3 Improvements to Monte Carlo Counterfactual Regret Minimization for Texas Heads Up No Limit Poker

Jonathan Lin
California Institute of Technology
jonathan@caltech.edu

May 30, 2024

1 Introduction

Monte Carlo Counterfactual Regret Minimization (MCCFR) is the most common learning algorithm used in modern poker bots. My contributions, many inspired by similar methods in machine learning, involve improving MCCFR in the context of poker - multithreading, exploration decay, and biased reward functions. These contributions decrease MCCFR's training time and variance.

I plan to release the source code of this project sometime in the future. Hopefully you can learn something, whether that's about game theory, reinforcement learning, incomplete information games, etc, that you can use in your own work. It is expected that the reader is familiar with Texas No Limit Holdem Poker and have a high level understanding of reinforcement learning, deep learning, multithreading, and basic data structures. The Nash equilibrium strategy, GTO, and the bto will be used synonymously, because the Nash equilibrium strategy is GTO. And the bot is simply executing the strategy. I originally started this project because I wanted to create a poker bot, but I found many areas of improvement for MCCFR during my process.

2 Basic Game Theory

Game theory is the study of decision making in social situations involving competing players. Game theory is used in business, finance, politics, economics, psychology, and many more situations[1]. There is a type of game called zero-sum games. A player's benefit in a zero-sum game is exactly the other person's loss. So if all the gains and losses are added together at the end of a zero-sum game, the sum would be zero. Nash equilibrium, in the context of this paper, will be defined as a set of game theory optimal (GTO) strategies[2]. This Nash equilibrium will be where neither player can improve their expected payout by changing their strategy. No matter the situation, a Nash equilibrium strategy is guaranteed to not lose money over many games. This does not mean that you will make money, just that you will not lose money. Nash equilibrium and GTO will be synonymous in this paper.

A strategy is a probability distribution of actions in a given situation. All GTO strategies cannot be altered to increase expected payout, assuming we know nothing about the opponents' strategy. This differs from exploitative play, which is when one changes their strategies to specifically counter the opponent's weakness. For example, one can play extremely tight if the other person is a calling machine. Of course, one can play exploitatively if they know information about how the other player plays, but we'll focus on GTO strategies because that's more generalizable, you won't always know what type of player the other players are, your assumption(s) about others may not always be correct, and I find GTO more interesting.

Let's use the classic game of rock, paper, scissors to demonstrate the difference between GTO and exploitative play. Given that you know nothing about the opponent, the best strategy you can use is to simply play $\frac{1}{3}$ rock, $\frac{1}{3}$ paper, and $\frac{1}{3}$ scissors. This is GTO. Now let's say you know the opponent plays 34% rock, 33% paper, and 33% scissors. If you were playing exploitatively, it is in your best interest to play 100% paper. But this is risky because your observations may have been from just noise instead of from an underlying pattern (in other words, the other person had an equal chance of all 3 hands, but it just happened to be that they played more rock than the other hands due to randomness) and/or the other player may change their play style to exploit your changed play style. These concepts are generalizable to poker.

