number of decision points or also the cumulative number of actions that can be chosen in every game state. For poker models, the number of unique game states is commonly used to provide the intuition of the number of possible scenarios in the game. More specifically, counting the game states means finding the number of possible sequences of actions taken by the players or by chance, as viewed by an external spectator. In the case of poker, game states include every possible way in which private and public cards can be dealt, but also all the possible betting sequences. Researchers working on AIs playing poker always try to increase the size of the game in which the AI is trained with respect to previous research, to increase their ability to face complexity and to generalize. In **limit** poker games the number of game states can be calculated from a closed form expression. This is possible since the betting actions available at each round are independent of the previous rounds, hence computing the number of game states is done by multiplying the possible chance events, the number of betting sequences that would bring to that round and the number of information sets of that single round. In HULHE, the size of the game amounts to be 3.162×10^{17} game states and 3.194×10^{14} information sets, even though in practice a slightly smaller version is used (Johanson, 2013). In HUNL, because of the interdependency between the round and the betting freedom, *enumerating* the possible game states is not feasible. More complex methods have been used to try to understand the number of game states in this variant in specific examples from ACPC competitions, but they fall outside the scope of this introduction about game states. Anyway, to provide a rough idea of the difference between the two versions, the

number of decision points in HUNL were estimated to be around 10^{161} (Johanson, 2013).

2.3. Extensive form games

Both sequential (i.e., games that are played as a sequence of actions) and imperfect information games are commonly represented in a way called *extensive form.* Basically, it consists of a tree in which each branch specifies an event in the game, and branches split when a player makes a decision. At the end of the tree, we have *terminal leaves*, which specify a certain return or utility for each player. *Information sets* contain all the states among which the player is not able to see the difference (for instance, raising with the opponent having a pair of aces or having a pair of 4s are two points in the same information set, but clearly the outcome can be very different). A complete strategy for a player needs to comprise a probability distribution over the possible actions for every information set in the game. Heads-up Poker, in common settings, is a *zero-sum game*: the utilities at each leaf sum up to 0. In other words, what one player loses is earned by the other one. Two powerful results help us in our quest for optimal strategies:

1) All finite games have at least one (mixed) Nash Equilibrium. By “finite” we mean that the set of players and the set of strategies are finite.
2) In zero-sum games, in each Nash Equilibrium the players have the same expected utilities. The latter is called the game-theoretic value of the game.

2.4. Introduction to RL

Now we introduce Reinforcement Learning (RL hereinafter), as it is the backbone of many bots that “learn”, and some of the best-performing systems playing Poker leverage RL.

RL is one of the most exciting fields of Machine Learning. It is quite old, as its first appearances in literature date back to the 50s. However, it has been on the rise lately, after in 2013 a British startup called Deepmind (which was then acquired by Google for more than 500 million dollars) implemented a system that could learn to play and later also outperform humans in an Atari game from scratch, without prior knowledge on the rules of the game. Then, in 2016, their “AlphaGo" won against a famous

professional player of Go and the following year also against the world champion. Nowadays RL has applications in many fields, ranging from recommender systems to self-driving cars. In RL settings, there is a software agent which observes an environment, and makes decisions in it. Following its choices, it receives positive or negative rewards. The agent learns by trial and error trying to maximize its expected rewards over time. The algorithm it uses to choose its actions is called its policy, which often is chosen to be a neural network. Every policy has its parameters to be tweaked. In order to do so, there are several methods, for instance a policy search (brute force approach), genetic algorithms (letting best policies survive over a generation, making offsprings and iterating), or policy gradients (optimization techniques).

2.4.1. Markov Decision Process

A Markov Chain is a stochastic process with no memory. It has a fixed number of states and at each step it stochastically evolves from one state to another (or the same state). The probability of going from state *s* to state *s'* depends only on the pair *(s,s')* and not on the previous history (hence explaining the "no memory" name). Markov chains have many uses in thermodynamics, chemistry, statistics, computer science and more.

Richard Bellman in 1950 introduced Markov Decision Processes (hereinafter, MDP), a modified version of Markov Chains in which at each state the agent can make a decision in a set of possible actions. Transition probabilities among states depend on the action chosen. Moreover, some steps include a positive or negative return for the agent. The latter will try to choose a policy that maximizes its return over time. The intuition is the following: MDPs are used to model decision making settings in which the outcome is partially stochastic and partially in the hands of a decision maker.
In reinforcement learning agents interact among themselves or with an environment and, over many iterations, they learn to maximize their cumulative future reward. The environment in which the agents "play" usually is modeled as a MDP. In games like Poker the stochastic component is due to the draw of the cards, and the player-controlled component that influences the outcome is the betting strategy.

At each step of the chain, the process is in some state *s* (in our case, the cards available in hand and on the table), and the decision maker may choose any action

available in that state (raise, fold, call). The process responds at the next time step by randomly moving into a new state *s'* and it gives the player a corresponding reward *R* (for instance a value for the strength of the hand). The two extensions with respect to a classic Markov Chain are the action and the respective reward of the decision maker. The agent chooses his/her strategy according to a certain distribution over any available action at each state of the MDP. She aims to maximize the gain, which is defined as the cumulative future rewards $G_t = \sum_{i=t}^{T} R_{i+1}$.

In the following figure we represented a simple Markov Decision Process. Circles represent game states whereas diamonds represent the possible actions. The black numbers represent the transition probabilities once an action is executed, green values are positive rewards and red values are negative rewards. A player starting in s_0 may choose action a_0, a_1 or a_2. Action a_1 brings back the player to the initial state with certainty, without any reward. Action a_2 will provide a positive payoff with 30% probability and a negative one with 70% chance. Finally, action a_3 will yield a gain of +5 (clearly, the unit of measure can be anything) or a "pain" of -10 with equal probability.

