
POKER AI: Using Monte-Carlo CFR with Monte-Carlo Simulations

School of Computer Science and Informatics, Cardiff
University



Manuel Miranda Mendes da Luz - 22042507

Supervised by Federico Liberatore

Moderated by Padraig Corcoran

CM3203

Abstract

There are countless poker bots in research. However, full implementations of competitive poker bots are scarcely available. My project implements Monte-Carlo Counterfactual Regret Minimization (MCCFR) alongside Monte-Carlo simulations to create a competitive Heads-Up No-Limit Texas hold'em (HUNL) poker AI. The MCCFR portion of the bot trains on a simplified “toy” poker, containing similar rules to real HUNL poker. The “toy poker” is heavily abstracted, resulting in a game tree size that is magnitudes smaller. This allows for strategy to converge to Nash Equilibrium through self-play.

Using these algorithms, my project aims to construct a lightweight and competitive poker agent that can be reproduced with personal computing hardware. While making a poker bot is not a new concept, most successful poker bots require large computational power which most hobbyists do not possess. Consequently, my bot allows a wider participation in game theory research. The complete implementation and experimental results are made publicly available to support reproducibility.

To evaluate performance, the agent was tested against a suite of baseline bots representing common heuristic and simple strategies. The agent consistently outperformed all baselines, achieving win rates exceeding 0.8 small blinds per game against weaker opponents whilst maintaining positive profitability across some research bots.

Acknowledgements

I would like to express my deepest appreciation to my family and close friends who supported me during this period. Additionally, this project stands on the shoulders of a plethora of researchers like the University of Alberta Computer Poker Research Group and more specifically Tuomas Sandholm and Noam Brown whose papers I have come across so many times, inspiring me to take on this project. These researchers' work on CFRs and abstraction of poker have been essential for me to build my own poker bot.

Contents

Abstract	2
Acknowledgements	2
1. Introduction.....	6
1.1. Research Motivation.....	6
1.2. Research Objectives and Aims	7
2. Background	7
2.1. Poker Overview	7
2.2. Unique Problems with Poker	8
2.2.1. Imperfect Information	8
2.2.2. Large Game Tree	9
2.3. Game Theory	11
2.3.1. Nash Equilibrium.....	11
2.3.2. Game-Theory optimal (GTO) vs Exploitative Strategy	11
2.4. Poker Bots Methods	12
2.4.1. Vanilla CFR	12
2.4.2. Monte-Carlo CFR (MCCFR)	13
2.4.3. CFR +	15
2.4.4. Deep CFR.....	15
2.4.5. Rule-Based Approach.....	15
2.4.6. Simulation-based Approach	15
2.5. Poker Bots Overview.....	16
3. Design and Implementation of CFRv1	18
3.1 Overview	18
3.2. MCCFR Training.....	19
3.2.1. MCCFR code	20
3.2.2. Action Abstraction Implemented.....	23
3.2.3. Information Abstraction Implementation.....	23
3.2.4. Limiting rounds.....	24
3.3. Data Processing	24
3.4. Monte-Carlo Simulations	24
3.5 PyPokerEngine Environment.....	25
3.6. APIs and Tools	26
3.6.1. PyPokerEngine.....	26
3.6.2. Faster Hand Evaluator	27

4. Experiments	27
4.1 Experimental Setup	27
4.2. Challenges and Limitations.....	28
4.2.1. Computational Cost of MCCFR Training	28
4.2.2. Lack of Accessible Research Source Code	28
4.2.3. Time Taken for Testing	28
5. Results	29
5.1 Experiment's Measure.....	29
5.2 Heads-up Games	29
5.3. Evaluating Performance	30
5.4. Variance and Statistical Significance	31
6. Conclusion	31
7. Future Work	32
7.1. Computational Improvements.....	32
7.2. Competing against Research Bots and Human Opponents.....	32
7.2.1 Research Bots.....	32
7.2.2 Human Opponents	33
7.3. Using a more sophisticated Monte-Carlo Simulation	33
7.4. Less Abstraction	33
8. Reflection on Learning.....	34
Appendices	34
Appendix A: CFR with Chance Sampling Pseudocode	34
Appendix B: MCCFR Training	35
Appendix C: Information Set Reduction.....	36
Appendix D: Time Taken for One Game:.....	38
References	38

List of Figures

Figure 1 – Kuhn Game Tree (Professor Bryce, 2023)	9
Figure 2 - Action Abstraction.....	10
Figure 3 - Action Abstraction Simplified	10
Figure 4 - External Sampling	14
Figure 5- External Sampling Explored.....	14
Figure 6 - CFR to PyPokerEngine Pipeline	19
Figure 7 - Class Node	20
Figure 8 - CFRTrainer	21
Figure 9 - CFR Part 1	22
Figure 10 - CFR Part 2	22
Figure 11 - Monte-Carlo Simulation Code	25
Figure 12 - Monte-Carlo Win Rate.....	25
Figure 13 – Monte-Carlo Thresholds	25
Figure 14 - PyPokerEngine Bot	26
Figure 15 - Reducing Memory Usage	28
Figure 16 -MCCFR.....	35
Figure 17 - Average game Value across Iterations.....	36
Figure 18 - Training- Time Taken.....	36
Figure 19 - Information Set Before	37
Figure 20 - Information Set After	38
Figure 21 - CFRv1 VS CFRv2.....	38

1. Introduction

This project aims to create a poker bot for Heads-Up No-Limit Texas Hold'em (HUNL) poker using the Monte-Carlo Counterfactual regret minimization (MCCFR) algorithm with Monte-Carlo simulations. Poker presents a compelling domain for AI research, given its inherent qualities of hidden information and vast game tree. Development of a bot capable of excelling in an environment like Poker provides valuable insights into decision-making under uncertainty.

Poker's distinctive characteristic: imperfect information and large game tree could be applicable to other practical problems such as actions, negotiations, and security applications. **(Sandholm, 2015)**

Furthermore, poker has historically served as benchmark in AI and Game theory, significantly influencing the evolution of strategies for handling incomplete information and opponent modelling. Therefore, creating a sophisticated poker bot can contribute valuable knowledge to the AI community, especially in refining algorithms like CFR to be adapted into other practical settings.

1.1. Research Motivation

Poker has long served as a benchmark problem in AI research due to its inherent complexities of imperfect/hidden information and large game state **(Brown & Sandholm, 2017)**. Unlike deterministic games such as chess or Go, where the environment is fully observable by any player, poker agents must make decisions with uncertainty regarding key aspects of the game like hidden cards and future community cards.

Over the recent years, many advancements have been achieved mostly by utilizing Counterfactual Regret Minimisation (CFR) algorithms. Notably, through the work of researchers at Carnegie Mellon University, Libratus and Pluribus have achieved superhuman performance in HUNL **(Brown & Sandholm, 2017)**.

Despite significant achievements, challenges remain, particularly related to computational efficiency and practicality. CFR methods, although effective, traditionally require extensive computational resources, posing challenges to their usage on moderate hardware. This motivates further exploration and standardization of MCCFR to a computationally cheap and competitive agent.

This project is motivated by an interest in extending recent successes by implementing and evaluating a MCCFR with Monte-Carlo Simulation bot specifically for Heads-Up No-Limit Poker. It aims to balance competitiveness and computational feasibility. It will be benchmarked against both baseline and research bots. This work contributes to a deeper understanding of how such algorithms perform in constrained computational contexts.

1.2. Research Objectives and Aims

The primary objective of this project is to design, implement and evaluate a Heads-Up Poker bot using MCCFR with Monte-Carlo Simulations. The evaluation will be conducted by assessing the bot's performance in competition with various opponents.

Key aims are:

1. **Beat Baseline Bots:** The agent should be able to beat baseline bots with relative ease over a large sample of games.
2. **Beat Research Bots:** The agent should be winning against Research bots over a large sample of games.
3. **Achieve Human-Comparable Response Times:** Response time should not exceed a human's normal response time for poker. PokerStars, a popular online poker application, allows for a maximum of 35 seconds to act in no-limit formats (Callahan, 2016). Therefore, the agent should not exceed 35 seconds when decision making.
4. **Be Lightweight:** The agent should run effectively on modest computing resources: optimized for speed and memory usage.

2. Background

2.1. Poker Overview

Poker is a widely popular card game with many variants. The focus of my research will be Heads Up No-Limit Texas Hold'em poker. Heads-Up No-Limit Poker restricts gameplay to only 2 players and No-Limit means that bets can be of any size.

I chose Heads-Up No-Limit Poker predominantly for the two following reasons:

- 1) **Popularity:** Heads-up Texas Hold'em poker is more popular than any other non-Texas hold'em poker variant, with videos of professional poker players amassing millions of views (PokerGo, 2020). Such popularity means my research will be more relevant in modern society.
- 2) **Tractability:** Heads-Up limits the game state size, with only 2 players needing consideration, making the game tree more tractable. Additionally, there are fewer chance nodes for 2 players since only 4 private cards need to be dealt (2 for each player)

Rules of Heads-Up Texas Hold'em Poker:

- Each of the 2 players is initially dealt 2 private cards, the private cards are referred to as "hole cards."
- Both players have a set amount of chips known as their "stack size", they can only bet up to the value of their stack size
- One of the players is assigned as small blind and the other big blind. These roles are swapped after every game; small blind and big blind are forced bets. Big blind is twice the value of the small blind
- Small blind player initiates the first action every round

- During their turn, players may choose the 3¹ following actions:
 - Fold: player forfeits the round, and therefore loses any chips they wagered (including forced bets)
 - Call: player matches the current highest bet to remain active in the round
 - Raise: The player increases their bet by more than the highest bet.
- There are 4 rounds of betting:
 - Preflop: players are given their hole cards
 - Flop: 3 community cards are shown
 - Turn: 1 more community card is shown
 - River: 1 more community card is shown
- The goal of the game is to have the strongest 5-cards from any combination of the hole cards and the 5 community cards. The strength of the cards correlates with its rarity. For example, a royal flush requires 5 of the 7 cards to be exactly A, K, Q, J, 10, with the same suit, thus making it the strongest combination of cards. By contrast, a pair, such as 3,3 of any suit, is relatively weak due to its common occurrence.
- The player holding the strongest combination of 5 cards at the end of the final round (river), or the player remaining after the opponent folds, wins the entire pot.

For a more detailed explanation of poker rules, please refer to **(Howcast, 2013)**

2.2. Unique Problems with Poker

2.2.1. Imperfect Information

The first major problem that makes solving poker difficult is imperfect information, since the player's cards are unknown to opponents. Poker agents must deal with bluffing and misinformation from the other player. Therefore, poker agents need to make decisions with uncertainty. In contrast, other popular games, like chess, possess perfect information, allowing algorithms such as minimax to be utilised. Traditional Minimax is fundamentally broken when used for poker since it relies on perfect information **(Brown & Sandholm, 2016)**.

Poker is analysed through information sets. An information set is a collection of game states that a player cannot distinguish between, given the information available to them **(Zinkevich, Johanson, Bowling, & Piccione, 2007)**. For example, if a player is dealt a pair of aces at preflop and it is their turn to act, the player's information set is simply AA (ignoring suits). This demonstrates that the player is only aware of their own cards and do not precisely know where in the game state they are, as the opponent could have a plethora of combination of cards. Therefore, information sets encompass many nodes.