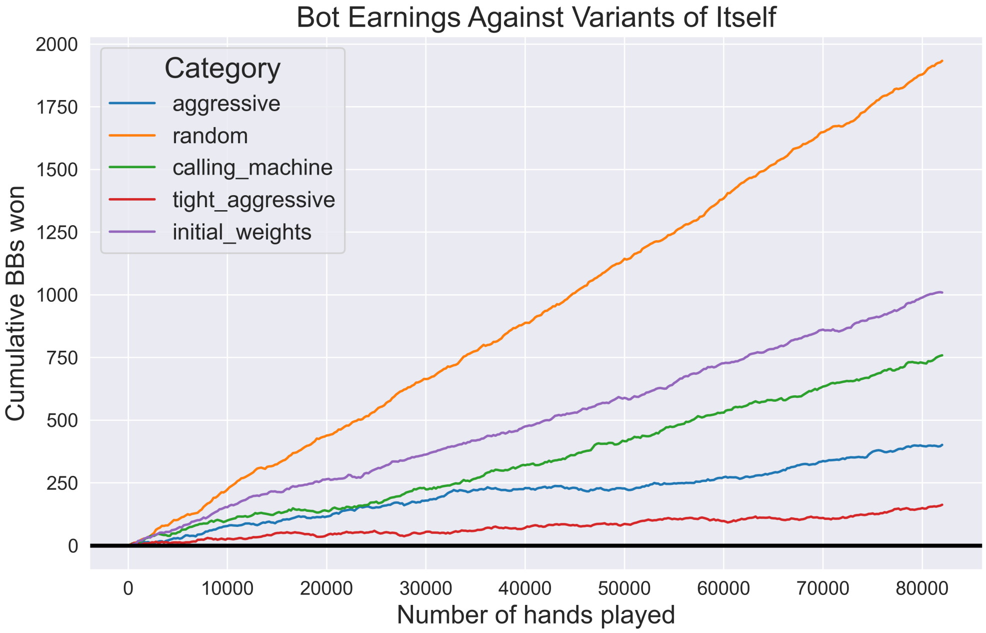

Figure 1: On average, the bot can beat all variants of itself. Thus, the bot is not exploitable and unbeatable, given that n games played is sufficiently large.

Poker, without a rake, is a zero-sum game because any chips a player gains is a direct result of other players' loss of chips. In heads up poker, virtually all poker bots attempt to reach Nash equilibrium. Lastly, poker is known as an incomplete information game. This means that the different players have access to different information, namely their cards. So each player has to make the best decision in a given situation without knowing all the information. You have to use information, such as past betting history, general priors, psychology, and other tells to make an optimal decision. The benefits of being able to make good decisions with incomplete information extends beyond poker. Many decisions in life are made with incomplete information.

3 Game Tree

A game tree is a way to represent all possible game states - see figure 2. A game state is a situation the game can possible be in. Each arrow, with a label, represents an action, such as calling. Orange nodes are states that require action. Black nodes are terminal nodes - there is no more action to be done. And green nodes are chance nodes, nodes where a chance action has to happen, such as dealing the flop or river. You can represent all possible game states by traversing down the game tree, starting at the root node.

Exploring the game tree means going down different unique paths. Obviously, some paths are going to be more common than others.

4 Counterfactual Regret Minimization

My bot, and most of today's best performing poker bots, use counterfactual regret minimization (CFR) to optimize their strategy. Essentially, CFR works by the model playing against itself and improving itself based on the outcome.

I recommend that you read the original CFR paper, first published by the University of Alberta in 2009, for more formal notation, but I will include

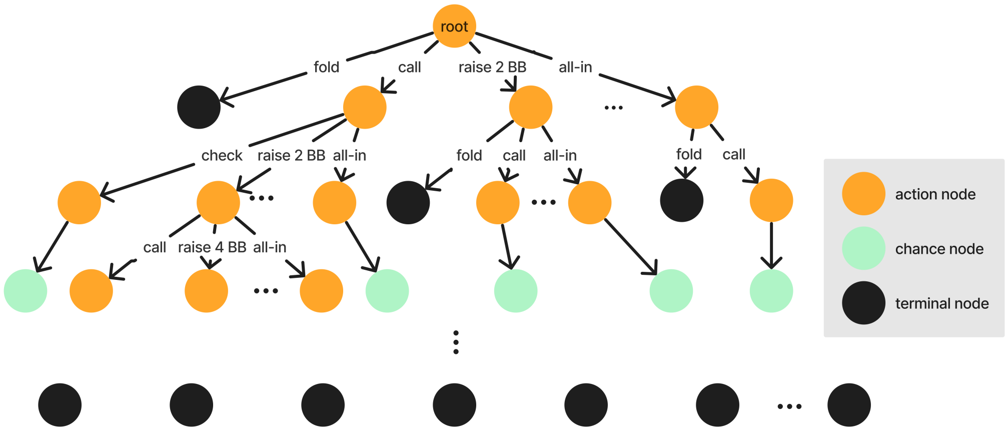

Figure 2: Poker Game Tree.

some important definitions here[3]. These definitions will be in the context of poker, but generalizing these concepts should not be too difficult.