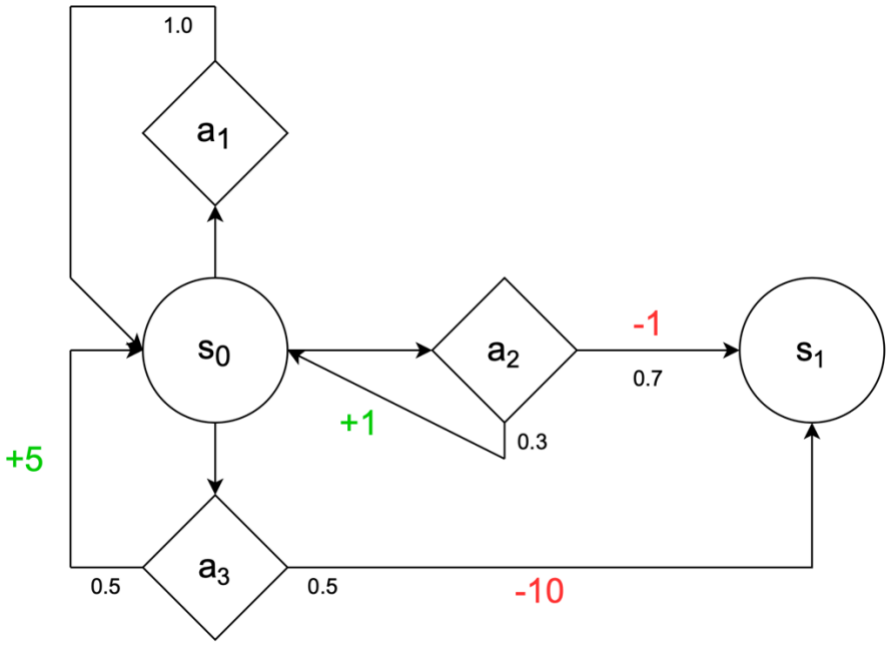

Figure 1: Example of the scheme of a Markov Decision Process

3. Regret and counterfactual regret minimization

Now that we have a first understanding of the complexity of the problem we are facing and the theoretical framework of RL, let's look at the main methodologies employed. Many state-of-the-art algorithms that allow the approximation of optimal strategies in imperfect information games are based on Counterfactual Regret Minimization. The latter is an iterative method that approximates Nash equilibria, through what is called "self-play" (even though more properly it should be play-against-a-copy-of-yourself). Self-play means that it plays against itself to learn to play a game and to improve its strategy. Two equal algorithms play against each other and learn. In other words, the algorithm learns by playing against itself without the need for human intervention. Every round it considers what decisions were made during that round and figures out how to improve its strategy. Hence, after many iterations, it may understand which is the optimal strategy. In order to do this, when playing, these algorithms minimize regret. In game theory, regret, in line with the meaning it assumes when talking about human feelings, is the loss that the player incurs for not having chosen the optimal strategy. In other words, it is a numeric value indicating how much you regret taking a specific action. Clearly, in imperfect information games, it is possible to compute it only a-posteriori, when the best possible choice, the choice made, and their outcomes are known.

From an intuition point of view, the Counterfactual Regret Minimization algorithm repeatedly *traverses* the game tree, following a certain strategy profile and accumulating regrets. When talking about games in extensive form, traversing the tree commonly means to repeatedly play the game. For each iteration over the tree, new regrets are collected, and the strategy profile is refined. The good news is that the average strategy among all the iterations will eventually converge to a Nash Equilibrium. If the aim of approximating Nash Equilibria is reached, it means that the software manages to play HUNL with great skill, possibly superhuman. We will now see in detail its functioning.

3.1. Regret and Nash Equilibria

We start by defining regret in mathematical terms and then we check its link with Nash Equilibria. Let's consider a round *t* of a game, a player *i* whose utility function[4] is u_i, its strategy at round *t* is ${\sigma_i}^t$, the optimal strategy being ${\sigma_i}^*$, and the strategy used by the other player(s) at round *t* is: ${\sigma_{-i}}^t$. Then the regret is defined as the difference between the best payoff possible assuming a certain strategy for the other player and the payoff obtained with the move chosen: $r_i^t = u_i(\sigma_i^*, \sigma_{-i}^t) - u_i(\sigma^t)$. Starting from this intuition, the *average overall regret* (Zinkevich, Johanson et al., 2007) of player *i* at time *T* is defined as the average regret of a player over *T* rounds. For each possible optimal strategy - so basically every possible strategy in the strategy set - the cumulative regret (cumulative means that it is summed over the set of iterations) is computed over the rounds in consideration. Then the maximum cumulative regret is taken to impose the condition that ${\sigma_i}^*$ is the optimal strategy. Finally, it is averaged over the number *T* of rounds: $R_i^T = \frac{1}{T} \max_{\sigma_i^* \in \Sigma_i} \sum_{t=1}^{T} \left(u_i(\sigma_i^*, \sigma_{-i}^t) - u_i(\sigma^t) \right)$. In words, the average regret is the difference between the payoff that the player obtains with the optimal choice and the actual choice he made, averaged over time. We defined the average regret because there is a strong link between average regret and Nash equilibria, and this result is particularly useful when building an algorithm to approximate the optimal strategy. Basically, in a zero-sum game at time *T*, if both players' overall regret is less than a threshold ε, then the **average strategy** of player *i* from time 1 to *T* is a 2ε-Nash equilibrium. Firstly, let's define what an average strategy is, then we will do the same with the 2ε-Nash equilibrium. Given an information set I_i and an action a_i, the *average strategy* after *T* rounds is the sum of the strategies for each round, weighted by the probabilities of reaching the information set I_i at each round, divided by the sum of such probabilities. Given an information set I_i, an action a_i, the strategy of player *I,* $\sigma_i^t(I_i, a_i)$ and the probabilities of reaching I_i at each iteration t equal to $\pi_i^{\sigma^t}$, then formally, we have that the average strategy is: $\overline{\sigma_i^T}(I_i, a_i) = \frac{\sum_{t=1}^{T} \pi_i^{\sigma^t}(I_i) \cdot \sigma_i^t(I_i, a_i)}{\sum_{t=1}^{T} \pi_i^{\sigma^t}(I_i)}$.

[4] Defined as a function, characteristic for each player and each game, that, given a certain strategy, returns the player's payoff.

An ε-Nash equilibrium is an approximate equilibrium. Instead of having no incentive to deviate, players may have a small incentive, bounded by ε. Formally, a strategy profile σ is an ε-Nash equilibrium if for every player *i*, with u_i being their utility functions, σ_i' being any alternative strategy in the strategy space, and σ_{-i} being the strategy of the other player, we have: $u_i(\sigma) \geq u_i(\sigma_i', \sigma_{-i}) - \varepsilon$.

The result about the average strategy implies that techniques that minimize regret can be effectively used to find approximate Nash Equilibria. We can recognize algorithms that minimize regret by checking whether the overall regret goes to 0 as *T* goes to infinity. Furthermore, the better the algorithm is in achieving the bounds for the regret, the stronger is the equilibrium found. From an algorithmic and decision-making point of view, how can computing the regret for a move being informative of future play? There are two main considerations that need to be done: firstly and logically, at every point in the game (except for the very first round) a player should choose the action he/she regretted the most not having selected in the past. Nevertheless, you also must avoid being predictable, as it would imply being exploitable. An interesting algorithm for this purpose is *regret matching* (Hart, Mas-Colell, 2000). Following the latter, agents select their strategies stochastically: they follow a probability distribution proportional to positive regrets. If an action has a positive regret associated to it, it means that the player previously had a loss for not choosing that action. More specifically, we use *normalized positive regrets*, which are regrets divided by their sum[5], to set the probabilities that define our mixed strategies. Over some rounds of the game, *cumulative regrets* are computed by adding the regrets found in each of the preceding rounds for each action.

[5] The same will be done with the algorithm using Counterfactual Regrets later in the paper.

3.2. Regret - Empirical approach

3.2.1. Pseudocode

Here we developed a simple pseudocode[6] for the regret matching algorithm[7] we described. For a matter of simplification, we considered a 2-players game in which one of the two players defines his/her strategy following the average strategy profile obtained through regret matching (i.e., the strategy that, as the number of iterations goes to infinity, converges to a Nash equilibrium) while the other one uses a fixed strategy. When talking about strategy profiles in this setting, since we allow for mixed strategies, we mean the probabilities with which the player decides to play each action. The main steps of the algorithm are:

- Compute a strategy profile that follows regret-matching. That is, it sets the probabilities of being played for each action as equal to the cumulative regret values (line 7). If every regret is less than or equal to 0, set the strategy as uniform among the feasible actions. It is what happens at line 15 and 17. The variable *Strategy,* which will store the mixed strategy probabilities*,* at that point contains only positive cumulative regrets, if any. If the sum of the values of the strategy profile is less than 0, it means that no regret is positive. The sum of all values in *Strategy* is computed and each value is normalized being divided by the sum (line 15).
- Sample the action profile following the strategy profile found (line 19)
- Compute the payoff of every action (line 21)
- Compute the regret of every action with respect to the action chosen (line 22). In other words, the difference in payoff between each action *a* and the action that was sampled following the strategy profile
- Add the regrets found to the cumulative regrets (line 23)

[6] I included in the appendix the Latex code I used to render the pseudocode.
[7] A thorough version in Java of the Regret Matching Algorithm for Rock Paper Scissors was implemented by Neller, Lanctot in 2013. Here we generalize using pseudo-code to remain language-neutral and game-neutral and to provide a more direct intuition.