Information sets can be visualised through a simpler version of poker, known as Kuhn poker. In Figure 1 the nodes connected with dotted lines represent nodes in the same information set.

¹ There is essentially only 3 meaningful actions. However, there is an action known as a check which is the same as calling if calling was free and, all in, which is raising to the max amount of the player's stack.

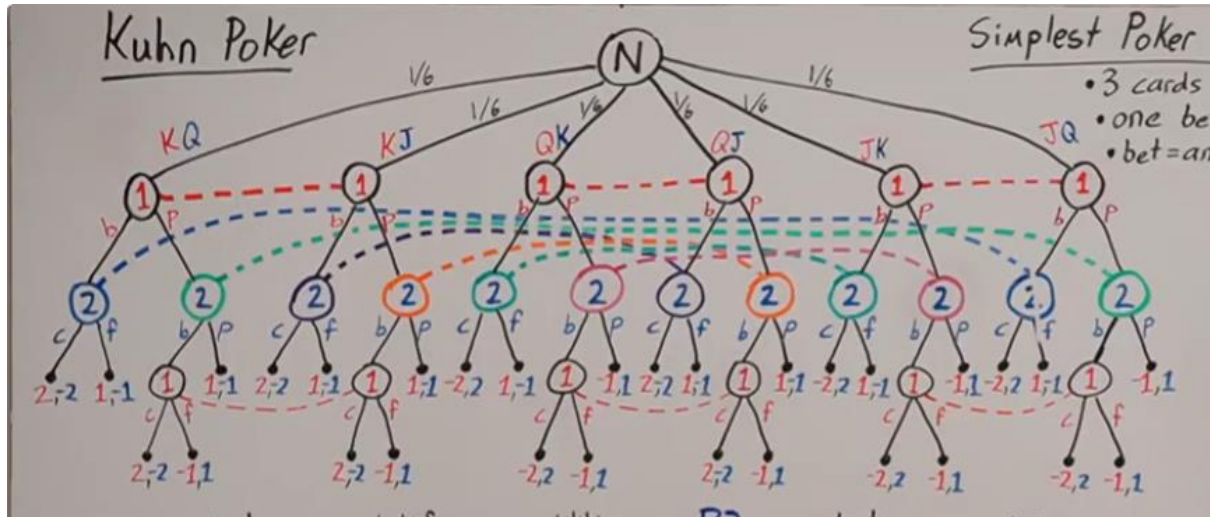


Figure 1 – Kuhn Game Tree (Professor Bryce, 2023)

2.2.2. Large Game Tree

Another problem that makes solving Poker difficult is the immense game tree size.

This size comes from several factors:

- Combinatorial explosion of private card combinations.
- Numerous possible betting sequences with variable bet sizes.
- Combinatorial explosion of community card combination.

Researchers estimate that Heads-Up No-Limit Texas Hold'em poker has 10^{161} decision points, making it magnitudes larger than Heads-Up Limit Texas Hold'em poker, with 10^{13} decision points (Brown & Sandholm, 2017).

Chance elements play a significant role in the large game tree for Heads-Up Poker. At the very first chance node, no cards are yet known thus there are $52C4 \approx 270,000$ possible combinations of private cards dealt. For agents searching the game tree, since they already hold 2 private cards, the total number of outcomes at the rest of all the chance node is $50C7 \approx 9.99 \times 10^7$. This makes poker computationally prohibitive to search, because algorithmically these chance nodes greatly increase the branching factor.

Traditional brute-force search and “lookahead” algorithms are made redundant, including minimax, since exploring game trees of this magnitude is infeasible (Gilpin & Sandholm, 2006).

To manage the large state space, game abstraction is used. This is a method that almost every CFR poker bot uses, from early bots like Hyperborean, to Libratus (Gibson, 2012; Brown & Sandholm, 2017).

Abstractions can either be lossy or lossless. Lossless abstraction arises from equivalent situations whereas lossy abstractions arise from near-equivalence. Lossless abstraction does not impact the strength of the model whereas lossy will. (Brown & Sandholm, 2019)

There are 2 types of abstractions:

1) Action abstraction: Groups actions that are “similar”, predominantly used for raising action.

Action abstraction can be seen in Figure 2.

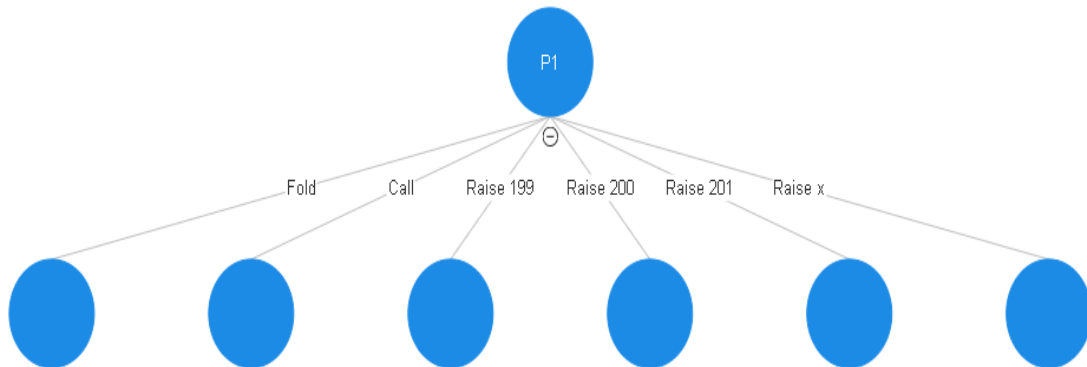


Figure 2 - Action Abstraction

Player 1 could have 100s of nodes to evaluate here if no action abstraction is utilized. This is because player 1 is allowed to raise by any integer value, ranging from minimum bet to their stack size. For instance, evaluating the difference between raising by 200 or 201 chips gains minimal benefit and requires more computation.

Player 1 can group these actions and assume it is raising by 200, as seen in Figure 3:

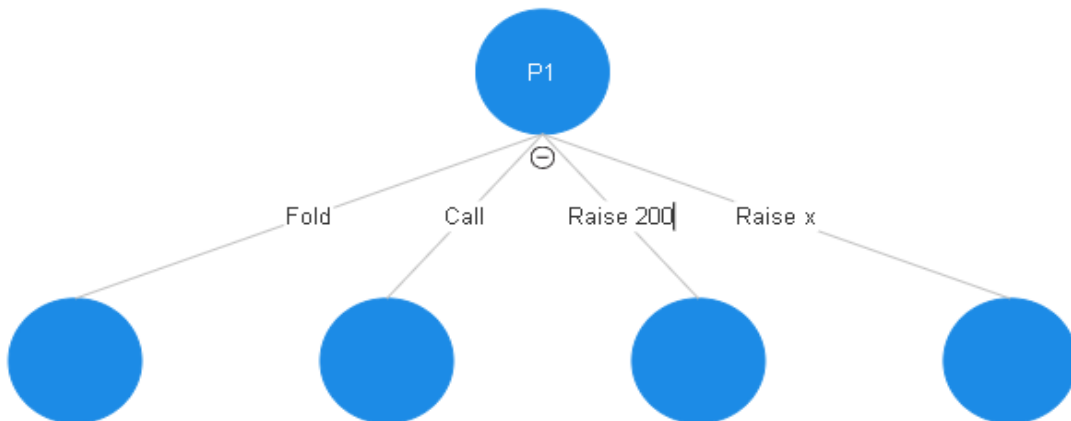


Figure 3 - Action Abstraction Simplified

Action abstraction is lossy form of abstraction since precise raising strategies are “washed away”.

2) Information Abstraction:

Information abstraction clusters private and community cards into buckets. These buckets are normally based on similarities.

A common Information abstraction used in CFR algorithms is altering the private card representation. For example, the only information an agent has during preflop is their own private cards and the actions of their opponents, so the agent should be unconcerned with the community cards. Since community cards do not impact pre-flop strategy, there is no difference between Ace of Hearts, Jack of Hearts and Ace of Spades and Jack of Spades

(AH, JH vs AS, JS). Consequently, the starting 1326 distinct possible card combinations (52C2) can be consolidated to 169 by representing hole cards as pairs (A, A) off suit combinations (AJo) and suited combinations (AJs). If this abstraction is only used for preflop it would be a lossless abstraction. **(Waugh, 2015)**

Moreover, agents such as Pluribus bucket similar information sets into a single bucket and process them uniformly, illustrating a form of lossy abstractions. However, it drastically reduces the game tree size and minimally hinders the bot's performance. **(Brown & Sandholm, 2019)**

An example of an original abstraction method is to remove card suits from information sets, thus only bucketing the rank of cards. The impact of this abstraction on performance is unknown, but it will drastically reduce the game tree, exemplifying a lossy abstraction method.

2.3. Game Theory

Game Theory is used to solve Poker; the aim is to reach a near optimal strategy by attempting to reach the Nash Equilibrium through self-play.

2.3.1. Nash Equilibrium

Nash Equilibrium is a situation whereby no player can gain utility by changing their strategy.

A prime demonstration of Nash Equilibrium is rock paper scissors. The Nash Equilibrium for each player is 1/3 for rock, 1/3 for paper and 1/3 for scissors, assuming a win gives a reward of 1, tie 0 and loss -1. If player 2 abandons this equilibrium and only plays scissors, their strategy is no longer optimal. Player 1 could exploit this by always choosing rock, maximizing their own utility. However, player 1's original equilibrium strategy (1/3 for each action) remains unexploitable, guaranteeing no long-term loss even if Player 2 deviates. **This is the core concept behind CFR poker bots.** They aim to approximate Nash Equilibrium strategies to avoid being exploited, regardless of opponent's suboptimal play

CFR algorithms have been proven to converge to Nash Equilibrium **(Zinkevich, Johanson, Bowling, & Piccione, 2007)**.

CFR-based algorithms, playing at Nash Equilibrium and thus unexploitable, are particularly useful for poker. Benefits and drawbacks will be explained in the next section 2.3.2 Game-Theory Optimal (GTO) vs Exploitative Strategy.

2.3.2. Game-Theory optimal (GTO) vs Exploitative Strategy

In Poker there are 2 strategies that are commonly used: Game-Theory Optimal and Exploitative Strategy.

- Game-Theory Optimal (GTO) maximizes the worst-case performance by attempting to reach Nash Equilibrium. GTO seeks to find the most 'balanced' strategy, meaning an agent only using GTO strategy would be unexploitable **(gtowizard, 2022)**. Consequently, it does not adapt to the type of opponent it is against. If a good GTO strategy is employed, it will likely win against an opponent over many games at a steady rate. All CFR algorithms are game theory optimal since they reach Nash

Equilibrium through self-play. CFR algorithms can be thought of as a “perfect defence in nature” (**Johanson, 2015**).

- Exploitative strategy is an older concept used in poker. It attempts to categorise the opponents into a model, changing its strategy to maximise profitability. For example, a common player model in poker is tight and aggressive (TAG). TAG players play a small selection of private cards, hence the term “tight”. They also bet frequently, hence the term “aggressive”. This player model exploits beginner poker players who are more likely to play a higher percentage of private cards and repeatedly call until the end of the game. Although TAG is effective against beginners, it is highly exploitable. Playing a low percentage of private cards allows opponents to take the TAG’s blinds and antes, resulting in a diminishing stack size in the long run. For more details on Opponent modelling see (**PokerStrategy.com, n.d.**)

It is important to note that most current Poker bots generally use a combination of GTO and exploitative strategy. For instance, GTO could be used during the first betting round to determine what sort of cards a player should or should not play. Exploitative play could be used during the last betting round to estimate what action could maximise profit. This is because exploitative strategy can outperform GTO in profitability during certain situations. Particularly, if the agent is the last to act, exploitative play is more beneficial because the opponent is not allowed to exploit it.