History contains everything that defines a given situation in poker: the hands, the betting street, the action history, and the cards on the board. An *infoset* is the part of history only accessibly to a given player and is unique for each player; it essentially describes the situation the player is in. The main difference between a history and an infoset is that the infoset does not contain the cards of the other player. The *utility* of an action is how much money that action would have made, and the utility is solely measured by how many chips you gained or lost. *Regret* is the difference between the utility of a certain action and the action that was chosen. Lastly, a *strategy* is the probability distribution of the events that a player would do in a given situation. For example, the strategy of the small blind's first move after the flop, given that they hit top pair, could be:

Action	Probability
Fold	0
Check	0.2
Raise 0.33x pot	0.2
Raise 0.5x pot	0.2
Raise 0.67x pot	0.15
Raise 1x pot	0.1
Raise 1.5x pot	0.1
Raise 2x pot	0.05

Figure 3: Strategy distribution for a given situation.

A strategy essentially describes the probability of each action you should do in a given situation. A common question is to wonder why you cannot just always do the action with the largest probability. You cannot do this or else you'll become predictable. For example, if you have a really good hand on the river, the most common action would be to raise high. And if you have a bad hand on the river, the most common action would be to check. But if you always raise when you have a good hand and check when you have a bad hand, you are *unbalanced* and predictable. The point of this bot is to learn the probability distributions for every situation so that the other player never knows the true strength of your hand, thereby unable exploit you.

Typically when I refer to *weights* in this paper, I refer to the collection strategies and/or a specific strategy for a given infoset. This concept is taken from machine learning. The point of training a model is to learn the values within the matrices - the weights. The model is learning the probability distributions, so I'm calling them the "weights."

Vanilla CFR plays many games against itself and improves based on the results. Counterfactual regret is the regret weighed by the probability of the opponent reaching that state. This makes it so that events that may have a large impact on behavior, but are extremely rare, do not have a large effect on the strategy. For example, if the opponent puts you all in, you have ace high, there was a lot of preflop action, there are a lot of high value cards on the board, the board is extremely connected, and the board is mostly monotone (so there is flush potential), you may win a lot of money by calling with ace high. But the probability of you being ahead is so low that the Nash equilibrium strategy would put a low percentage on calling and a high percentage on folding in this situation.

For each training iteration in vanilla CFR, the algorithm traverses the entire game tree. A game tree is a tree representation of all the possible game states. This is extremely inefficient and not practical in large scale games because it takes too long for the algorithm to converge to Nash equilibrium. So we utilize MCCFR. MCCFR essentially utilizes random sampling in the learning part of the agent. Although more iterations are required to reach Nash for MCCFR, it takes a lot less time per iteration. MCCFR is less precise than vanilla CFR, but it takes a lot less time total for MCCFR to approximate Nash than vanialla CFR. So instead of multiplying each regret by the probability of it happening, we directly use the fact that random sampling represents the probability that the other player will reach the current state to put more weight towards events that happen more often and vice versa for less common events. Similar to vanilla CFR, MCCFR improves its strategy by playing against itself.

5 Bucketing (categorization)

It is not possible to reach all unique infosets in a feasible amount of time. In fact, there are more infosets available in poker than atoms in the universe. Therefore, a concept called "bucketing" is required for large games, such as poker. Bucketing is putting similar infosets, that would have similar strategies, in "buckets". An example of a bucket is top pair and good kicker. You would not play AcTc with the board Ts5s2h much differently than AcJc with the board Js5s2h. In both situations, you have top pair, a good kicker, and virtually no draws. So the training task becomes figuring out the strategies for all the buckets. And when the bot plays real games, it categorizes the infoset into the appropriate bucket, then uses the bucket's strategy. Therefore, bucketing decreasing the number of necessary strategies by many orders of magnitude, making approaching Nash faster. I'm also training on my laptop (2021 Macbook Pro with M1 Max), so minimizing training time is crucial.

Unfortunately, I could not find any existing effective bucketing algorithms on the internet; therefore, I had to create my own. It is lossless for preflop and lossy for postflop. But the way different hands in the same bucket should be played are approximately the same for postflop and exactly the same for preflop.

The preflop bucketing algorithm is simply <the rank of card #1><the rank of card #2><suited or not suited>. Hands, such as AhKh and AcKc are technically different, but they will both be bucketed as AKs and played the exact same way.