- Repeat the above steps N times. Then, compute the average strategy (line 26) following its definition provided at the previous page. If N is sufficiently large, the profile will converge to a Nash Equilibrium.

Algorithm 1 Regret Matching algorithm pseudo code

```
1:  procedure REGRET(opponentStrategy, setOfActions, N)
2:      cumulativeRegrets ← 0
3:      for N iterations do
4:          strategy ← 0
5:          for each action a do
6:              if cumulativeRegrets[a] ≥ 0 then
7:                  strategy[a] ← cumulativeRegrets[a]
8:              else
9:                  strategy[a] ← 0
10:             end if
11:         end for
12:         normalize strategy:
13:         sumStrategy ← sum(strategy)
14:         if sumStrategy > 0 then
15:             strategy ← strategy / sumStrategy
16:         else
17:             strategy ← uniform over actions
18:         end if
19:         sample a random action from the strategy
20:         for each action a do
21:             compute the payoff of a
22:             compute the regret of a
23:             cumulativeRegrets[a] ← cumulativeRegrets[a] + regret[a]
24:         end for
25:     end for
26:     compute the average strategy
27: end procedure
```

Figure 2: Pseudo code for the regret matching algorithm

In the code above, *cumulativeRegrets* is an array that stores the values of cumulative regrets and gets updated in real time during running, *Strategy* represents the strategy profile and is an array containing probabilities with which each action is chosen by the player. *Sum(Strategy)* emulates the behavior of the Python Numpy function *Numpy.sum* which sums over all the elements of the array. Payoffs may be computed following the payoff tables of specific of the game.

3.2.2. Empirical check of the convergence

A “toy” problem often analyzed through the lens of Game Theory is the common game called Rock Paper Scissors. It is a complete information, non-sequential game. Thomas Trenner (2020) developed a simple implementation of the regret matching algorithm for it, in a similar fashion to the pseudocode and the intuition provided above. We used its implementation[8] to test empirically, in a very simple environment, the actual convergence of the average strategy obtained through regret matching to a Nash Equilibrium. We did 2 kinds of test. In the first one, one player plays following the regret minimization algorithm, whereas the other one follows a fixed strategy. What we expected was that the optimal strategy found by the algorithm would be leveraging the fixed policy of the opponent. In other words, if the opponent played *paper* 70% of the time, we would expect the average strategy after a sufficiently high number of iterations to always choose scissors, in order to win 70% of the matches. The following are the results obtained each after 10000 iterations of the regret matching algorithm. Data reported on the table represent the probabilities with which each action gets chosen by the regret-minimizing agent against fixed policy players. P1 is the “Rock” player: it plays Rock 50% of the time. P2 is the “Paper” one as it plays paper with 80% probability, and P3 slightly privileges scissors, playing it 40% of the times.

	Rock	Paper	Scissors
P1: [0.5, 0.2, 0.3]	0.003	0.997	3.3e-05
P2: [0.1, 0.8, 0.1]	6.6e-05	4.8e-04	0.999
P3: [0.3, 0.3, 0.4]	0.999	3.3e-05	3.3e-05

The results are in line with what expected: the regret-minimizing player exploits the naïve fixed strategy of his/her opponent. He/she almost always uses *paper* against P1, *scissors* against P2 and *rock* against P3. In the second test instead, two regret-minimizing players played against each other. The average solution after 10000

[8] In the appendix I included the code used for the simulations, which is based on the work of Trenner (2020)

iterations converged to $[\frac{1}{3}, \frac{1}{3}, \frac{1}{3}]$, as none of the two can be exploited. It is a Nash Equilibrium, as none of the two players has incentive to deviate.

3.3. CFR

Regret Minimization is a powerful tool. Nevertheless, it works well in small games that can be easily represented in normal form. When considering, for instance, Poker, the normal form representation is not a viable path. Therefore, the notion of counterfactual regret was introduced by Zinkevich and his colleagues (2007) for extensive games. There are two main complications to factor in: when computing regrets in an information set, one needs to consider the probability of reaching it, given the players' strategies; because of the sequential nature of extensive games, information about game states and probabilities of player's strategies must be "passed forward" over the branches of the game tree, and similarly, utility information must be "passed backward", in the sense that when taking a decision a time *t* you need to consider the expected value of future rewards, foreseeing the potential payoffs down the branches, which is nontrivial. The main idea of the solution is to split the overall regret in a set of terms, called counterfactual regrets, and instead of minimizing the overall regret, you minimize the single terms. Such methodology is called counterfactual regret minimization (CFR). Two results together allow the use of counterfactual regret minimization instead of classical regret minimization to approximate Nash equilibria:

- The overall regret is bounded by the sum of the counterfactual ones
- It is feasible to minimize counterfactual regret at each single information set.

To get the mathematical side of the algorithm, we need to introduce two concepts: instantaneous regret and counterfactual regret[9].

Instantaneous regret is a very similar concept to the regret seen above but constrained at a certain information set and an action. Let's consider an information set I_i of player i, an action a_i belonging to that information set. Let's define $\pi^{\sigma^t}(I_i)$ as the probability

[9]Here I deviate slightly from the notation used by Zinkevich et al. and I use the terminology used by Bertsimas and Paskov (2021) as it is more straightforward. The concepts and the algorithm are identical to the original ones.

of reaching information set I_i following strategy σ_t and $\pi_{-i}^{\sigma^t}(I_i)$ the contribution to such probability coming from the decisions of each player except for player i. The expected value of future rewards in a certain information set I, given that all players play following the strategy profile σ_t, is $u_i^{\sigma^t}(I_i)$. Similarly, if agent i chooses action a_i in that information set, its expected reward is $u_i^{\sigma^t}(I_i, a_i)$. Then the instantaneous regret is defined as: $r^t(I_i, a_i) = \pi_{-i}^{\sigma^t}(I_i) \cdot \left(u_i^{\sigma^t}(I_i, a_i) - u_i^{\sigma^t}(I_i)\right)$. The intuition is the following: it is the regret for always choosing action a with respect to the expected reward given the information set we are in. All weighted by other players' contribution to the probability to reach that specific information set. Then the counterfactual regret is defined as the average of the immediate regrets over T rounds: $R_i^T(I_i, a_i) = T^{-1} \sum_{t=1}^{T} r_i^t(I_i, a_i)$. The game strategy given by the CFR algorithm leverages counterfactual regrets. The intuition is the following: how to choose the action depends on the positive counterfactual regret for **not** playing that action. In case there aren't any actions with positive values of counterfactual regret for not choosing them, then the action is chosen following a uniform random distribution over the feasible actions. What we are doing is applying the regret matching algorithm we saw above, using the newly introduced concept of counterfactual regret.