2.4. Poker Bots Methods

This section will explore some of the more prominent poker bots and methodologies employed. It is important to note that some of these methods are not mutually exclusive, and modern poker bots usually utilise a hybrid approach.

2.4.1. Vanilla CFR

The first CFR was successfully implemented by researchers in 2007 (**Zinkevich, Johanson, Bowling, & Piccione**), known as Vanilla CFR.

A key component of CFR is regret-matching. Regret-matching is an algorithm that explores other options to check if utility could be gained (**Zinkevich, Johanson, Bowling, & Piccione, 2007**). For example, if Player A picks scissors against rock in wagered rock paper scissors, where the winner gains 1 chip and the loser minus 1, Player B regrets not picking paper by +1, rock by 0 (since it’s a tie) and scissors by -1. Positive regret means that the player would have done better by picking that action and negative regret means the player would have done better by **not** picking that action.

If Player B plays against player A again, and player A favours scissors by a slight amount [0.3, 0.3, 0.4], whereby the first element is their chance of playing rock, second element paper and third element scissors. Then if player B initially starts with a uniform strategy [0.33, 0.33, 0.33], overtime Player B is going to start to regret not playing more rock (since rock beats scissors). So much so that if Player B starts regret minimising against player A (assuming Player A’s strategy will not change) Player B’s strategy would change to always playing rock as that would minimise the most regret. This scenario is unrealistic because Player A is likely to change their strategy since Player B is also changing their strategy. If the

opponent also uses regret minimisation, then the average strategies of both players will eventually reach a Nash Equilibrium, whereby in the long run, the regret will be 0.

CFR is a regret-based algorithm for sequential games like poker (**Zinkevich, Johanson, Bowling, & Piccione, 2007**). It adds upon regret matching and minimisation by introducing counterfactual regret. Counterfactual regret is the idea that regret can be broken down into additive regret and if you minimise the smaller additive regrets, you minimise the overall regret. Because of imperfect information in poker, a player does not necessarily know what node they are in since they do not know the opponents hole cards. However, the player is aware of the history of gameplay and their hole cards. This constructs an information set whereby the player must act accordingly, as if they could be in multiple nodes at once. CFR iterates over every information set and every action possible in the information set and calculates the instantaneous regret. Instantaneous regret is the difference in payoff between:

- (a) the outcome if, at an information set, the player had taken an action a (and played optimally thereafter),
- (b) the outcome under the current strategy profile.

These regrets are then accumulated over many iterations, using regret matching, the next iteration's strategy at that information set is chosen proportionally to the positive part of these cumulative regrets. Over many iterations, cumulative regrets tend to 0, and the strategy reaches an equilibrium. (**Zinkevich, Johanson, Bowling, & Piccione, 2007**).

2.4.2. Monte-Carlo CFR (MCCFR)

Monte-Carlo CFR was introduced by researchers in 2009 because vanilla CFR could not easily scale to No-Limit Texas Hold'em (**Lanctot, Waugh, Zinkevich, & Bowling**). In Monte-Carlo CFR, a sampling method is used to reach the end of the game (a terminal state node) and only the sample is updated, not the full tree. This is beneficial because it updates the tree much faster than vanilla CFR, however the tree updates are more erratic due to sampling (**Chiswick M. , 2021**). It also converges to a Nash equilibrium. Two sample methods are commonly used, external sampling and chance sampling:

- External sampling samples both chance nodes (dealing out hole cards and community cards) and the opponent's actions.
- Chance sampling only samples chance nodes.

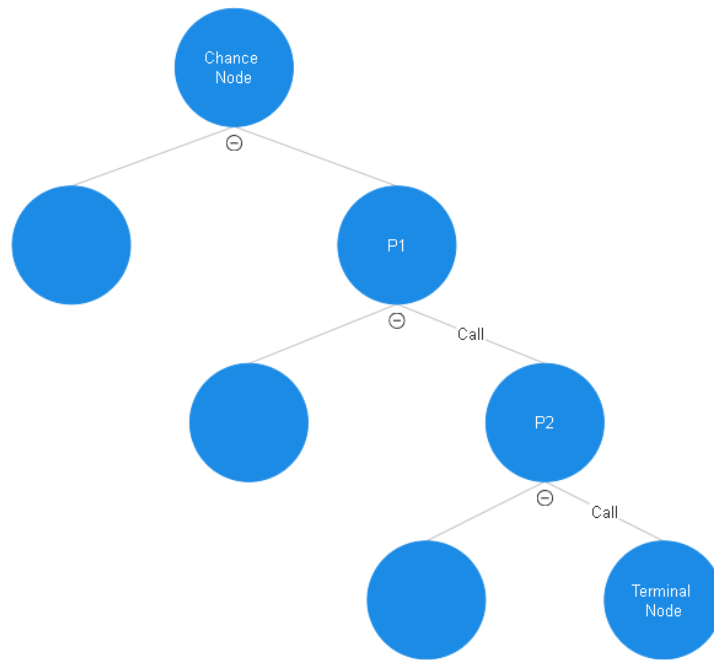


Figure 4 - External Sampling

Figure 4 shows an abstracted version of the game tree, whereby External Sampling has been used. It shows that chance nodes are sampled therefore only one outcome of the chance node is explored and additionally, P2's actions have also been sampled to a single action (call) to reach the terminal node. After the game is finished the transversing player sees how much better they could have done by choosing other actions and this is added to the action's regret.

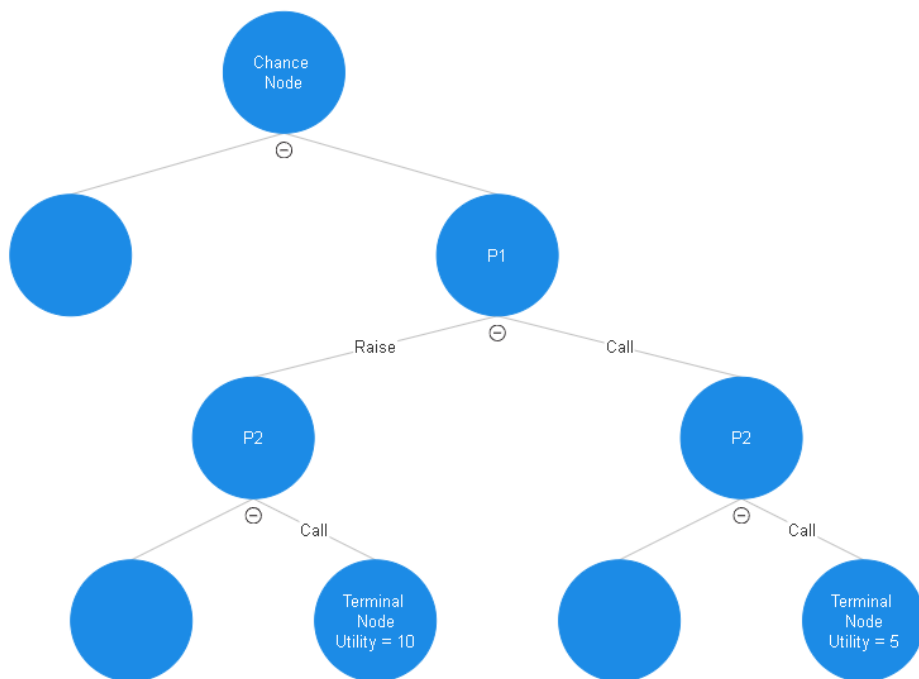


Figure 5- External Sampling Explored

In Figure 5, Player 1 regrets not raising by +5 since the raising action leads to the terminal node that contains +10 utility, whereas the call action leads to the terminal node that contains only +5 utility.

2.4.3. CFR +

CFR + was introduced by a group of researchers in 2015. It improves convergence speed by altering regret updates. This is done by not allowing negative cumulative regret to decrease strategy probability. This differs from vanilla CFR since in vanilla CFR, if an action's regret becomes negative, that action is effectively suppressed until the regret slowly climbs back into positive regret. CFR+ avoids this delay by resetting negative regrets to 0. This allows actions that appeared suboptimal to no longer be "held back" by large negative regrets and can re-enter the strategy as soon as they accumulate positive regret. Each iteration goes over the entire game tree like Vanilla CFR. **(Bowling, Burch, Johanson, & Tammelin)**

2.4.4. Deep CFR

Deep CFR has been the latest development of CFR and was introduced in 2018 **(Brown, Lerer, Gross, & Sandholm)**. Whereas other CFR methods are tabular methods (storing a table of regrets/ strategies for every information set). Deep CFR replaces the large tables with neural network function approximators. The core idea is to approximate the mapping from game states to action regrets and strategies using neural networks, instead of storing exact values for every state. Since Deep CFR does not maintain explicit tables of regret for every information set it is much more memory efficient.

2.4.5. Rule-Based Approach

Rule-Based Approach was used for most early poker bots. This approach is characterized by using hand-crafted rules and expert heuristics to make decisions. They usually use hand strength (private card + community cards strength) and fixed strategies. University of Alberta's first poker program, Loki, **(Billings et al., 1998)** is a prime example of a Rule-based approach bot. Rule-based approaches have, for the most part, been abandoned by researchers. This is because, as researchers noted, it is "impossible to write rules for all the scenarios that can occur" **(Billings et al., 1998)** and any static rule set will fail to cover all dynamic situations in poker. Furthermore, maintaining a static strategy invites exploitative play for opponents who detect the bot's predictable pattern.

2.4.6. Simulation-based Approach

Simulation-based approaches, addresses hidden information in poker. Unlike deterministic tree-search algorithms which are difficult to implement because of hidden information, simulation-based approaches estimate the quality of a decision by simulation numerous potential future states of the game, then averaging the results to approximate expected values for each action. **(Billings, Davidson, Schaeffer, & Szafrom, 2002)**

Simulation based approaches provide many benefits as they are able to deal with hidden information and probabilistic elements that are inherent in poker. However, simulation-based approaches are infeasible for the whole game of poker, especially during early-stages of the game whereby there are many unknowns. Since poker simply has too many nodes, simulations are usually left for the final rounds of poker to enhance the agents strategy.

2.5. Poker Bots Overview

Table 1 shows some of famous poker bots throughout history and the approach used. The table below is not an exhaustive list of poker bots, simply some of the more popular bots.