Figure 4 displays my bucketing algorithm for postflop situations. The postflop bucketing algorithm takes into account the board texture (if the board is double paired, number of suited cards, etc), the made hand, potential draws, and the number of previous betting streets that each player raised. Board texture describes different arrangements of the board. For example, a "monotone" or "flushy" board texture means a board of many cards of the same suit. A made hand is the hand you currently have. For example, pairs, trips, and flushes are made hands. Draws are hands that are incomplete and requires additional cards to become valuable. For example, if your hand and the board creates 2345, then it is currently worthless (high-card). But any ace or six that comes out will turn your hand into a straight. My bucketing algorithm only considers flush and straight draws because they are the most common draws people consider. Lastly, taking into account raising history is important for concepts like gauging villain hand strength and making a bluffing story more believable. For example, if the other player did not raise for all the betting streets and raises large on an in-

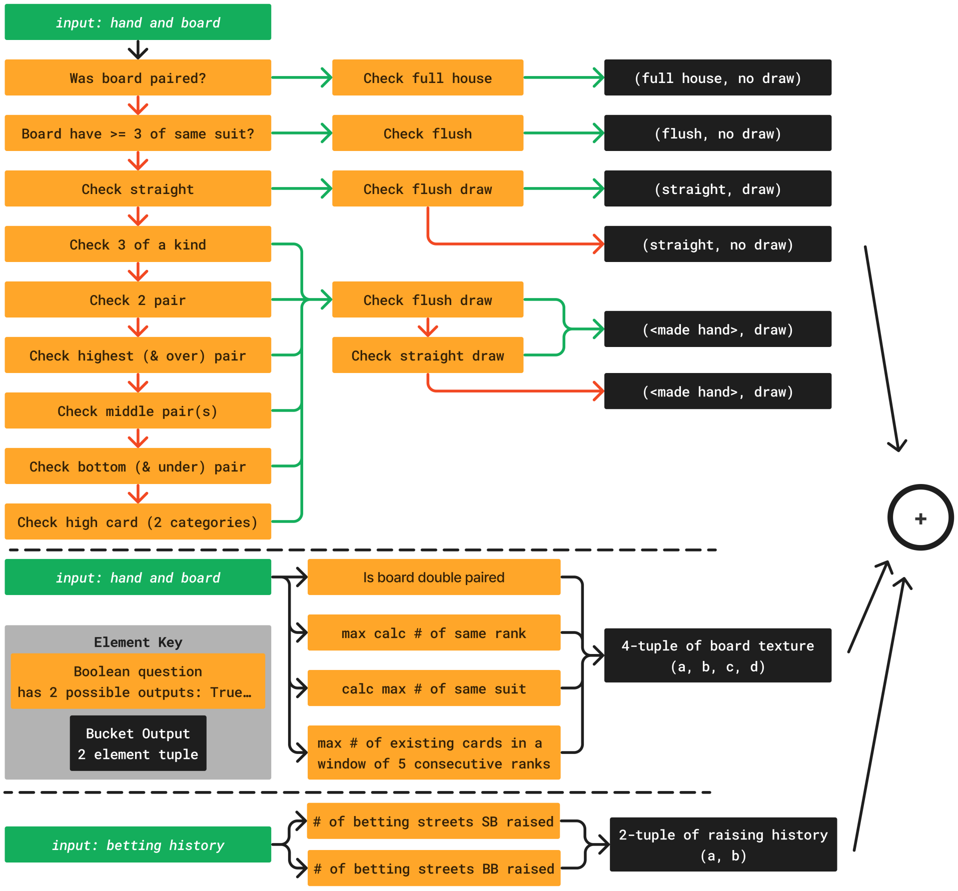

Figure 4: Postflop Bucketing Algorithm. A green arrow represents "yes", a red arrow represents "no", and a black arrow signifies to proceed. The three sections - made hand, board texture, and raising history - are concatenated to represent a bucket.

consequential river card, it is likely they are bluffing.

6 My Contributions

My contributions revolve around the nuances existing from training on a laptop, instead of a supercomputer. Therefore, I implemented multithreading to increase speed, biased reward functions to adjust learning, and exploration decay to decrease variance. Decreasing variance is important because I'm not able to train my model for a very large n. Therefore, exploration decay helps to decrease variance that is usually present when a model isn't trained for long enough.