First of all, we impose non-negative counterfactual regrets: $R_i^{T,+}(I, a) = \max(R_i^T(I, a), 0)$. Then, the resulting CFR strategy at time *t+1* is the following:

$$\sigma_i^{t+1}(I_i, a_i) = \begin{cases} \frac{R_i^T(I_i, a_i)}{\sum_{\alpha_i \in I_i} R_i^T(I_i, \alpha_i)} \; if \; \sum_{\alpha_i \in I_i} R_i^T(I_i, \alpha_i) > 0 \\ \frac{1}{|\{a_i\}_{a_i \in I_i}|} \; otherwise \end{cases}$$

As said, strategy at time *t+1* is selected following the amount of positive counterfactual regret for each action. The counterfactual regret for each action is divided by the sum of regrets, as we did in regret-matching. Indeed, it is the normalization done to obtain probabilities to build up a mixed strategy. The second case just applies the random uniform distribution. The similarity of the CFR algorithm with the regret matching one is evident. It is important to underline that the strategy used during the traversal of the tree does not converge to a Nash Equilibrium, the average strategy does.

Finally, it was shown that:

- Such strategy works in finding Nash Equilibria through self-play
- The bound on the average overall regret is linear in the number of information sets. This allows scalability to large domains, such as Poker.

To sum up, CFR iterates over the whole game tree, accumulates counterfactual regrets, and then refines the average strategy. The average iterative strategy converges to a Nash equilibrium.

For their experiments, Zinkevich et al. applied a modification to the theoretical CFR procedure they introduced. Since many information sets include hundreds of different game states, they considered sampling a fraction of them, improving efficiency without losing much information. They sampled deterministic actions according to a certain probability distribution. They found that thanks to this modification, they had a quadratic decrease in the cost per iteration and just a linear increase in the number of iterations needed to converge.

3.4. Monte Carlo CFR

After the original CFR concept was introduced, researchers developed several variants of it, some of which proved to be particularly effective in the domain of Poker. We will go through two relevant modifications of the vanilla CFR: the family of Monte Carlo CFR algorithms and CFR$^+$, which was the backbone of the implementation that brought to essentially weakly solving HULHE. The reason for the need of refinements is that the number of game states of Poker is hardly tractable with vanilla CFR. Hence two main modifications are usually applied: either the number of possible game states is reduced, or the number of iterations needed for the CFR algorithm to converge are reduced.

The main issue with the vanilla CFR from the paper written by Zinchevich et al, is that, theoretically, you need to traverse the entire game tree at each iteration. As we saw, they proposed a way around this through sampling in the specific domain of Poker. Even considering this sampling, there are still 2 limitations of their CFR: first of all, the sampling variant used and its relative bound are very Poker-specific and hence cannot be generalized. Secondly, classical CFR requires knowledge about the opponent's

strategy, making it hard to be practically used for online[10] regret minimization in extensive games. For the latter, there exist convex programming methods such as Lagrangian Hedging, but they are very costly in terms of computation power needed.

To solve those issues, a general framework to handle sampling in counterfactual regret minimization was proposed (Lanctot, Waugh et al., 2009). They presented a family of methods called Monte Carlo CFR (MCCFR, we like acronyms) minimizing algorithms that include: Zinchevich's vanilla CFR and its generalization, *outcome-sampling* and *external-sampling*. Many successful Poker AIs use MCCFR as their core algorithm. The basic idea behind MCCFR is that instead of traversing the whole tree for every iteration, one *block* is sampled at each iteration. A block is defined as a subset of the set of terminal histories, that are sequences of actions that reach the end of the game tree. In other words, let Z be the set of terminal histories, Q be a set of subsets of Z, such that their union spans Z. You sample one of these subsets, called blocks, and look at the terminal histories present in that single block. The above intuition is expressed in mathematical terms through the *sampled counterfactual value* at each block update and is defined as the summation over the histories z in the block, of the product of the payoff given by history z times the two probability contributions to reach that history: the one of player *i* and the one of all other players and chance, all divided by the probability of choosing history z inside the block. Formally:

$$v_i(\sigma, I|j) = \sum_{z \in Q_j \cap Z_I} \frac{1}{q(z)} u_i(z) \pi^{\sigma}{}_{-i}(z[I]) \pi^{\sigma}(z[I], z)$$

where:

- v_i is the sampled counterfactual value when block *j* is being updated
- $q(z)$ is the probability of taking history z in the current iteration
- $u_i(z)$ is the payoff of player *i* when choosing history *z*
- $\pi^{\sigma}{}_{-i}(z[I])$ is the probability of history z occurring with all players except i playing following strategy σ

[10] Online learning techniques in Machine Learning are methods in which data is made available in sequential order, and the algorithm needs to be updated in real time.

- $\pi^{\sigma}(z[I], z)$ is the probability of history z occurring with player i playing following strategy σ

It was shown that the sampled counterfactual value is equal to counterfactual value in expectation. This provides interesting evidence for considering MCCFR a possible refinement with respect to the original CFR algorithm. Tuning the choice of the elements inside the subsets Qs and the sampling probability *q(z)* we can get very different algorithms[11]. Lanctot and his colleagues considered specifically two choices of Qs, that give place to two different sampling algorithms with interesting features: outcome-sampling and external sampling.

In outcome-sampling MCCFR, Q is chosen such that each block includes just one terminal history, so the cardinality of each Q is 1. After each iteration, information sets are updated following the only history in the block. You still need to choose a sampling probability *q(z)* over terminal histories. The powerful feature is that you can choose such probability to be equal to the opponent's policy, making the latter cancel out in the update rule. Hence, we do not need to know in advance the opponent's strategy, and this technique can be used for online regret minimization. In external-sampling MCCFR, only the opponent's actions and chance (i.e. choices external to the players) are sampled. The block probabilities result in being $q(z) = \pi^{\sigma}{}_{-i}(z)$, so equal to the probability of the occurrence of history *z* given other players' strategies. External-sampling traverses just a fraction of the tree at each iteration. It can be shown that, if we assume the players will make more or less the same number of decisions, we get that the cost of each iteration in external sampling is $O(\sqrt{|H|})$, whereas vanilla CFR needs $O(|H|)$, where H is the set of possible histories in the game. So, the former needs asymptotically less time to find an approximate Nash Equilibria.

To sum up, building upon the work of Zinkevich et al, Lanctot and his colleagues defined a whole family of sampling counterfactual regret minimization algorithms, the MCCFRs. Moreover, they introduced outcome-sampling, which samples just one

[11] For instance, setting Q to contain all the elements of Z, and setting $q_1 = 1$, we would get that sampled counterfactual value would be equal to counterfactual value, so we would get the vanilla CFR.

history per iteration and leaves room for online regret minimization, and external-sampling, which, sampling deterministic strategies for opponents and for chance, manages to significantly reduce the time complexity of the iterations.

4. Is Poker solved?

4.1. Solving a game

We now try to understand whether Poker can be considered a solved game and how a further refinement of Counterfactual Regret Minimization that we saw was used in order to approximate Nash Equilibria in HULHE.

First of all, what does it mean to solve a game? Following Allis (1994), there are three different extents to which a game can be solved. A game is said to be:

- *Ultra-weakly solved* if, for the initial positions, the game-theoretic value has been determined;
- *Weakly solved* if, for the initial positions, a strategy has been determined to obtain at least the game-theoretic value, for both players, under reasonable resources[12];
- *Strongly solved* if, for all legal positions, a strategy has been determined to obtain the game-theoretic value of the position, for both players, under reasonable resources.