Bot name	Year developed	Poker variant	Method used	Comments	References
Loki	1997	Limit Texas Hold'em	Rule-Based, +Simulation approach	University of Alberta's first poker bot, changes strategy based on opponent modelling	(Billings, Papp, Schaeffer, & Szafron, 1998)
Poki	1999	Limit Texas Hold'em	Rule-Based + Simulation based	Improvement of Loki	(Billings, Davidson, Schaeffer, & Szafron, 2002)
SparBot /PsOpti	2003	Limit Texas Hold'em	Game Theory Optimal (Linear programming)	SparBot abstracted the game state so from $O(10^{18})$ to $O(10^7)$ and then linear programs finds nash equilibrium	(Billings, et al., 2003)
Hyperborean	2006-2011	Limit, no-limit, 3 - player variant	GTO (equilibrium finding), later CFR	Adapted later for other variants of poker.	(Zinkevich, Johanson, Bowling, & Piccione, 2007)
Slumbot	2011-2019	All no-limit Texas hold'em poker variants	CFR+ and MCCFR	Independent research bot. Openly available to play against at https://www.slumbot.com/	(Jackson, 2011)
Cepheus	2015	Heads up limit Hold'em	CFR+	Weakly solves heads-up limit poker	(Bowling, Burch, Johanson, & Tammelin, 2015)
Libratus	2017	Heads up no-limit Hold'em	MCCFR and real time search	Uses CFR as a blueprint, uses real time search in key positions where real time search is more beneficial. Real time updates to blueprint strategy	(Brown & Sandholm, 2017)
Deepstack	2017	Heads up no-limit Hold'em	CFR with search and neural networks	No abstractions are used	(Moravcik, et al., 2017)
Pluribus	2019	6 player no-limit Hold'em	MCCFR, Linear CFR and depth limited search	Played against professionals and produces superhuman strategies	(Brown & Sandholm, 2019)

3. Design and Implementation of CFRv1

3.1 Overview

CFRv1 uses the MCCFR strategy for the first 2 betting phases (up until the flop). For the last 2 betting phases it uses Monte-Carlo Simulations. This was chosen because CFR algorithms provide a near optimal playing style when converged. However, converging beyond the flop is not attainable with limited resources, even with aggressive abstractions that CFRv1 uses. For the rest of the betting phases, Monte-Carlo Simulations is used. Monte-Carlo simulations are unattainable for the whole game since it requires an immense amount of tree searching. Monte-Carlo simulations are not game optimal strategies and thus are exploitable in the long run unlike nash equilibrium approximating strategies (like CFR algorithms). Therefore, dividing the game into 2 sections for the 2 different strategies may provide a competitive advantage for a computationally efficient poker agent. Once MCCFR training is completed, a CSV file is created with strategies for the first 2 betting rounds. Monte-Carlo simulations need only explore the final two betting rounds, thereby reducing the search's branching factor and retrieving a strategy quickly.

Figure 6 shows a high-level pipeline of how my agent was created:

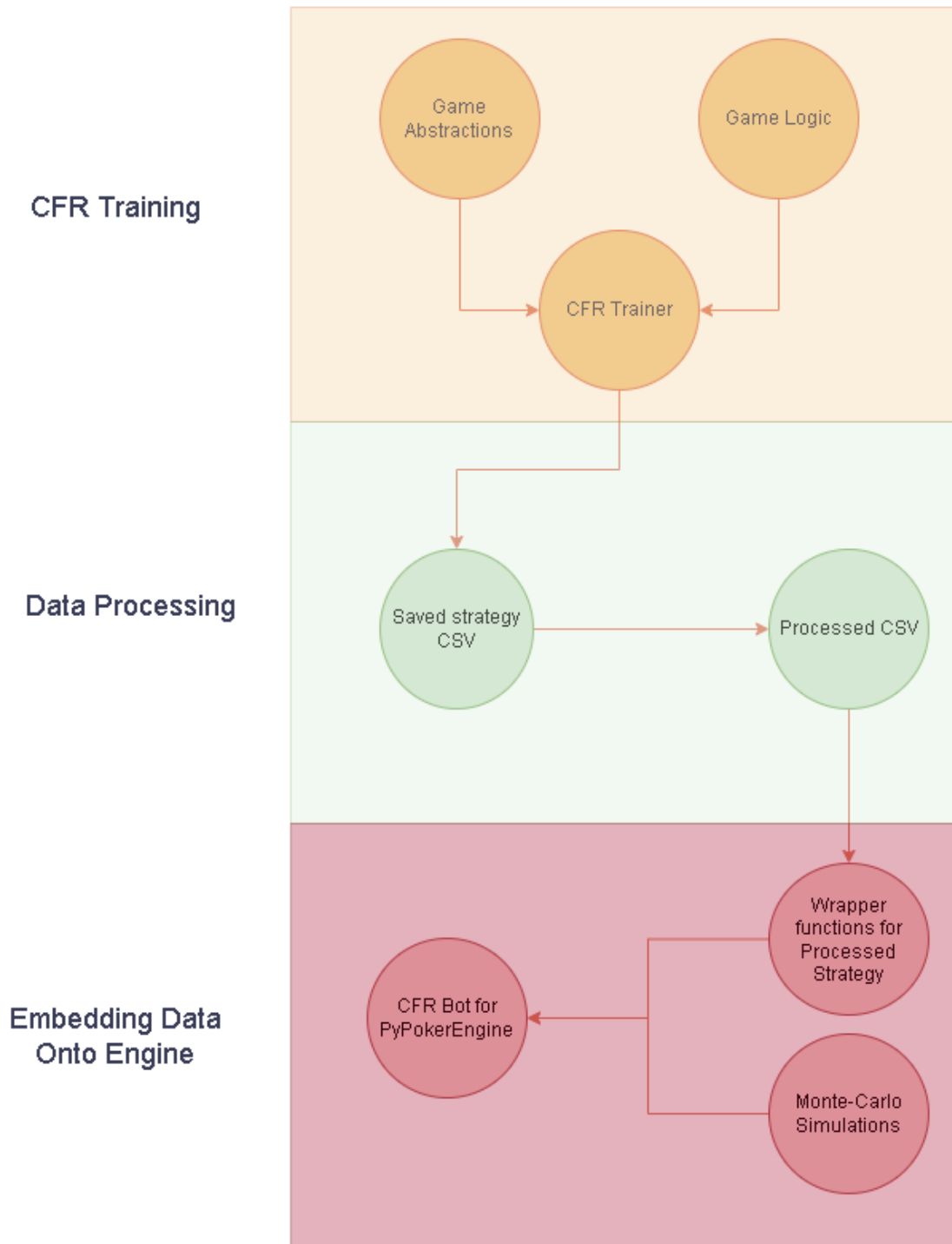


Figure 6 - CFR to PyPokerEngine Pipeline

3.2. MCCFR Training

My model CFRv1 was built with the pseudocode from “**An Introduction to Counterfactual Regret Minimization**” (Neller & Lanctot, 2013). Pseudocode is presented in Appendix A.

CFRv1 uses a specific type of CFR called Monte-Carlo CFR (MCCFR) which utilises chance sampling (see section 2.4.2 for more details). MCCFR was chosen because other CFR’s that

require full tree transversal (like vanilla CFR) can be difficult to debug in practice. Since some CFRs require full tree transversal of a poker game, even with abstractions, such a process can last hours for a single iteration. On the other hand, transversal of an iteration through MCCFR will reach 200 iterations per second with action abstractions, although it may need more iterations to reach nash equilibrium. It can be stopped and analysed to see if it is trending towards nash equilibrium easier. Moreover, MCCFR has been empirically shown to converge faster (Lanctot, Waugh, Zinkevich, & Bowling, 2009).

CFRv1 was trained for 10 million iterations of MCCFR, as this number provided a good balance between convergence toward equilibrium and computational efficiency. Training lasted approximately 11 hours, resulting in a final iteration game value of 0.025, which indicates minimal remaining regret and suggests proximity to Nash Equilibrium. This approach effectively strikes a balance between both aim one and two – beating baseline and research bots and aim four – ensuring computational lightness. Graph showing convergence is shown in Appendix B – figure 17.

Training was conducted using a system with 8GB of memory, and the produced CSV file was approximately 200 MB. These outcomes satisfy the fourth aim, and the computational requirements are lower than those reported in existing literature. (Brown & Sandholm, 2017; Moravcik, et al., 2017)

3.2.1. MCCFR code

To handle the information sets, a class named Node is created.

```

14 class Node:
15     def __init__(self, information_set):
16         self.information_set = information_set
17         self.regret_sum = np.zeros(NUM_ACTIONS, dtype=np.float32)
18         self.strategy_sum = np.zeros(NUM_ACTIONS, dtype=np.float32)
19
20     def get_strategy(self, reach_prob):
21         # Regret-matching
22         strategy = np.maximum(self.regret_sum, 0)
23         norm_sum = np.sum(strategy)
24         if norm_sum > 0:
25             strategy /= norm_sum
26         else:
27             strategy = np.ones(NUM_ACTIONS) / NUM_ACTIONS
28
29         self.strategy_sum += reach_prob * strategy
30         return strategy
31
32     def get_average_strategy(self):
33         norm_sum = np.sum(self.strategy_sum)
34         if norm_sum > 0:
35             return self.strategy_sum / norm_sum
36         return np.ones(NUM_ACTIONS) / NUM_ACTIONS
37

```

Figure 7 - Class Node

The Class Node stores the regret and strategy data for each information set.

The get_strategy function is used in CFR to compute the strategy using regret matching. If there has been no strategy, or the regret is negative, a uniform strategy is used.

The `get_average_strategy` function is responsible for computing the average strategy across all iterations, which converges to Nash equilibrium. This function is what is later used for the CFR bot.

The class `CFRTrainer` encompasses the inner workings of the training process and handles its output.

```

1 class CFRTrainer:
2     def __init__(self):
3         self.nodes = defaultdict(lambda: None)
4
5     def train(self, iterations, filename="cfr_training.csv"):
6         util = 0.0
7         total_steps = iterations * len(PLAYERS)
8         with tqdm(total=total_steps, desc="Training CFR") as pbar:
9             for _ in range(iterations):
10                 for player in PLAYERS:
11                     state = GameState.new_hand()
12                     util += self.cfr(state, player, 1, 1)
13                     pbar.update(1)
14
15             print(f"Average game value: {util / (iterations * len(PLAYERS))}")
16             with open(filename, 'w', newline='') as f:
17                 writer = csv.writer(f)
18                 writer.writerow(["info_set", "strategies"])
19                 for information_set, node in self.nodes.items():
20                     if node:
21                         strat = node.get_average_strategy()
22                         writer.writerow([information_set, f"[{','.join(f'{s:.2f}' for s in strat)}]"])
23
24     def cfr(self, state, traversing_player,  $\pi_1$ ,  $\pi_2$ ):
25         if state.is_terminal():
26             return state.get_showdown_value(traversing_player)

```

Figure 8 - CFRTrainer

The `train` function is responsible for beginning the training process and sets the number of iterations for training. This is where the program calls for the start of a poker game using Game state which contains all poker-specific logic. It also initiates the traversing player to see which player's regret is being updated. Then, the function calls the `average_strategy` function from earlier, stores the average strategy for each information set and constructs a CSV file to be processed later in the pipeline. It also stores the average game value to see how close it is to converging. An average game value close to 0 would mean that the strategy the players adopt would have minimal regret therefore are close to an optimal solution (reached nash equilibrium).