6.1 Multithreading

I implemented multithreading in the MCCFR algorithm. For those unfamiliar, multithreading is where computer can (essentially) do multiple things at the same time. I created a thread lock for every pair of infoset and matching regret sum, strategy, and strategy sum so different threads can alter the data of different infosets simultaneously. I altered the original MCCFR program to create a new thread each time the external_cfr function, the function that runs MCCFR for a given game state, is called if there are more than 3 possible actions and it is preflop or flop. The specific hyperparameter of the minimum number of actions and rounds left required to create a thread was determined by a hyperparameter sweep. You cannot create a thread for every possible recursive call because it is expensive to create a thread and the diminishing positive gain from each additional thread. Essentially, you want to balance creating a thread for new game tree branches, but ensuring that there are enough calculations to be done in that game subtree for adding a thread to be worth it. For example, the bot would not have many regret calculations left if it was on the river round. So, sacrificing computing resources to create the thread and clog up all the other threads only to barely do any work is wasteful. Also, the locks are the main bottle neck of the multithreading algorithm anyway.

Contrary to this section's introduction, a computer can actually still only do one thing at a time. Multithreading gives the illusion that the computer is doing multiple things at a time by constantly doing a little bit one one task and then switching to another one. The amount of time a processor spends on each thread is dependent on that system, but the time is usually extremely small. To put it nontechnically, each task is put on a thread. But the computer cannot instantly switch between threads. It takes time and resources for the computer to switch between threads and to create a thread. That is why we cannot simply create a new thread for each subtree in the game tree. Instead, we have to maximize the number of threads up until adding more threads actually slows down the learning.

The multithreading optimizations decreased the time it takes for one iteration to run from ≈ 30 seconds to $\approx \frac{1}{2}$ seconds, a 60x improvement. Note: this was done on a M1 Max. Exact results may vary.

6.2 Exploration Decay

As we all know, poker is a game with high variance. The MCCFR algorithm can be sensitive to extreme situations, especially in the beginning, possibly throwing the learning algorithm off balance. Thus, we must sacrifice accuracy in the beginning for the sake of exploration. This is essentially a form of regularization, a term taken from machine learning, that decreases as training time increases.

We want the algorithm to explore different game paths in the beginning, because a few uncommon situations in the beginning of training has the potential to completely throw of the weights for a given infoset. This is similar to you learning a new game. If you've only played the new game a couple of times, it would make sense for you to continue trying different actions to see if what happened to you in a given situation was simply due to randomness or an underlying pattern you can exploit. The idea behind prioritizing game tree exploration is the same idea. The algorithm wants to explore many different actions in the beginning, even if these actions may seem unfavorable according to the simulated games the algorithm has trained on. Messed up (non-GTO) weights in the beginning can also have a positive reinforcement loop, because the bot plays against itself to learn. Let's say that the bot learns early on to go all-in when the board is dry (rainbow, low value cards, disconnected suits, etc). Then the bot may learn to call a lot in that situation, even when it's not supposed to, because itself would frequently shove with nothing. Therefore, this is an effective way to integrate regularization in the learning algorithm.

We keep this exploration open by setting a minimum probability value for a given action. In the beginning, since not many games have been played, we would naturally want to set it relatively high. I personally set it to 1%. Then as the number of simulated games increases, you would want to decrease this probability, because it is likely that the observed data is more accurate. The minimum linearly decreases until it hits 0% at $40,000$ games. So, actions that the algorithm deems should be played 0% of the time will actually never be played (such as folding pocket aces preflop) by the end. The formula used do determine the minimum, where y = minimum percentage and x = games played is modeled by:

$$y(x) = \begin{cases} -2.5 \cdot 10^{-7}x + 0.01) & x \leq 40,000 \\ 0 & x > 40,000 \end{cases}$$

This formula was created from simple trial and error. When the minimum was too large, the algorithm wouldn't learn as well since it's not really learning from its mistakes. And when the minimum was too small in the beginning of training, it would virtually not make a difference. Figures may be added in the future comparing the variance of training accuracy with and without exploration decay in future versions of this paper.