The **game-theoretic value** is defined as the value that a player is guaranteed to get if he/she and his/her opponents act optimally. This means that for ultra-weakly solved games, at the beginning of the game you know what the outcome would be if both players played optimally. For instance, *tic-tac-toe* is a game theoretic draw, meaning that if both players play optimally, it always ends in a draw (as is clear to every player after a few matches). Nevertheless, for weakly solved games, you may not know which is the optimal strategy that brings to the game theoretic value found. Instead, strongly solved games require the ability to find the optimal strategy starting from all legal positions in the game, hence starting from every game state. Examples of strongly

[12] "Reasonable resources" include the use of a state-of-the-art computer and several minutes of time per move.

solved games are, again, *tic-tac-toe* (just think about those moments in which, while you were playing your opponent deviated from the "right" move sequence and you exploited it immediately), but also many chess endgames[13] (see the Appendix for a graphical example of an endgame).

When talking about imperfect information games, we often do not have a finite set of possible outcomes. The strategies of the players, and the game itself, are stochastic. The game-theoretic values are then not "win", "loss" and "draw" but instead are real values, and can be obtained only in expectation after several repetitions of the game. Hence, they are often approximated. The right level of approximation could be discussed, a natural level is considering the game weakly solved if a human lifetime of play is not sufficient to understand whether the strategy proposed is optimal or not. Bowling and his colleagues, in the paper (Bowling, Burch et al., 2017) that will be analyzed in detail in the next paragraph, introduced a fourth level of solved games, useful when dealing with imperfect information games. They declare a game *essentially weakly solved* if an ε-Nash eq. is found for an epsilon sufficiently small such that its difference with 0 is statistically not significant (95% confidence) considering the matches that can be played in a human lifetime.

4.2. The solution claim

In their paper, Bowling and his colleagues declare that Heads-up Limit Texas Hold'em poker is essentially weakly solved. To try to approximate Nash Equilibria, they implemented a variant of CFR, called CFR$^+$.

4.2.1. CFR$^+$

There are two main challenges when trying to solve a game of the size of HULHE through CFR: memory and computation.

Memory is crucial since, during the whole procedure, CFR needs to save the solution and the accumulated regret values for every information set (which, after some simplifications, are still $1.38*10^{13}$, three orders of magnitude larger than the size of the games solved before the paper under consideration was published). To minimize the memory needed, they exploited compression. What they did was the following:

[13] Solving a chess endgame means determining perfect play in endgames, which are situations in which just a few pieces are left on the board, generally less than or equal to 7

1) They multiplied all values by a scaling factor
2) They truncated the values to integers
3) The resulting integers were ordered, to improve compression efficiency

This way they managed to achieve compression ratios as high as 13-to-1 for regrets and 28-to-1 for strategies. Being still several terabytes, they split the data among several computation nodes, each of which being responsible for a set of subgames[14].

On the other hand, there is the computation issue to be faced. Increasing by 3 the orders of magnitude of the information sets requires approximately three times the computational power (it's not proved, but it's a quite worrisome empirical result). Hence, Bowling et al. resorted to CFR$^+$ (introduced by Tammelin, 2014), a variant of Counterfactual Regret Minimization. There are some core differences between CFR and CFR$^+$ that allow a higher efficiency:

- Instead of sampling portions of the game tree at each iteration, like most CFR implementations do, CFR$^+$ iterates over the entire game tree, which may seem counterintuitive, but manages to reduce the computational cost.
- The regret matching procedure is altered: the one used is called *regret-matching*$^+$. The difference is the following: if an action has less than 0 regret, but at a certain point it reveals to be useful, the action will be chosen immediately after, instead of waiting for the cumulative regret to become positive.
- The last step of classic CFR, as we saw in chapter 3.3, is to compute the average strategy. The latter procedure is quite computationally costly and nontrivial. With CFR$^+$ computing and storing the average strategy is no more needed, since it was seen that, when iterating, current strategies of the players can be used as solution, as its exploitability approaches 0 (see below for the definition of exploitability).

[14] Subgames are a common concept in Game Theory. Referring to the extensive form of a game, they are subsets of games with specific characteristics: its initial node is a singleton in its information set; given a node contained in the subgame, all its successors are contained as well; if a node in the subgame is in a non-singleton information set, all other nodes in the set are part of the subgame as well.

Overall, CFR+ was observed to require much less computation power than classical CFR sampling and it allows parallelization.

The authors chose the *exploitability* as an indicator of the quality of the strategy found. It is defined as the difference between the game value and the expected amount the strategy under analysis achieves against the worst-case opponent strategy. They identified a threshold of exploitability over which an ε-Nash equilibrium, hence an approximate solution, is not distinguishable from a proper one (from here the definition of essentially weakly solved game). The intuition for the threshold is the following: let's consider a human playing 200 games of Poker per hour, 12 hours a day non-stop, for his entire life (estimated to be 70 years long). Assuming as well that that (insane) player is able to play the maximally exploitative strategy, never committing mistakes. His/her winnings will be normally distributed, and, considering a 95% confidence interval, at least one time out of 20 he/she will get a value 1.64 times (or more) the standard deviation below the expected value. From previous studies (Bowling, Johanson et al., 2008), it was seen that the standard deviation of the payoffs in a game of HULHE is 5 bb/g. The unit of measure is *big blinds per game*. So, the threshold is found by $(1.64 \times 5) \,/\, \sqrt{200 \times 12 \times 365 \times 70} \approx 0.00105$ bb/g. The value is obtained by taking the payoff that the player will get once out of 20 plays (which is 1.64 times the standard deviation, 5), all divided by the square root of the total number of matches that he/she will play in his/her lifetime. This means that if a solution has an exploitability less than 1mbb/g (milli-big-blinds/game) it can be considered to essentially weakly solve the game. It would indeed not be distinguished from an exact solution after an entire lifetime spent at playing and has a chance out of 20 to win against the worst-case-scenario opponent.

4.2.2. Insights from the solution

The strategy found through CFR+ has an exploitability of 0.986 mbb/g, so HULHE is declared to be essentially weakly solved. Moreover, they also found that the exploitability value for the dealer is higher than the one of the other players, meaning that the dealer has a competitive advantage in the game. The strategy found includes some peculiar strategic decisions, that could improve humans' play:

- It almost never *limps*, that is calling the first action instead of raising the bet

- It almost never *caps*, that is making the final raise, even when it has a pair of aces
- It *plays*, in the sense that it does not *fold*, a lot more than human players

5. Poker bots and RL

Finally, we look at some of the best performing implementations and methods that leverage Machine Learning techniques, either together with CFR or at its place. We see some examples of the possible levels of interplay between game theoretical methods as CFR and advanced machine learning methods. When implemented together, ML and especially neural networks are often used for the last step of the procedure, that is computing the average strategy looking backward[15].

5.1. Neural Fictitious Self Play

Counterfactual regret methods seen up till now focused on yielding Nash equilibria in abstractions of the domain of Poker. Many game-theoretic approaches for finding Nash Equilibria have the following issue: they are domain-specific and hardly generalizable. In those cases, the domain is generally defined through human expertise. Alternative methods rely on reinforcement learning in order to approximate Nash equilibria without need of prior knowledge.