After calling a new game with pre-sampled hole cards and community cards, one of the players transverse through the game tree through the `CFR` function.

```

def cfr(self, state, traversing_player,  $\pi_1$ ,  $\pi_2$ ):
    if state.is_terminal():
        return state.get_showdown_value(traversing_player)

    if state.is_chance_node():
        sampled_state = state.sample_chance_outcome()
        return self.cfr(sampled_state, traversing_player,  $\pi_1$ ,  $\pi_2$ )

    acting_player = state.acting_player
    info_set = state.get_info_set()
    node = self.nodes.get(info_set)
    if node is None:
        node = Node(info_set)
        self.nodes[info_set] = node

    strategy = node.get_strategy( $\pi_1$  if acting_player == 1 else  $\pi_2$ )
    action_utils = np.zeros(NUM_ACTIONS)
    node_util = 0

    for action in range(NUM_ACTIONS):
        if state.is_invalid_raise(action):
            continue

```

Figure 9 - CFR Part 1

```

    for action in range(NUM_ACTIONS):
        if state.is_invalid_raise(action):
            continue

        next_state = state.apply_action(action)

        if acting_player == 1:
            action_utils[action] = self.cfr(next_state, traversing_player,  $\pi_1$  * strategy[action],  $\pi_2$ )
        else:
            action_utils[action] = self.cfr(next_state, traversing_player,  $\pi_1$ ,  $\pi_2$  * strategy[action])

        node_util += strategy[action] * action_utils[action]

    if acting_player == traversing_player:
         $\pi_{\text{opponent}}$  =  $\pi_2$  if acting_player == 1 else  $\pi_1$ 
        for a in range(NUM_ACTIONS):
            regret =  $\pi_{\text{opponent}}$  * (action_utils[a] - node_util)
            node.regret_sum[a] += regret

    return node_util

if __name__ == "__main__":
    trainer = CFRTrainer()
    trainer.train(iterations=100000)

```

Figure 10 - CFR Part 2

During each traversal of the game tree, the CFR algorithm performs the following steps:

1. **Information set creation:** If the current decision node corresponds to an information set that has not yet been encountered, a new information set is created.

2. **Chance node sampling:** If the node is a chance node (dealing a community card), a single outcome is sampled, and the algorithm proceeds along that branch.
3. **Terminal node evaluation:** If the node is terminal, the payoffs for each player are computed and returned for backpropagation.
4. **Recursive decision-node exploration:** Otherwise, for a decision node belonging to player i , CFR iterates over all actions and recursively calls itself on each successor node and accumulates the counterfactual value of each action.
5. **Regret updating:** After exploring all children of an information set that belongs to the traversing player, CFR computes the regret for each action as the difference between its counterfactual value and the information set's average counterfactual value. These regrets are then used to update the strategy probabilities for later iterations.

3.2.2. Action Abstraction Implemented

For the concept of action abstraction, refer to section 2.2.2.

CFRv's1 action abstraction was aggressive. CFRv1 had only 3 betting options for the first two betting rounds: fold, call and raise, making it synonymous with limit poker. This was chosen to drastically reduce the branching factor. Limiting decisions to only 3 actions, sacrifices some realism and strategic nuance, however, may still be capable of producing a strong strategy.

Furthermore, there was a limit of 3 raises per round per player to further limit the branching factor. This abstraction should have a minimal impact on the performance of the bot because it is rare to see more than 3 raises per round per player. This is because at the point where there would be a multitude of raises, the minimum bet would increase to an amount close to stack size. Moreover, without specifying limiting raises per round per player, the algorithm never ends, and no strategy would be outputted.

An important note for any abstraction is that CFR algorithms will reach nash equilibrium only for the abstracted game, how close the nash equilibrium is to the unabridged version of the game is dependent on the quality of the abstraction. Less abstraction produces higher quality strategy but sacrifices computation resources.

3.2.3. Information Abstraction Implementation

For the concept of action abstraction, refer to section 2.2.2.

The implementation for Information Abstraction for CFRv1 was removing suits from community cards and for private cards categorizing them between suited and off suited. For the first betting round, grouping private cards if they are suited or off suited (AKo or AKs) is commonplace in CFR training abstractions. This is because in preflop the strategy between a pair of cards that are both off suit are the exact same, therefore both do not need to be processed. However, if this categorisation is furthered onto the flop, it saves considerable computation but makes the abstraction coarser.

Additionally, CFRv1 removes suits from community cards. This abstraction is more aggressive than most abstractions, but limits information sets greatly. Moreover, the chance of having a ranking of a flush (5 cards with the same suit) with any starting private card combination is only 0.2% in second betting round (**888poker, n.d.**). Thus, the impact of

disregarding suits for preflop and flop should not have a significant impact on the strength of the strategy. However, flush potential cards strategy will be undervalued using this abstraction.

3.2.4. Limiting rounds

Another important abstraction used was limiting MCCFR strategy to the second betting round. This means that it only creates strategy until the flop, which means that MCCFR is not used for a whole game of poker, instead just the first two betting phases, greatly speeding up convergence. This is because the reduction of chance nodes is significant ($48C5 = \sim 2$ million combinations of community cards down to $48C3 = \sim 17,000$). Not having as many rounds also makes memory usage more manageable, as memory can exceed my computers hardware.

Limiting MCCFR to the second betting round, suit-less abstraction and three legal actions produced ~ 5 million information. Iteration speed was roughly tenfold faster for every betting round deducted.

3.3. Data Processing

Once Training is finished, the function `train` inside `CFRTrainer` creates a CSV file that contains all the information sets visited. However, this data includes many information sets that are rarely visited and therefore have not updated their strategy (\sim two million information sets). This is where the script, `Cleaning_default_strategies.py`, is ran. The script reads through all five million information sets and removes any information set which still has the uniform strategy (0.33,0.33,0.33). This is required because of two reasons:

1. Uniform Strategy is rarely the best strategy so even if these unlikely information sets are reached in a real game, the bot can use Monte-Carlo Simulations instead.
2. Information sets are cut down from five million to two million, making it faster for searching later for when playing poker games in real time. (See appendix C for information set reduction)

3.4. Monte-Carlo Simulations

Monte-Carlo Simulations are used when there is no trained information set for the gameplay information set. This will mostly occur during the third and fourth betting round because MCCFR is not trained in these rounds.

The implementation of Monte-Carlo Simulations used is from a publicly available git repository. (**Drumsta, 2020**)

The function `montecarlo_simulation` retrieves game state variables from `PyPokerEngine` (the game engine) like community cards and hole cards. If there are no community cards/missing community cards, `PyPokerEngine` function `_fill_community_card` populates the community cards. From the remaining cards, the opponent's hole cards are also derived. The simulation evaluates the bots hand ranking against the opponent's hand ranking and calculates the winner.


```
def montecarlo_simulation(nb_player, hole_card, community_card):
    # Do a Monte Carlo simulation given the current state of the game by evaluating the hands
    community_card = _fill_community_card(community_card, used_card=hole_card + community_card)
    unused_cards = _pick_unused_card((nb_player - 1) * 2, hole_card + community_card)
    opponents_hole = [unused_cards[2 * i:2 * i + 2] for i in range(nb_player - 1)]
    opponents_score = [HandEvaluator.eval_hand(hole, community_card) for hole in opponents_hole]
    my_score = HandEvaluator.eval_hand(hole_card, community_card)
    return 1 if my_score >= max(opponents_score) else 0
```

Figure 11 - Monte-Carlo Simulation Code

A single Monte-Carlo simulation is not statistically significant because the combination of potential hands the opponent has, and community cards are quite large; many simulations are required. The bot's wins are aggregated across all simulations and then the win rate is derived from the number of wins divided by how many simulations have occurred.

```
# Estimate the ratio of winning games given the current state of the game
def estimate_win_rate(nb_simulation, nb_player, hole_card, community_card=None):
    if not community_card: community_card = []

    # Make lists of Card objects out of the list of cards
    community_card = gen_cards(community_card)
    hole_card = gen_cards(hole_card)

    # Estimate the win count by doing a Monte Carlo simulation
    win_count = sum([montecarlo_simulation(nb_player, hole_card, community_card) for _ in range(nb_simulation)])
    return 1.0 * win_count / nb_simulation
```

Figure 12 - Monte-Carlo Win Rate

The code includes thresholds for each action and the minimum win rate for each action to be used at each round. If the win rate does not exceed these thresholds, then the bot folds. It is important to note that even though preflop and flop thresholds are available, they are rarely used because the blueprint MCCFR strategy oversees strategy for those rounds.

```
vChanceCall = {'preflop': 0.13, 'flop': 0.3, 'turn': 0.4, 'river':0.7}
vChanceRaise_x1 = {'preflop': 0.17, 'flop': 0.45, 'turn': 0.47, 'river':0.8}
vChanceRaise_x2 = {'preflop': 0.20, 'flop': 0.55, 'turn': 0.47, 'river':0.9}
def declare_action(self, valid_actions, hole_card, round_state):
    # Estimate the win rate
    # HAS GOT NUMBER OF SIMULATIONS
    win_rate = estimate_win_rate(100, self.players_remaining, hole_card, round_state['community_card'])

    if (win_rate > self.vChanceRaise_x2[round_state['street']]):
        return self.ActRaise_x2(valid_actions)
    if (win_rate > self.vChanceRaise_x1[round_state['street']]):
        return self.ActRaise_x1(valid_actions)
    elif (win_rate > self.vChanceCall[round_state['street']]):
        return self.ActCall(valid_actions)
    else:
        return self.ActPass(valid_actions)
```

Figure 13 – Monte-Carlo Thresholds

3.5 PyPokerEngine Environment

To integrate the blueprint MCCFR strategy stored in the CSV file into PyPokerEngine, the system must find the information set from the CSV file that matches the information set currently being played. For example, in the first betting round, the information set could look

like “AA c” so the agent searches for this information set in CSV and retrieves it alongside its strategy.

The bot’s decision-making routine is activated during its turn. PyPokerEngine provides essential game state variables that should be visible to the bot like hole cards and community cards. PyPokerEngine does not natively support information sets. Thus, information sets must be built and homogenised to the trained information sets in the CSV file. The function `get_information_set_engine` builds the information set based on the current gameplay and the function. `information_set_to_abstracted_set` applies the same abstraction used during training to the gameplay information set so that the gameplay information sets are alike to the trained information sets in the CSV file.

Once gameplay information sets and trained information sets are alike, the corresponding trained information set is searched and retrieved, `get_strategy` oversees this behaviour. The bot then probabilistically chooses an action based on the action probabilities from the information set using the `select_action_from_strategy` function.

Finally, the chosen action is translated into PyPokerEngine’s operational syntax. For example, the action raise needs to be accompanied by an amount (or specified if it’s a minimum or maximum raise). The function `action_to_engine` handles this conversion and ensures compliance with PyPokerEngine’s rules.

```
def declare_action(self, valid_actions, hole_card, round_state):  
    # BLUEPRINT  
    community_cards = round_state["community_card"]  
    #get the raw information set from engine  
    raw_information_set = CFREngine.get_information_set_engine(round_state, hole_card, community_cards)  
    #abstract it using the abstraction used in training  
    abstracted_information_set = CFREngine.information_set_to_abstracted_set(raw_information_set)  
    #get the strategy looks like [0.9,0.1,0.1]  
    strategy = CFREngine.get_strategy(df,abstracted_information_set)  
    #select the specific action  
    selected_action = CFREngine.select_action_from_strategy(strategy)  
    #print(selected_action)  
  
    if selected_action == "not blueprint":
```

Figure 14 - PyPokerEngine Bot

3.6. APIs and Tools

3.6.1. PyPokerEngine

PyPokerEngine library was chosen for this project because it provides a solid framework for poker and poker simulation. It offers useful functions that facilitate bot development and already possesses baseline bots that can be used for testing. (Ishikota, n.d.)