Feel free to skip this paragraph if you are *not* familiar with MCCFR. Every time the strategy weights are called for an infoset, I set the minimum probability to the given minimum chance at that moment. And then when the strategy is added to the strategy sum data, the altered strategy is added. But theoretically, as time goes on and the minimum probability decreases, the average strategy should still converge to equilibrium. This method simply decreases variance in the beginning of training.

6.3 Biased Reward Function

I modified the reward function for MCCFR to be biased. A modified version of the amount of money won or lost in a given is fed to the learning algorithm. Essentially, I experimented with how punishment for losing money would affect the learning. For example, if the bot won 10 BB when using y_2, then nothing changes. But if it lost 10 BB, then the algorithm would treat it as if it lost 1.5(10) = 15 BB. This essentially makes it more painful to lose x amount of chips compared to winning the same amount.

$$y_1(y) = \begin{cases} y & y < 0 \\ y & y \geq 0 \end{cases}$$

$$y_2(y) = \begin{cases} 1.5y & y < 0 \\ y & y \geq 0 \end{cases}$$

$$y_3(y) = \begin{cases} -0.35 \cdot (-y)^{1.32} - 5 & y < 0 \\ y & y \geq 0 \end{cases}$$

$$y_4(y) = \begin{cases} \min\{-0.35 \cdot (-y)^{1.32}, y\} & y < 0 \\ y & y \geq 0 \end{cases}$$

y_1 represents no modifications to the reward function, y_2 is a linear multiplier, y_3 is an exponential multiplier, and y_4 is a less impactful version of y_3.

	y_1	y_2	y_3	y_4	Worst Profit
y_1	-	-0.3	-0.18	-0.22	-0.3
y_2	0.3	-	0.10	0.06	0.06
y_3	0.18	-0.10	-	-0.05	-0.10
y_4	0.22	-0.06	0.05	-	-0.06

Figure 5: Profits for each reward style against other variants. The play style on the left is the player of interest, and the play styles on the top are the opponents. The last column is the worst performance for each play style. Each pair of (y_i, y_j) played $n = 2,000,000$ games against each other, swapping SB and BB positions between each game.

Figure 5 illustrates the performance of each playstyle against the other ones. As you can see, y_2, a linear multiplier on the punishment aspect of the reward function, performed the best. In fact, it was profitable against every other playstyle. The second best is y_3, the polynomial function. y_4 is third, a less harsh version of y_3. And in last, is the unmodified version of MCCFR.

Therefore, it is beneficial to use a biased reward function. This is similar to class imbalances in machine learning problems.

7 Initial Weights

Typically in CFR, initial weights are uniform. So the probability of doing any given action is equal to the probability of doing any other action. Minimizing training time is crucial, so I did not have the starting weights be uniform for an unseen infoset.

First, I simulated tens of millions of games of poker. With each game, I kept track of the win, tie, and loss rates of each bucket. Then, I created a preassigned probability distribution for actions based on ranges of win rates. For example, if the win rate of a given infoset is below 20%, then the starting probability of folding/checking is extremely high, and the starting chance of going all-in is relatively low. Theoretically, this method should decrease the required training time for the algorithm to reach Nash, because the initial weights should be closer to GTO than strictly uniform strategies. I used existing knowledge I already knew about poker, called "domain knowledge" in general terms, to create a process for choosing the initial weights.

Most demonstrations of CFR have the initial weights to start uniformly because we assume that we don't know what GTO actually looks like. But, this paper is more of a proof of concept rather than a mathematically rigorous proof. So I've taken some liberties by starting the weights closer to true GTO rather than uniform.

8 Exploitability

Exploitative poker is a playing style that is dependent on your opponent. For example, if you know that your opponent plays extremely tight and only plays extremely good hands, then you know to fold anytime that person raises. Being unexploitable means that even if the opponent knew your exact strategy, there is nothing they can do to win more chips from you. A strategy that reaches Nash equillibrium is unexploitable.

I measure exploitability by playing the bot against different versions of itself. Theoretically, if the bot plays GTO, then it should be profitable against any variant of itself. The strategies for these bot variations are created by taking the strategy profile of the original bot and modifying it appropriately for the given play style. If the bot is able to consistently be profitable against all the following play styles, the bot is unlikely to be exploitable. Obviously, exploitability is more detailed and nuanced than described, and there are more complex poker strategies than the ones I listed. But, my evaluation of exploitability is adequate for the time being.