Fictitious play (Brown, 1951) is a technique that allows us to find Nash Equilibria through learning in normal form games. Basically, each fictitious player chooses its strategies following best responses[16] to the opponents' average behaviors. Since fictitious play is used in normal form games, it does not scale well to extensive games that can hardly be represented in normal form because of their huge sizes (in terms of game states for instance), as Poker. Therefore, an extension of Self Play was introduced, called Full-Width Extensive-Form Fictitious Play (XFP) (Heinrich et al., 2015). The latter allows fictitious players to update their strategies in extensive form. Moreover, again Heinrich and his colleagues in 2015, developed another extension,

[15] In other cases, it has been used to build a model of the opponent or to find mistakes in his/her strategy
[16] Best responses are a common concept in game theory, they are strategies that yield the best possible outcome for a player, given the strategies of the opponents.

Fictitious Self-Play (FSP), which is a class of algorithms based on machine learning, able to approximate XFP. Instead of a game-theoretical approach to compute best responses and average strategies, it leverages reinforcement and supervised learning simultaneously. FSP algorithms are structured in the following way: the fictitious agents, while self-playing, generate 2 datasets from their experience. In a memory, M_{RL}, transition tuples $(s_t, a_t, r_{t+1}, s_{t+1})$ are stored, composed by: the state of the MDP at time *t* (s_t), the action chosen by the player (a_t), the reward received (r_{t+1}) and the next state the agent switches to after the action is taken (s_{t+1}). This memory is used for reinforcement learning. In another memory, M_{SL} we store just the behavior of the agent (s_t, a_t), and such memory is used for supervised learning. Basically, the reinforcement learning method allows one to find the approximate best responses to approximate data in the Markov Decision Process defined by the other players' average strategy profiles. The supervised classification technique is used to approximate the agent's own average strategy. Researchers several times built upon previous work: starting from Fictitious Play, Extensive form fictitious play was developed, then Fictitious self-play and finally Neural Fictitious Self Play (Heinrich, Silver, 2016). Every time the technique became gradually more sophisticated.

In NFSP, FSP is combined with the use of neural networks in order to approximate functions. Each agent learns from self-play. They interact with the other agents and store both the transitions and their best response behaviors, in M_{RL} and M_{SL}. Each agent trains a neural network and uses Q-learning[17] to predict the values of actions. The approximate best response strategy β is the following: it chooses a random action with probability ε, otherwise it chooses the alternative that maximizes the action values as predicted by the reinforcement learning algorithm. Separately and simultaneously, a supervised classification algorithm links game states to action probability and

[17] Q-learning is the most common off-policy algorithm, it works by watching an agent play and gradually improves its estimates for the Q-values. Q-values are estimates for values of state-action couples. Off-policy, as opposed to on-policy methods, are algorithms that, when estimating the cumulative return, assume a certain policy is followed whereas the behavior policy it executes is different. In other words, it learns from the experience of someone else (e.g., a previous policy or the one of another agent).

"looking back" it defines the average strategy π. In play, the agent chooses its moves from a mixture of the two, β and π.

The agents trained with NFSP were tested on Limit Texas Hold'em, against top 3 computer players from the Annual poker Competition. Performance was measured in milli-big-blinds per hand (mbb/h), the starting bet divided by 1000. NFSP agents obtained win rates like the ones of the top computer players of the ACPC, hence is considered to be competitive with superhuman poker software.

5.2. Deep CFR

Deep CFR (DCFR) approximates CFR using traversals of portions of the game tree (Noam Brown, Adam Lerer et al., 2019). At a high level, DCFR trains two neural networks, similarly to Neural Fictitious Self Play. The first one is a value network, used to predict the average regret. Basically, data is collected through sampling using the policy we want to approximate. More specifically, it fits a network to give approximate "advantages" for each action, divided by a factor that accounts for the probabilities of the information states. Differently from a vanilla CFR, in case all actions have negative or null advantage, it still chooses the one with the highest advantage, and not randomly. We know that the final strategy does not converge to a Nash Equilibrium, but the average strategy does. In order to compute the average policy, a second neural network is employed[18].

Finally, we have a look at two powerful AIs that managed to achieve superhuman performances (directly or indirectly, meaning beating humans in experiments or beating other AIs who previously beat human players): Libratus and the algorithm from Bertsimas and Paskov (2021). The exceptional characteristic is that both play HUNL, which has 10^{161} decision points, versus 10^{13} of HULHE, the one essentially weakly solved through CFR$^+$ (see ch. 3).

[18] An interesting implementation that takes DCFR a step further is DREAM: (D)eep (RE)gret minimization with (A)dvantage Baselines and (M)odel-free learning (Steinberger et al, 2020). DREAM builds upon Deep Counterfactual Regret Minimization and does not need a simulator of the game, being "model-free". More specifically, it achieves state-of-the-art performances between algorithms employing model-free self play reinforcement learning.

5.3. CFR and RL

Bertsimas and Paskov (2021) created a powerful poker bot for HUNL, addressing at the same time the problem of AI interpretability.

They claimed that they built an AI not only able to win against Slumbot, one of the best performing Poker bots in competitions, but also able to produce interpretable results. Interpretability and explainability are hot topics in Data Science nowadays since many of the most common algorithms (especially neural networks) are black boxes, in the sense that provide a solution without an explanation about where the solution comes from, what was the reasoning behind it.
Their bot stands at the intersection of CFR and RL. Indeed, the system is trained in self-play through a refined version of Counterfactual Regret Minimization, and then the average solution (the last step in the CFR procedure, see chap. 3.3) is found through learning algorithms, such as CART, OCT and XGBoost, and Neural Networks. More specifically, they trained four models for each learning algorithm employed: one different model for every round of HUNL Poker. OCT, which was the most interpretable method used, is based on classification trees, so the number of features the bot considers is limited (in Neural Networks that number can be very high) and at each non-final node of the decision tree it chooses a branch based on a binary question about a feature. In other words, the “decisions” the bot makes are recognizable from the tree and this makes the reasoning of the AI understandable by humans.

5.4. Libratus

In recent years, one of the most successful implementations was Libratus (Brown and Sandholm, 2018) and it was published by Nature. In a hand competition of 120000 hands, featuring a $200k prize pool, Libratus managed to defeat top human professional players in HUNL. It does not make use of expert knowledge and its methods are not poker-specific. We will now briefly discuss its game-solving approach. Libratus comprises three modules:

1) One module computes a simpler and easier-to-solve abstraction of the game, followed by game-theoretic tactics specific for the abstraction. The solution to

the latter gives a thorough strategy for the game's early rounds, but just an estimate for how to play in the game's later rounds, which are more numerous. Such strategy is called *blueprint strategy.* The algorithm employed is a modified version of Monte Carlo CFR (ch. 2).

2) In later parts of the game, another module creates a “fine-grained” abstraction for the subgame and solves the subgame in real time. It also checks that the solution to the subgame fits the larger blueprint strategy for the game. It applies *nested subgame solving*: basically, when the opponent of Libratus makes a move which is not enclosed in the abstraction, a new subgame with that action is created and solved.
3) The third module is the self-improver, which improves the blueprint strategy. It computes game-theoretic strategies for the branches missing in the blueprint. Because of the intractable size of the tree, it understands where the filling is worthwhile by looking at the opponent’s moves.

Reinforcement learning is leveraged in the third module of Libratus. It stores the data about the bets of the opponent to understand and choose which branches it should add to the blueprint strategy. Then, in the background, it computes the game theoretic strategies to solve those new branches.

6. Conclusions

“It would be disingenuous of us to disguise the fact that the principal motive which prompted the work was the sheer fun of the thing” - Alan Turing, talking about his work on games.