Additionally, PyPokerEngine is widely recognized among poker bot developers, which provides access to a community of hobbyists who create bots that can be used for benchmarking. The Monte-Carlo Bot used later in the results section is an example of a bot created for the PyPokerEngine environment.

3.6.2. Faster Hand Evaluator

Initially, the native hand evaluator provided by PyPokerEngine was employed during MCCFR training. However, preliminary analysis showed that this evaluator took up a significant portion of training time. Consequently, a more efficient alternative hand evaluator was found from a Git repository, developed by **Lee (2025)**, significantly cutting down MCCFR training time.

4. Experiments

This project evaluates two bots, CFRv1 and CFRv2 against two categories of opponents: baseline bots provided by PyPokerEngine and research-oriented bots. The difference between CFRv1 and CFRv2 is the number of Monte-Carlo simulations used. CFRv1 uses only 50 simulations per decision, whereas CFRv2 uses 10,000 simulations. The intent of testing two bots was to assess the impact of Monte-Carlo simulations component of the bot's strategy. Furthermore, 50 simulations provide minimal statistical accuracy for win-rate estimation, whereas CFRv2's 10,000 simulations per decision should give more precise outcomes.

However, Monte-Carlo simulations impact response time negatively, which counteracts aim number three – Achieve human-comparable response times. To adhere to this objective Monte-Carlo simulations were capped at 10,000 for CFRv2. Increasing the number of simulations should improve competitive performance. All games were conducted using PyPokerEngine library

4.1 Experimental Setup

- Baselines bots are:
 - Fish – always calls to every situation
 - Honest – has an inbuilt estimate hand strength from PyPokerEngine. Runs lots of simulations and uses average win rate as estimation
 - Random Action Model – randomly selects either fold, call or raise
- Research bots:
 - Rule-based Model – a rule-based bot I created that utilises common poker heuristics. Could be described as an amateur poker player. Some heuristics used are value betting and only playing stronger starting hands.
 - Monte-Carlo Model – runs Monte-carlo simulations, estimates its win rate and plays accordingly, uses the same code as for CFRv1 and CFRv2 but for the whole game not just the last two rounds
- Parameters:
 - All bots versed each other for 100,000 games in PyPokerEngine
 - Starting Stacks are 1000 for each bot, small blind is 5 and big blind is 10
 - CFRv1 was trained on 10 million MCCFR iterations and 50 Monte-Carlo Simulations.
 - CFRv2 was trained on 10 million MCCFR iterations and uses 10,000 Monte-Carlo Simulations
 - Monte-Carlo bot uses 10,000 Monte-Carlo Simulations

4.2. Challenges and Limitations

4.2.1. Computational Cost of MCCFR Training

MCCFR training for Poker is computationally demanding. One aspect of this is memory usage. Due to the millions of information sets and their associated variables generated during training; memory usage can quickly exceed an average person's computer hardware. Early training iterations without round limiting (on a full game of poker) quickly surpassed 8GB of RAM and crashed the system.

Optimisation for limiting memory usage was necessary. One effective approach of this optimisation was to reduce memory requirements via the python library numpy. Numpy allows altering data types of arrays, so information set variables were changed from default float64 to float32, resulting in a 50% reduction in memory consumption per value.

```
class Node:
    def __init__(self, information_set):
        self.information_set = information_set
        self.regret_sum = np.zeros(NUM_ACTIONS, dtype=np.float32)
        self.strategy_sum = np.zeros(NUM_ACTIONS, dtype=np.float32)
```

Figure 15 - Reducing Memory Usage

Furthermore, millions of iterations of MCCFR incurs considerable computational time, regardless of the game abstraction used. A couple of potential ways to speed up training is multiprocessing and utilizing GPUs. Further elaboration on optimization techniques is discussed in Section 7.1. Computational Improvements.

4.2.2. Lack of Accessible Research Source Code

The scarcity of publicly available source code for advanced poker research bots was a significant limitation for evaluating my bot. Due to minimal standardization for CFR algorithms for poker, the MCCFR section of my bot was implemented mostly from scratch, including all underlying game logic.

Although limited availability in open-source implementations of advanced poker bots is understandable, mainly due to ethical considerations associated with using bots against human opponents with real money. Some research groups, such as the University of Alberta's Computer Research Group, provide some implementations of CFR algorithms. However, these were coded in C and C++, a programming language I have limited experienced in and is incompatible with my Poker Environment. Furthermore, creating wrappers compatible with my Poker Environment would have introduced substantial complexity that would not necessarily achieve my project aims.

4.2.3. Time Taken for Testing

Although the time performance of both CFRv1 and CFRv2 were kept in line with aim number three - Achieve human-comparable response times, testing still required significant time. To minimise variance effects, a total of 100,000 games were played. Considering an average duration of one second per game, depending upon the participating bots, reaching 100,000 games takes 27 hours to complete. While baseline bots typically completed

individual games in less than one second, CFRv1, CFRv2, Rule-Based Model and Monte-Carlo Model often exceeded 1 second per game. Although 100,000 games is a robust sample size for poker, increasing the sample to one million games could give even more statistically reliable outcomes. (Billings, et al., 2003)

5. Results

The performance of the CFRv1 and CFRv2 is now measured and analysed. They will be compared against baseline and research agents. It is important to note that a “hand” in this context is the same as a poker game

5.1 Experiment’s Measure

- **Small Blind per hand (sb/h):** This type of measurement of performance is commonplace in both research and for recreational players analysing their profit (Chiswick M. , n.d.). A winning player would have a positive sb/h, and a losing player would have a negative sb/hand. It is important to note that research papers use different variations of this equation to measure performance. Common differences are big blind per hand (bb/h), which uses the value of the big blind (2x the value of small blind) instead of the small blind. Additionally, milli big blind or small blinds are sometimes used.

$$\frac{sb}{h} = \frac{chips\ earned}{small\ blind * number\ of\ hands}$$

5.2 Heads-up Games

Table 2 shows the result all the models against each other, all values are in sb/h:

	Fish	Honest	Random action Model	Rule-based Model	Monte- Carlo Model	CFRv1	CFRv2
Fish	0	+0.81	+68.65	-6.08	-2.00	-1.56	-1.56
Honest	-0.81	0	+16.04	-0.48	-1.99	-0.82	-0.81
Random action Model	-68.65	-16.04	0	+1.00	-44.31	-5.95	-11.51
Rule- Based Model	+6.08	+0.48	-1.00	0	-2.00	-0.06	-0.09
Monte- Carlo Model	+2.00	+1.99	+44.31	+2.00	0	+2.75	-1.75
CFRv1	+1.56	+0.82	+5.95	+0.06	-2.75	0	-2.13
CFRv2	+1.56	+0.81	+11.51	+0.09	-1.75	+2.13	0

Table 2 – Results

5.3. Evaluating Performance

CFRv1's and CFRv2's results align with GTO strategy. They win against all baseline opponents after 100,000 games by a large margin. Some baseline opponents outperform CFR in beating a certain opponent. This is concurrent with GTO strategy because it does not aim to maximise profit, it simply aims not to lose against an opponent, in the long run, and waits for the opponent to make a mistake to win. Additionally, the exploitative part of CFRv1, the Monte-Carlo Simulation is only 50 simulations therefore the win rate it calculates is inaccurate thus its exploitation of the opponent is minimal. On the other hand, CFRv2's Monte-Carlo Simulation is more prominent and thus achieves better results against some baseline opponents, notably Rule-Based Model. Furthermore, the loss against the Monte-Carlo Model is significant better by +1.00sb/h

CFRv1/2 are winning against Rule-Based. Taking CFRv1's result, it may seem small (+0.06 sb/h), however, across 100 games this is an average gain of 6 small blinds. For a real-life poker playing, $6\text{sb}/100 = 3\text{bb}/100$ is a solid win rate. (Walker, n.d.)

CFRv2 beats CFRv1 a large margin of +2.13sb/h, showing that the impact of Monte-Carlo Simulations is important to the model.

CFRv1/2's performance against Monte-Carlo Model shows that Monte-Carlo Model managed to exploit both by a significant margin. Monte-Carlo search seems to outperform both versions of CFR. This result is understandable against CFRv1 since CFRv1 only uses 50 simulations for its Monte-Carlo component, therefore its search is very primitive.

However, it is surprising that CFRv2 underperforms against the Monte-Carlo Model despite both using the same number of simulations. One explanation is that the abstraction used during training was too coarse, allowing the bots to be exploited in an unabstracted poker game. Additionally, the model appeared to be nearly converged, but running more iterations could have driven it closer to equilibrium to produce a stronger strategy.

All games took less than 30 seconds, therefore aim two was achieved. Figure 21 in Appendix D shows the average game time taken between CFRv1 and CFRv2.

5.4. Variance and Statistical Significance

Poker is a high-variance game. Variance in poker is made up of several reasons, like playing styles of players and the large number of chance elements. This makes it very difficult to calculate the variance, so I set the sample size to 100,000 to minimise the luck factor in the game. This aligns with research methods as seen in **(Billings, et al., 2003)** whereby many bots versed against each other for at least 20,000 games whilst some matches were up to 100,000. In this paper the variance for each match differs depending on the styles of the two players involved but it typically is +/- 6sb and the standard deviation for each match is normally +/- 0.03 sb/h.

6. Conclusion

In conclusion, my Poker bot serves as a useful framework for a lightweight yet competitive Heads-Up No-Limit Texas Hold'em poker AI. The implemented bot successfully beat all baseline opponents, fulfilling aim one.

Furthermore, both CFRv1 and CFRv2 demonstrate an average response time considerably below the 30-second threshold, satisfying the third aim – response time for decisions to be comparable to human poker players.

Additionally, the bot achieved the fourth aim of being computationally lightweight, with MCCFR training requiring approximately 11 hours to approach a near Nash equilibrium, which is an improvement compared to other implementation of CFR algorithms in existing literature. **(Brown & Sandholm, 2017; Moravcik, et al., 2017)**. Moreover, MCCFR training was computed in a computer with only 8gb of memory and the unprocessed data from training is approximately 200mb in size.

However, further developments are required to fulfill aim number two, which involves consistently outperforming research bots. The bot did not surpass the Monte-Carlo bot, and evaluation against state-of-the-art bots was not conducted due to constraints outlined in Section 7.2.1

Overall, this project fulfilled three out of four of its aims, demonstrating that the MCCFR algorithm with Monte-Carlo simulations is a viable method for an efficient intermediate poker AI and with more development could potentially achieve superhuman results.

7. Future Work

7.1. Computational Improvements

Performance could have been optimized further for both training and evaluation phases. While integration of a faster hand evaluator produced significant gains, abstraction to limit MCCFR strategy up until the flop affected strategy negatively, since relying on the fly Monte-Carlo simulations rather than the MCCFR policy makes the bot more exploitable. Future implementations could expand MCCFR into the third betting round, however, computational improvements need to be implemented for this. A few ideas are:

- **More efficient Data retrieval and storage methods** for pre-trained MCCFR data; although my bot currently does not have a need for this optimisation, if expanding pre-trained data onto the third round is to be done, the increase in millions of information sets are unlikely to be handled speedily without better data storage and data retrieval methods.
- Moreover, **parallelisation** for both MCCFR training and gameplay simulations for testing could be implemented. On a normal multi-core system, this optimisation is between a speed up of around 4- 8x depending on the number of cores. This is because python only utilizes one core, thus other cores are idle during training and simulations. Although I did attempt to implement parallelisation for MCCFR training, my implementations failed and the speed up of 4x was not necessary since training only had to be ran once to reach near equilibrium.
- Alternatively, **GPU acceleration** could be explored for vectorising key operations like regret-matching. However, during training most time taken was dominated through non-vectorizable game-logic functions, suggesting that this implementation may not provide meaningful gains.