- Aggressive: The bot folds less and raises larger amounts more often.
- Tight: The bot folds more and raises larger amounts less often.
- Random: All strategies in the strategy profile are uniform. All possible actions in a given situation are equally likely to be selected.
- Calling Machine: The bot is more likely to call than any other action.
- Tight Aggressive: The bot plays tight if the win rate of the current situation is less than a certain value and plays aggressive if the win rate of the current situation is more than a certain value. This is essentially an unbalanced player - their actions perfectly reflect their hand value.
- Deterministic: The bot only plays the action with the highest probability. It's not balanced, and the action it does is essentially a direct representation of the strength of its hand.
- Initial Weights: The strategy is simply the initial strategy for a given unseen infoset.

9 Results

The learned algorithm is not exploitable. In other words, as long as n games is high enough, the bot will not lose money.

Figure 6 illustrates the win rates of the combined bot against variants of itself. $100,000$ games were played between the bot and each variant, totaling $500,000$ games.

Figure 7 displays the exploitability of the 4 individual bots against variants of itself. Each bot was trained with the same initial setup, and version is profitable against each of its variants, for the most part. It sometimes struggles against the tight aggressive play style because the bot currently has no real-time learning. Therefore, it does not adapt to how the other player is playing. So if the other player is blatantly only playing aggressive when it has a good hand and tightly when it has a bad hand, the bot cannot adjust to this. The bot still remains profitable against all variants of itself, despite this restriction. The exploit lines are provided for the individual bots to show the viability of training a single bot to reach GTO.

The results in Figure 1 were generated by averaging the results in Figure 7. This was done to reduce variability and simplify the results.

Variant	Original Win Rate (BB/hand)
Aggressive	1.749
Tight	0.609
Random	4.000
Calling Machine	1.788
Tight Aggressive	0.125

Figure 6: This table shows the bot's win rate, measured in BB/hand, against several of its variants.

The viability of the bot is still being tested against experienced players.

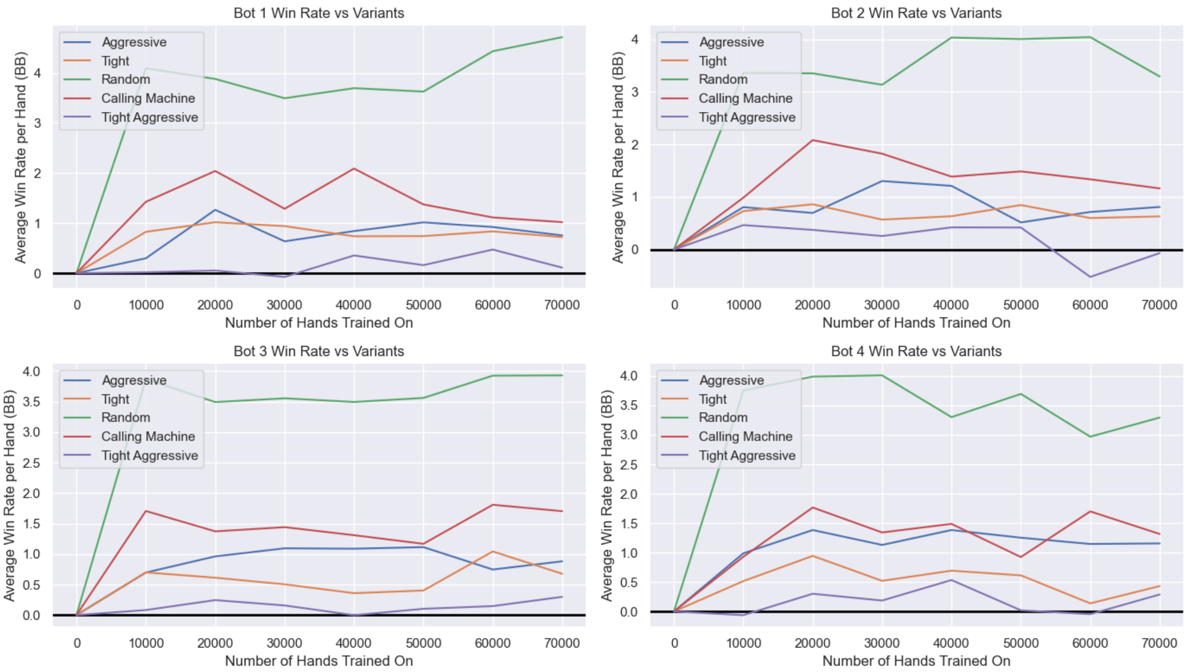

Figure 7: Exploitability of the individual bots trained. All 4 versions stay profitable against virtually all its variants.