One could wonder about why bothering at all with Poker. Every researcher on this topic would admit that partially the reason is the fun and the inherit attractiveness of the game. Nevertheless, there are several other good reasons why studying computationally this game is relevant. Modeling and finding equilibria in multi-agent imperfect information games is complex. And Poker is as well. So, building from scratch systems able to find equilibria in Poker is a challenge, both for game theorists and computer scientists. And hard challenges are exactly what push the scientific community forward and this way efforts of researchers bring spectacular results. But it’s not just this. Real life decision settings very often (almost always) feature many

agents and imperfect information. Learning to solve poker means to be a step closer to solve general complex multi-agent situations. Hence such methods can and will be generalized to more *serious* environments (non-recreational games). Research on this topic bring general algorithmic improvements to the scientific community, which can subsequently be implemented in other game-theoretical settings, especially when dealing with large-scale models of rational multi-agent settings. Poker is simple from a logistical point of view, but very complex from a strategic one. It has many characteristics not seen in Chess, Checkers or other games that have been studied thoroughly. Managing incomplete information in a crucial topic in Computer Science, therefore Poker represents an arena for studying decision making in uncertain conditions.

Lately, game theory has been applied in several new circumstances, such as Airport Checkpoints, Flight Scheduling, Cost Guard Patrolling (Tambe, 2011). CFR methods are currently being studied to be used as decisional support in the healthcare sector (Chen, Bowling, 2012). Potential future applications include political decision-making support and agreement in self-driving car fleets. Equilibrium computation is key to achieve optimal multi-agent decision making in a robust manner.

Given the novelty of the topic - or also, the steep learning curve of the last few years - literature is scattered and it is not easy to orientate oneself among the terminology and the different methodologies. The aim of this paper was indeed to provide an introduction, as comprehensive as possible, to the world of Poker bots, through the lens of Game Theory. We saw the power of regret matching, its counterfactual evolution, and their ability to approximate Nash Equilibria. We saw theoretically and empirically the method of Regret Minimization, some refinements, and its interplay with Reinforcement Learning. We saw how Poker can be considered *essentially weakly solved,* and we also analyzed some successful implementations. Several of the methods studied look promising, and there is also room for improvement, as computational power capacity increases and as more studies are released in the coming years. We will probably see both a vertical and horizontal evolution: vertical in

the sense that AIs playing poker will become more and more sophisticated and maybe generalize to harder versions of the game; horizontal meaning that the techniques we saw will be applied to diverse domains.

Bibliography

Allis, Louis V (1994). “Searching for solutions in games and artificial intelligence.” *Ph.D. thesis, University of Limburg*. http://fragrieu.free.fr/SearchingForSolutions.pdf.

Bellman, Richard (1957). “A Markovian Decision Process.” *Journal of Mathematics and Mechanics* 6 (5): 679-684. https://www.jstor.org/stable/24900506?casa_token=FWyt0PRSkLQAAAAA%3Aju047_1qIp0jOGaSTC77NFMjA8V-dQFz7CzPwgwFkp6CQKmS-FFW9Swx4BiSOtoZGyb-DojpqLdVvWTbeE16PhywPWHz9zMpOymFp7xMuOJchwvuGbE&seq=1#metadata_info_tab_contents.

Bellman, Richard, and Robert Kalaba (1966). *Dynamic programming and modern control theory*. N.p.: Academic Press.

Bertsimas, Dimitris, and Alex Paskov (2021). “World-Class Interpretable Poker.” https://dbertsim.mit.edu/pdfs/papers/2021-InterpretablePoker.pdf.

Blackwell, David (1956). “An analog of the Minmax theorem for vector payoffs.” *Pacific Journal of Mathematics*. http://www-stat.wharton.upenn.edu/~steele/Resources/Projects/SequenceProject/Blackwell/Blackwell56.pdf.

Bowling, Michael, Neil Burch, Michael Johanson, and Oskari Tammelin (2017). “Heads-up Limit Hold’em Poker is Solved.” *Science* 347 (6218): 145-149. http://webdocs.cs.ualberta.ca/~bowling/papers/15science.pdf.

Bowling, Michael, Michael Johanson, Neil Burch, and Duane Szafron (2008). “Strategy evaluation in extensive games with importance sampling.” *ICML '08:*

Proceedings of the 25th international conference on Machine learning, 72-79. https://dl.acm.org/doi/10.1145/1390156.1390166.

Brown, George W (1951). “Iterative solution of games by fictitious play. Activity analysis of production and allocation.”

Brown, Noam, Adam Lerer, Sam Gross, and Tuomas Sandholm (2019). “Deep counterfactual regret minimization.” *International Conference on Machine Learning*, 793-802. https://arxiv.org/pdf/2006.10410.pdf.

Brown, Noam, and Tuomas Sandholm (2018). “Superhuman AI for heads-up no-limit poker: Libratus beats top professionals.” *Nature* 359, no. 6374 (January): 418-424. https://science.sciencemag.org/content/sci/359/6374/418.full.pdf?_ga=2.248058311.1559371770.1612126135-1067001828.1610992973.

Chen, Katherine, and Michael Bowling (2012). “Tractable Objectives for Robust Policy Optimization.” *Advances in Neural Information Processing Systems* 25:2078-2086. https://papers.nips.cc/paper/2012/hash/ce5140df15d046a66883807d18d0264b-Abstract.html.

Deepmind (2016). “Mastering the game of Go with deep neural networks and tree search.” *Nature*. https://www.nature.com/articles/nature16961.

Deepmind (2017). “Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm.” https://arxiv.org/abs/1712.01815.

Géron, Aurélien (2019). *Hands-On Machine Learning With Scikit-Learn, Keras, And TensorFlow: Concepts, Tools, And Techniques To Build Intelligent Systems*. N.p.: O'Reilly.

Hart, Sergiu, and Andreu Mas-Colell (2000). “A simple adaptive procedure leading to correlated equilibrium.” *Econometrica* 68, no. 5 (September): 1127-1150. https://www.ma.imperial.ac.uk/~dturaev/Hart0.pdf.

Heinrich, Johannes, and David Silver (2016) “Deep Reinforcement Learning from Self-Play in Imperfect-Information Games.” *arXiv*. https://arxiv.org/pdf/1603.01121.pdf.

Johannes, Heinrich, Marc Lanctot, and Daniel Silver (2015). “Fictitious self-play in extensive-form games.” *Proceedings of the 32nd International Conference on Machine Learning*. http://proceedings.mlr.press/v37/heinrich15.pdf.

Johanson, Michael (2013). “Measuring the Size of Large No-Limit Poker Games.” *arXiv*. https://poker.cs.ualberta.ca/publications/2013-techreport-nl-size.pdf.

Kuhn, H. W. (1950). "Simplified Two-Person Poker". In Kuhn, H. W.; Tucker, A. W. (eds.). Contributions to the Theory of Games. 1. Princeton University Press. pp. 97–103.

Lanctot, Marc, and Deepmind (2017). “A Unified Game-Theoretic Approach to Multiagent Reinforcement Learning.” *Advances in Neural Information Processing Systems 30 (NIPS 2017)*, (November). https://arxiv.org/pdf/1711.00832.pdf.

Lanctot, Marc, Kevin Waugh, Martin Zinkevich, and Michael Bowling (2009). “Monte Carlo Sampling for Regret Minimization in Extensive Games.” *Advances in Neural Information Processing Systems 22 (NIPS 2009)*. https://papers.nips.cc/paper/2009/file/00411460f7c92d2124a67ea0f4cb5f85-Paper.pdf.