7.2. Competing against Research Bots and Human Opponents

7.2.1 Research Bots

Most good research bots remain inaccessible for ethical reasons. This hinders direct comparison against the most advanced poker agents. Additionally, many research bots operate in their own environments, making development of compatible interfaces a significant challenge.

A bot that should compete against CFRv1/2 for future works is Slumbot 2017. Slumbot 2017 is a research bot that has an API to be played against however, since CFRv1 was built specifically for PyPokerEngine, adapting it to compete against Slumbot would require substantial changes to the bot.

7.2.2 Human Opponents

Assessing performance against human opponents of varying skill levels introduces two complications: lack of sample size and statistical significance of results.

- **Lack of Sample Size:** Finding poker players of different skill levels who are willing to partake in a free study. Already we're selecting only a small portion of the population to those who know how to play poker. Moreover, evaluating their skill level is difficult since poker does not have a standardised rating system like chess and a winning player in a specific table is not necessarily a winning table in another.
- **Statistical Significance of Results:** The number of poker games required for statistical significance is high. The normal speed for 6 player poker online is around 75-100 games per hour (**Upswing poker, 2020**), Heads up poker will likely be faster than 6 player poker however not significantly faster. Even if we grant that there will be 500 games per hour, for just 10,000 games it will take each human player roughly 20 hours. This is a big ask for most people and 10,000 games is still not very statistically significant

Although the bot could be programmed to run on popular poker applications to play against human opponents, this breaks the terms of services of the applications. (**Pokerstarsuk, N/A**)

7.3. Using a more sophisticated Monte-Carlo Simulation

Monte-Carlo Simulations can be refined so that the bot performs better.

For example, the current implementation of Monte-Carlo Simulations does not incorporate pot sizes. By modelling how the pot evolves in different situations, the simulation can accurately estimate equity thresholds for calling, raising and folding

Moreover, Monte-Carlo Simulations could integrate action histories. This is because, certain sequences of actions are indicative of the opponents' private cards. Coupling this with opponent modelling allows the agent to differentiate between the sort of player it is against and act to maximise utility.

Finally, the current implementation of Monte-Carlo Simulations does not have dynamic bet sizing or more betting options. This change would allow a better performance in strategy, allowing the bot to incorporate human- like strategy of over-bets, under-bets and pot-sized bets.

Incorporating any of these solutions would invariably increase the agent's decision-making time, therefore should be incorporated with optimisation in mind.

7.4. Less Abstraction

To achieve a more optimal strategy, less abstraction should be used. For example, the MCCFR section of the bot does not include turn and river because of the added computational requirements. However, if computation was not a problem, less abstraction during training should be used. The quality of some combination of hands and community cards are not realised due to this form of lossy abstraction. For example, connected private cards (e.g. 7, 8) in the current environment fair far worse than in unabridged poker, since unabridged poker has a total of five community cards and the abstracted version only has

three. The current MCCFR therefore undervalues connected cards, making their strategy course.

Moreover, current MCCFR abstraction removes suit information from information sets. Suit information could be reintroduced after the first betting round (when community cards are shown) so that the bot could distinguish between the different strategies relating to flush and flush potentials.

Current action abstraction leaves little strategic depth to the bots' decisions. Adding more raising options, like half of pot, pot, 2x pot and all-in options will enrich the strategy space, allowing for a greater extraction of profit from opponents and thus a greater sb/h.

8. Reflection on Learning

Implementing MCCFR with Monte-Carlo simulations proved itself to be an enriching journey. Although the background reading was dense and highly technical at first, patterns soon emerged across papers such as CFR and its variants along with methods of abstraction. These ideas evolved across papers however the foundations on why they were applicable to this problem stayed constant.

Delving into game theory concepts, like Nash Equilibrium, was a highlight and coding theory into a practical scenario was both a challenging and fulfilling experience.

The project had clear obstacles. Developing a functional MCCFR algorithm presented unique challenges I had not faced before. There was little standardisation for any CFR algorithm or coded examples in literature. Initially many attempts of writing both the MCCFR algorithm and the game logic were unsuccessful however, through continuous attempts at debugging and refreshing my knowledge on the concepts it proved to be successful.

In retrospect, this project has taught me a lot. It expanded my programming skills, introduced me to game theory and improved my report writing skills. More importantly, it forced me to examine my assumptions and decisions at every step of the project.

Appendices

Appendix A: CFR with Chance Sampling Pseudocode

Pseudocode code used for building the CFR portion of the bot:

Algorithm 1 Counterfactual Regret Minimization (with chance sampling)

```
1: Initialize cumulative regret tables:  $\forall I, r_I[a] \leftarrow 0$ .
2: Initialize cumulative strategy tables:  $\forall I, s_I[a] \leftarrow 0$ .
3: Initialize initial profile:  $\sigma^1(I, a) \leftarrow 1/|A(I)|$ 
4:
5: function CFR( $h, i, t, \pi_1, \pi_2$ ):
6:   if  $h$  is terminal then
7:     return  $u_i(h)$ 
8:   else if  $h$  is a chance node then
9:     Sample a single outcome  $a \sim \sigma_c(h, a)$ 
10:    return CFR( $ha, i, t, \pi_1, \pi_2$ )
11:  end if
12: Let  $I$  be the information set containing  $h$ .
13:  $v_\sigma \leftarrow 0$ 
14:  $v_{\sigma_{I \rightarrow a}}[a] \leftarrow 0$  for all  $a \in A(I)$ 
15: for  $a \in A(I)$  do
16:   if  $P(h) = 1$  then
17:      $v_{\sigma_{I \rightarrow a}}[a] \leftarrow \text{CFR}(ha, i, t, \sigma^t(I, a) \cdot \pi_1, \pi_2)$ 
18:   else if  $P(h) = 2$  then
19:      $v_{\sigma_{I \rightarrow a}}[a] \leftarrow \text{CFR}(ha, i, t, \pi_1, \sigma^t(I, a) \cdot \pi_2)$ 
20:   end if
21:    $v_\sigma \leftarrow v_\sigma + \sigma^t(I, a) \cdot v_{\sigma_{I \rightarrow a}}[a]$ 
22: end for
23: if  $P(h) = i$  then
24:   for  $a \in A(I)$  do
25:      $r_I[a] \leftarrow r_I[a] + \pi_{-i} \cdot (v_{\sigma_{I \rightarrow a}}[a] - v_\sigma)$ 
26:      $s_I[a] \leftarrow s_I[a] + \pi_i \cdot \sigma^t(I, a)$ 
27:   end for
28:    $\sigma^{t+1}(I) \leftarrow$  regret-matching values computed using Equation 5 and regret table  $r_I$ 
29: end if
30: return  $v_\sigma$ 
31:
32: function Solve():
33: for  $t = \{1, 2, 3, \dots, T\}$  do
34:   for  $i \in \{1, 2\}$  do
35:     CFR( $\emptyset, i, t, 1, 1$ )
36:   end for
37: end for
```

Figure 16 -MCCFR

Appendix B: MCCFR Training

MCCFR algorithm approaching convergence:

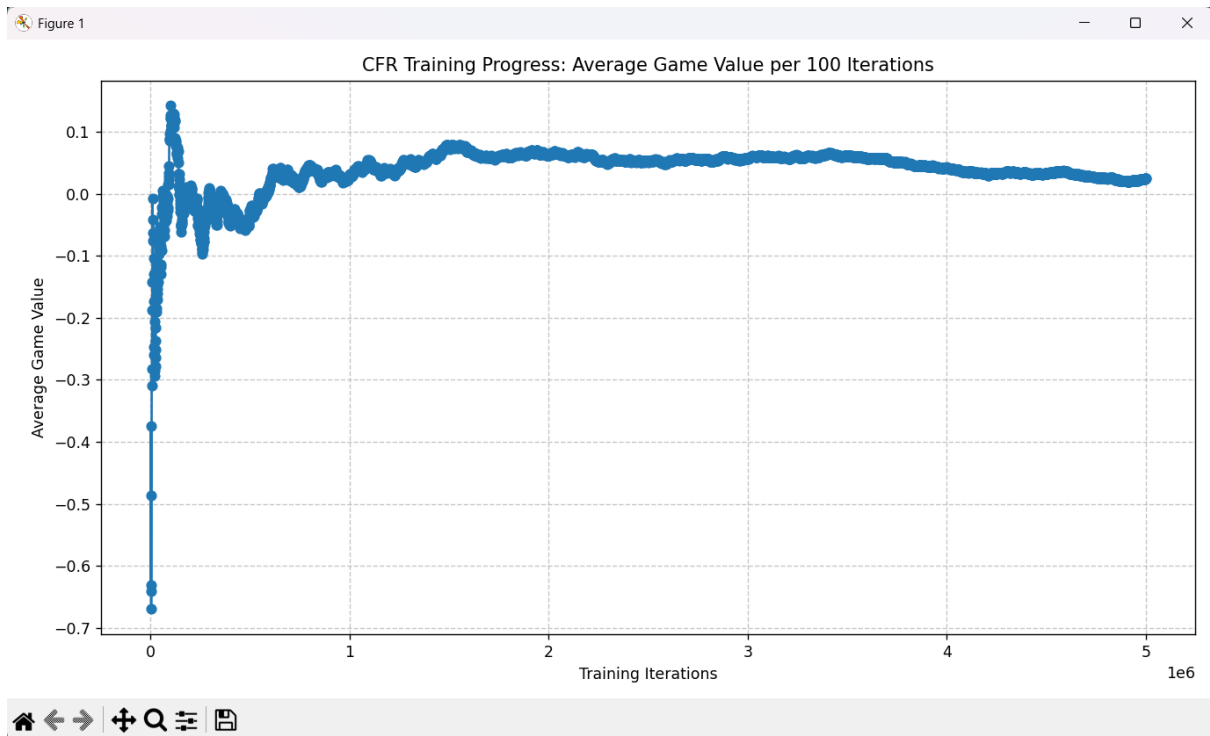


Figure 17 - Average game Value across Iterations

Time taken for training:




Figure 18 - Training- Time Taken

Appendix C: Information Set Reduction

Unprocessed information set CSV contains 5 million entries

```

training_data >  cfr_training.csv
5534167 K7s 555 nc,"[0.33,0.33,0.33]"
5534168 K7s 555 nccr,"[0.33,0.33,0.33]"
5534169 K7s 555 nccrrr,"[0.33,0.33,0.33]"
5534170 K7s 555 nccr,"[0.33,0.33,0.33]"
5534171 K7s 555 nccrrr,"[0.33,0.33,0.33]"
5534172 K7s 555 rccc,"[0.33,0.33,0.33]"
5534173 K7s 555 rccrrr,"[0.33,0.33,0.33]"
5534174 K7s 555 rccrrrrr,"[0.33,0.33,0.33]"
5534175 K7s 555 rccr,"[0.33,0.33,0.33]"
5534176 K7s 555 rccrrr,"[0.33,0.33,0.33]"
5534177 K7s 555 rrrc,"[0.33,0.33,0.33]"
5534178 K7s 555 rrrccr,"[0.33,0.33,0.33]"
5534179 K7s 555 rrrccrrr,"[0.33,0.33,0.33]"
5534180 K7s 555 rrrcrrr,"[0.33,0.33,0.33]"
5534181 K7s 555 rrrccrrrrr,"[0.33,0.33,0.33]"
5534182 K7s 555 rrrrrc,"[0.33,0.33,0.33]"
5534183 K7s 555 rrrrrccr,"[0.33,0.33,0.33]"
5534184 K7s 555 rrrrrccrrrrr,"[0.33,0.33,0.33]"
5534185 K7s 555 rrrrrcr,"[0.33,0.33,0.33]"
5534186 K7s 555 rrrrrcrrr,"[0.33,0.33,0.33]"
5534187