10 Game Modifications

The first Artificial Intelligence (AI) that definitively beat humans in poker was Libratus, created by a Carnegie Mellon University research team. Libratus used a similar algorithm to MCCFR for part of its solving process, but this bot took 15 million core hours to train on the Pittsburgh Supercomputing Center's Bridges computer, amounting to 2.5 petabytes of data[4]. The computing power and storage required to train a less abstracted version of the game is not feasible with my resources. Therefore, I had to make several changes to the game that still preserve the core ideas of poker, could easily be reversed, and would barely affect one's game strategy.

Both players start with 100 BB in every game, instead of starting with the number of chips left over from the previous hand. The stack size affects gameplay. So, my software would have to train on every single combination of starting stack pairs from 1 BB to 100 BB if this modification was not made. Even with this modification, my bot generated 1 gigabit of data. I chose 100 BB because that is the standard buy in amount.

There is a limited number of raise sizes. Preflop, a player can only raise with the following number of BB: $1, 2, 4, 8, 16$. These selection of raise values were chosen because each raise value after a given raise value are valid. For example, if a player raised to 2 BB, all values after 2, namely $4, 8, 16$, are valid raise values that the next player could raise to. Postflop, a player can raise $0.3x, 0.5x, 0.7x, 1.0x, 1.5x, 2.0x$ pot. The validity of the raise logic is similar to preflop raise values, and these values are typical raise values postflop. Restricting the raise values also removes the need for an action abstraction algorithm, such as the problem if an opponent's raise of 3 should be categorized as 2 or 4 in the bot's decision making. There is a wide enough range of raise values to cover most actions, yet granular enough for the raise values to distinguish between intended strategies.

11 Deep Learning

A problem with utilizing CFR for such a large scale game, such as HUNL Poker, is that the average strategy takes a long time to converge to equilibrium. A possible solution to this is to utilize deep learning to output the strategy for a given infoset because deep learning has been proven to be effective in generalizing complex patterns. Noam Brown, et al. have successfully created a version of CFR that utilizes deep learning [5].

12 Future Plans

There is still a lot of work to be done on this project. Here is a list of things I plan to do in the near future regarding this project:

- Continue playing with deep learning and try to implement material I learned in CS155 (a theoretical machine learning course at Caltech) and see if performance improves.
- Create a poker bot for multi-way tables of Texas No-Limit Holdem Poker, though this will be far more complex than heads up and only be pursued if this bot proves to be extremely effective.

13 Additional Resources

Here are some resources, in order of decreasing importance, I found helpful in teaching myself the concepts behind creating this bot. I also recommend reading through the cited references at the end of this paper for a formal framing of CFR, game theory, and adjacent topics.

1. A YouTube video that helps to visualize vanilla CFR: link
2. A video lecture by Noam Brown, one of the leaders of the CMU team that created Libratus, about Libratus, game theory, and various forms of CFR: link
3. A YouTube video that gives an idea on what coding CFR may look like: link

References

[1] E. Picardo, “How game theory strategy improves decision-making,” Dec 2023.

[2] J. Nash, “Non-cooperative games,” *Annals of Mathematics*, vol. 54, no. 2, pp. 286–295, 1951.

[3] M. Zinkevich, M. Johanson, M. Bowling, and C. Piccione, “Regret minimization in games with incomplete information,” in *Advances in Neural Information Processing Systems* (J. Platt, D. Koller, Y. Singer, and S. Roweis, eds.), vol. 20, Curran Associates, Inc., 2007.

[4] B. Spice, “Poker play begins in “brains vs. ai: Upping the ante”,” Jan 2017.

[5] N. Brown, A. Lerer, S. Gross, and T. Sandholm, “Deep counterfactual regret minimization,” 2019.