Neller, Todd W., and Marc Lanctot (2013). “An Introduction to Counterfactual Regret Minimization.” http://modelai.gettysburg.edu/2013/cfr/cfr.pdf.

OpenAI (2020) “Emergent tool use from multi-agent autocurricula.” https://arxiv.org/pdf/1909.07528.pdf.

Sam Ganzfried and Tuomas Sandholm (2014). Potential-aware imperfect-recall abstraction with earth mover’s distance in imperfect-information games. In Proceedings of the TwentyEighth AAAI Conference on Artificial Intelligence, page 682–690. AAAI Press.

Steinberger, Eric, Adam Lerer, and Noam Brown (2020). “DREAM: Deep Regret minimization with Advantage baselines and Model-free learning.” *arXiv*. http://aaai-rlg.mlanctot.info/papers/AAAI21-RLG_paper_56.pdf.

Tambe, Milind (2011). *Security and Game Theory: Algorithms, Deployed Systems, Lessons Learned*. N.p.: Cambridge University Press.

Tammelin, Oskari (2014). “Solving Large Imperfect Information Games Using CFR+.” *ArXiv*. https://arxiv.org/abs/1407.5042.

Trenner, Thomas (2020). “Steps to building a Poker AI - Part 4: Regret Matching for Rock-Paper-Scissors in Python.” Medium. Last access: 10/06/21 https://ai.plainenglish.io/steps-to-building-a-poker-ai-part-4-regret-matching-for-rock-paper-scissors-in-python-168411edbb13.

Von Neumann, John, and Oskar Morgenstern (1944). *Theory of Games and Economic Behavior*. United States: Princeton University Press.

Zinkevich, Martin, Michael Johanson, Michael Bowling, and Carmelo Piccione (2007). “Regret Minimization in Games with Incomplete Information.” *NIPS'07:*

Proceedings of the 20th International Conference on Neural Information Processing Systems, (December), 1729-1736. https://papers.nips.cc/paper/2007/file/08d98638c6fcd194a4b1e6992063e944-Paper.pdf.

Appendix

Latex code for regret matching pseudocode

The following is the Latex code used to render the pseudo-code in chapter 3.2.1.

```
\begin{algorithm}
    \caption{Regret Matching algorithm pseudo code}
    \label{regret}
    \begin{algorithmic}[1] % The number tells where the line numbering should start
        \Procedure{Regret}{$opponentStrategy,\;setOfActions, \; N$}
        \State $cumulativeRegrets \gets 0 $ %\Comment{We initialize cumulative regrets}
            \For{$N \; iterations$}%\Comment{N should be set as large as possible}
                \State $strategy \gets 0 $
                \For{$each \; action \; a$}%\Comment{Create strategy through regret matching}
                    \If{$cumulativeRegrets[a] \geq 0$ }
                        \State $strategy[a] \gets cumulativeRegrets[a]$
                    \Else \State $strategy[a] \gets 0$
                    \EndIf
                \EndFor
                \State \textbf{normalize} strategy:
                    \State $sumStrategy \gets sum(strategy)$  %\Comment{Sum of all strategies}
                    \If{ sumStrategy $>$ 0 }
                        \State $strategy \gets strategy \; / \;sumStrategy$
                    \Else \State $strategy \gets uniform\; over\; actions $
                    \EndIf
                \State \textbf{sample} a random action from the strategy
                \For{$each \; action \; a$}
                \State \textbf{compute} the payoff of $a$
                \State \textbf{compute} the regret of $a$
                \State $cumulativeRegrets[a] \gets cumulativeRegrets[a] + regret[a]$
            \EndFor\label{regretendfor}
            \EndFor
            \State \textbf{compute} the average strategy
        \EndProcedure
    \end{algorithmic}
\end{algorithm}
```

Figure 3: Latex code for the pseudocode

Python code for RPS simulations

Here I enclose the Python Code I used for the Rock Paper Scissors simulations in chapter 3.2.2.

In [27]:
```python
from enum import Enum
from random import choices
from typing import List
import numpy as np
from matplotlib import pyplot as plt

class Action(Enum):
  ROCK = 0
  PAPER = 1
  SCISSORS = 2


def get_payoff(action_1: Action, action_2: Action) -> int:  #payoff for player 1
    mod3_val = (action_1.value - action_2.value) % 3
    if mod3_val == 2:
        return -1
    else:
        return mod3_val

def get_strategy(cumulative_regrets: np.array) -> np.array:    #regret-matching strategy
    pos_cumulative_regrets = np.maximum(0, cumulative_regrets)
    if sum(pos_cumulative_regrets) > 0:
        return pos_cumulative_regrets / sum(pos_cumulative_regrets)
    else:
        return np.full(shape=len(Action), fill_value=1/len(Action))


def get_regrets(payoff: int, action_2: Action) -> List[int]: #non negative regrets
    return np.array([get_payoff(a, action_2) - payoff for a in Action])

```

In [16]:
```python
opp_strat = [[0.5, 0.2, 0.3], [0.1, 0.8, 0.1], [0.3, 0.3, 0.4]]
```

In [26]:
```python
opt_strats=[]
for i in range(3):
    cumulative_regrets = np.zeros(shape=(len(Action)), dtype=int)
    strategy_sum = np.zeros(shape=(len(Action)))
    opp_strategy = opp_strat[i]
    num_iterations = 100000
    optimal_strategy_tot=[]
    for _ in range(num_iterations):
        strategy = get_strategy(cumulative_regrets)
        strategy_sum += strategy
        our_action = choices(list(Action), weights=strategy)[0]
        opp_action = choices(list(Action), weights=opp_strategy)[0]
        our_payoff = get_payoff(our_action, opp_action)
        regrets = get_regrets(our_payoff, opp_action)
        cumulative_regrets += regrets
        optimal_strategy = strategy_sum / num_iterations
        optimal_strategy_tot.append(optimal_strategy)
    optimal_strategy = strategy_sum / num_iterations
    opt_strats.append(optimal_strategy)
    print(optimal_strategy)

```

Figure 4: Python code for the simulations

Images

Example of a Chess Endgame (source: *Fundamental Chess Endings: A New One-volume Endgame Encyclopedia for the 21st Century*):

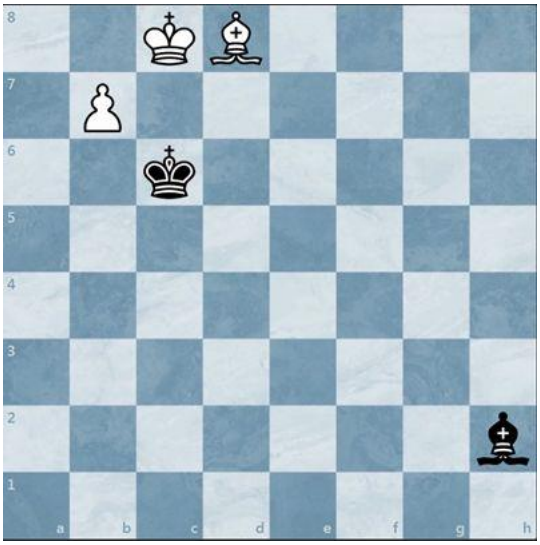

Figure 5: 5-pieces Chess endgame