```

Figure 19 - Information Set Before

Processed CSV file has around 2.5 million entries

```

training_data > cfr_training_data_clean.csv
2751283 T7s 444 ccc, "[0.13,0.00,0.13]"
2751284 T7s 444 cccr, "[0.07,0.07,0.86]"
2751285 T7s 444 cccrrr, "[0.03,0.03,0.95]"
2751286 T7s 444 cccr, "[0.41,0.29,0.29]"
2751287 T7s 444 crcc, "[0.40,0.33,0.27]"
2751288 T7s 444 crcr, "[0.43,0.31,0.27]"
2751289 T2o 222 rc, "[0.17,0.39,0.45]"
2751290 T2o 222 rccr, "[0.14,0.48,0.38]"
2751291 T2o 222 rccrrr, "[0.13,0.75,0.13]"
2751292 T2o 222 rccr, "[0.12,0.49,0.38]"
2751293 T2o 222 rccrrr, "[0.11,0.78,0.11]"
2751294 76s 777 ccc, "[0.11,0.35,0.54]"
2751295 76s 777 ccr, "[0.11,0.37,0.52]"
2751296 76s 777 rcc, "[0.11,0.44,0.45]"
2751297 76s 777 rccrr, "[0.08,0.51,0.41]"
2751298 76s 777 rcr, "[0.11,0.73,0.16]"
2751299 76s 777 rccrrr, "[0.23,0.55,0.23]"
2751300 J3s 666 rcc, "[0.49,0.34,0.17]"
2751301 J3s 666 rcr, "[0.49,0.34,0.17]"
2751302 Q5s 777 rc, "[0.38,0.31,0.31]"
2751303 Q5s 777 rccr, "[0.67,0.24,0.09]"
2751304 Q5s 777 rccr, "[0.67,0.24,0.09]"
2751305 53s 666 ccc, "[0.57,0.27,0.17]"
2751306 AJo JJJ rrc, "[0.17,0.38,0.46]"
2751307 AJo JJJ rrcrr, "[0.15,0.44,0.41]"
2751308 AJo JJJ rrcrrr, "[0.12,0.41,0.47]"
2751309 AJo JJJ rrcrr, "[0.12,0.40,0.48]"
2751310

```

Figure 20 - Information Set After

Appendix D: Time Taken for One Game:

Bottom right shows it takes an average iteration (a game) 2.14s to complete between CFRv1 and CFRv2.

```

Cumulative after 815 game(s):
Player CFRv1: -1.01 sb/h
Player CFRv2: 1.01 sb/h
Simulating games: 1% | 815/100000 [37:18<58:52:57, 2.14s/it]

```

Figure 21 - CFRv1 VS CFRv2

References

888poker. (n.d.). *Poker Hand Rankings*. Retrieved from 888poker:
<https://www.888poker.com/how-to-play-poker/hands/flush-poker-hand-odds/>

- Billings, D., Burch, N., Davidson, A., Holte, R., Schaeffer, J., Schauenberg, T., & Szafron, D. (2003). Approximating Game-Theoretic Optimal Strategies for Full-Scale Poker. *International Joint Conference on Artificial Intelligence* (p. N/A). Acapulco, Mexico: N/A. Retrieved from <https://poker.cs.ualberta.ca/publications/IJCAI03.pdf>
- Billings, D., Davidson, A., Schaeffer, J., & Szafron, D. (2002). The challenge of poker. *Artificial Intelligence*, 201-240. doi:[https://doi.org/10.1016/S0004-3702\(01\)00130-8](https://doi.org/10.1016/S0004-3702(01)00130-8)
- Billings, D., Papp, D., Schaeffer, J., & Szafron, D. (1998). Opponent Modeling in Poker. *Proceedings of the Fifteenth National Conference on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conference* (pp. 493-499). Wisconsin: AAAI Press / MIT Press.
- Bowling, M., Burch, N., Johanson, M., & Tammelin, O. (2015, 01 09). Heads-up limit hold'em poker is solved. *Science*, 145-149. doi:DOI: 10.1126/science.1259433
- Brown, N., & Sandholm, T. (2016). Baby Tartanian8: Winning Agent from the 2016 Annual Computer Poker Competition. *International Joint Conference on Artificial Intelligence*. Retrieved from <https://www.cs.cmu.edu/afs/cs/Web/People/sandholm/BabyTartanian8.ijcai16demo.pdf#:~:text=,the%20challenges%20of%20imperfect%20information>
- Brown, N., & Sandholm, T. (2017). Superhuman AI for heads-up no-limit poker: Libratus beats top professionals. *Science*(Issue 6374), 418-424. doi:DOI: 10.1126/science.aao173
- Brown, N., & Sandholm, T. (2019, 07 11). Superhuman AI for multiplayer poker. *Science*, 365(6456), 885-890. doi:10.1126/science.aay2400
- Brown, N., Lerer, A., Gross, S., & Sandholm, T. (2018, 11 01). *arxiv*. doi:10.48550
- Callahan, K. (2016, 11 10). *Online Poker*. Retrieved from pokernews: <https://www.pokernews.com/strategy/tips-getting-started-pokerstars-26303.htm?>
- Chiswick, M. (2021, 02 03). *CFR - The CFR Algorithm*. Retrieved from aipokertutorial: <https://aipokertutorial.com/the-cfr-algorithm/>
- Chiswick, M. (n.d.). *AIPT Section 4.3: CFR - Agent Evaluation*. Retrieved from aipokertutorial: <https://aipokertutorial.com/agent-evaluation/#agent-vs-agent>
- Drumsta, V. (2020, 05 09). *Playable_poler_bot*. Retrieved from Github: https://github.com/vdrumsta/playable_poker_bot
- Gibson, R. (2012). Meet Hyperborean, the Poker-Playing AI What strategies help a computer program win at Texas Hold 'Em? *IEEE Spectrum*, N/A. Retrieved 04 24, 2025, from <https://spectrum.ieee.org/meet-hyperborean-the-poker-playing-ai>
- Gilpin, A., & Sandholm, T. (2006). A competitive Texas Hold'em poker player via automated abstraction and. *The Association for the Advancement of Artificial Intelligence*. Retrieved from <https://www.cs.cmu.edu/~sandholm/texas.aaai06.pdf#:~:text=dual%20variables,In%20this%20paper%2C%20we%20present>

- gtowizard. (2022, 05 09). *gto wizard blog*. Retrieved from gtowizard:
<https://blog.gtowizard.com/what-does-gto-aim-to-achieve/>
- Howcast. (2013, 12 6). *Poker Rules | Poker Tutorials*. Retrieved 04 26, 2025, from Youtube:
https://www.youtube.com/watch?v=oLSMasFvzxE&ab_channel=Howcast
- Ishikota. (n.d.). *PyPokerEngine*. Retrieved from Github:
<https://github.com/ishikota/PyPokerEngine>
- Jackson, E. (2011). *slumbot2019*. Retrieved from Github:
<https://github.com/ericgjackson/slumbot2019>
- Johanson, M. (2015, 02 19). *What is an intuitive explanation of counterfactual regret minimization?* Retrieved 04 27, 2025, from Quora: <https://www.quora.com/What-is-an-intuitive-explanation-of-counterfactual-regret-minimization/answer/Michael-Johanson-2>
- Lanctot, M., Waugh, K., Zinkevich, M., & Bowling, M. (2009, 12 07). Monte Carlo sampling for regret minimization in extensive games. *NIPS'09: Proceedings of the 23rd International Conference on Neural Information Processing Systems*, 1078 - 1086. Retrieved from <https://dl.acm.org/doi/10.5555/2984093.2984215>
- Lee, H. (2025, 04 23). *PokerHandEvaluator*. Retrieved from Github:
<https://github.com/HenryRLee/PokerHandEvaluator>
- Moravcik, M., Schmid, M., Burch, N., Viliam, L., Morrill, D., Bard, N., . . . Bowling, M. (2017). DeepStack: Expert-level artificial intelligence in heads-up no-limit poker. *Science*, 356(6337), 508-513. doi:10.1126/science.aam6960
- Neller, T., & Lanctot, M. (2013, 07 09). *An Introduction to Counterfactual Regret Minimization*. Retrieved 04 27, 2025, from <http://modelai.gettysburg.edu/2013/cfr/cfr.pdf>: <http://modelai.gettysburg.edu/2013/cfr/>
- PokerGo. (2020, 11 5). *High Stakes Feud | Daniel Negreanu vs Doug Polk*. Retrieved 4 25, 2025, from Youtube:
https://www.youtube.com/watch?v=CsDj1viQ4E4&ab_channel=PokerGO
- pokerstarsuk. (N/A). *Third Party Tools and Services Policy*. Retrieved from pokerstarsuk:
<https://www.pokerstars.uk/poker/room/prohibited/#:~:text=The%20following%20are%20prohibited%20at,a%20human%20to%20make%20decisions.>
- PokerStrategy.com. (n.d.). *The Five Player Types*. Retrieved from PokerStrategy.com:
<https://www.pokerstrategy.com/strategy/bss/five-player-types/>
- Professor Bryce. (2023, 04 25). *Youtube*. Retrieved 04 27, 2025, from Youtube:
https://www.youtube.com/watch?v=ygDt_AumPr0&ab_channel=ProfessorBryce
- Sandholm, T. (2015). Solving imperfect-information games. *Science*, 347(6218), 122-123. doi:10.1126/science.aaa4614
- UPSWING POKER. (2020, 02 05). *hands-per-hour-live-poker-vs-online*. Retrieved from upswingpoker.com: <https://upswingpoker.com/hands-per-hour-live-poker-vs->

online/#:~:text=a%20live%20game.-
,Online%20Poker,dramatically%20by%20playing%20multiple%20tables.

Walker, G. (n.d.). *Poker Winrates*. Retrieved from The Poker Bank:
<https://www.thepokerbank.com/strategy/other/winrate/#:~:text=Any%20winrate%20above%200%20is,should%20be%20happy%20about%20that>.

Waugh, K. (2015). *A Fast and Optimal Hand Isomorphism Algorithm*. Retrieved from
cs.cmu.edu: <https://www.cs.cmu.edu/~kwaugh/publications/isomorphism13.pdf>

Zinkevich, M., Johanson, M., Bowling, M., & Piccione, C. (2007). Regret minimization in games with incomplete information. *NIPS'07: Proceedings of the 21st International Conference on Neural Information Processing Systems*, 1729 - 1736. Retrieved from <https://dl.acm.org/doi/10.5555/2981562.2981779>