Creating a Texas Heads Up No Limit Poker Bot

Jonathan Lin
California Institute of Technology
jonathan@caltech.edu
Note: This paper is incomplete and still being worked on

1 Introduction

This paper serves as a journal for my experiences in creating a bot that approximates Nash Equilibrium for Texas Heads Up No Limit (HUNL) Poker. Hopefully you can learn something, whether that's about game theory, reinforcement leraning, 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 and the general concept of reinforcement learning and deep learning.

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. 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. Poker, without a dealer rake, is a zero-sum game because any chips a player gains is directly as a result of another player's loss of chips. In a two-player zero-sum game, virtually all poker bots attempt to reach Nash equilibrium. It is a strategy where neither players can improve their expected utility by changing their strategy. 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 - this state of uncertainty is why I find the game so fun. You have to use information such as past betting history, general priors, psychology, and other tells to make an optimal decision.

3 Counterfactual Regret Minimization

My bot, and most of today's best performing poker bots, use counterfactual regert minimization (CFR) to optimize their strategy. The first CFR paper was originally published in 2009 by the University of Alberta (double check). In a nutshell, CFR works by the model playing against itself and improving itself based on the outcome. I recommend that you read the original paper for more formal notation, but I will define some important definitions here. History contains everything that defines a given situation in poker: the hands, the betting round, the betting history, and the cards on the board. An infoset is the part of history only accessibly to a given player. 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 much money you made. Regret is the difference between the utility of a certain action and the action that was chosen. Lastly, the 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 preflop when they have pocket aces could be:

Action	Probability
fold	0.0
call	0.1
raise 2 BB	0.1
raise 4 BB	0.25
raise 8 BB	0.25
raise 16 BB	0.3

The vanilla CFR algorithm essentially 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, 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 in this situation.

For each training iteration in vanilla CFR, the algorithm traverses the entire tree. This is extremely inefficient and not practical in large scale games because it takes too long for the strategy to converge to Nash Equilibrium. So we utilize Monte Carlo CFR (MCCFR). MCCFR essentially utilizes random sampling in the learning part of the agent. Although more iterations are required to reach Nash, it takes a lot less time per iteration. 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. But similar to vanilla CFR, MCCFR improves its strategy by playing against itself.

4 Bucketing

A concept called "bucketing" is required for large games because it is not possible to each reach of the unique states. In fact, there are more infosets available in poker than atoms in the universe. The bucketing algorithm I use is lossless for preflop and lossy for postflop. But the buckets are close enough for postflop situations that the strategy should be approximately the same for all the possible situations in a given bucket. Therefore, bucketing decreasing the number of infosets by many orders of magnitude, making approaching Nash faster. I'm also training on my laptop (2021 Macbook Pro with M1 Max), so performance efficiency is crucial.

Unfortunately, I could not find any existing effective bucketing algorithms on the internet; therefore, I had to create my own.

5 Deep Learning

A problem with utilizing CFR for such a large scale game, such as HUNL Poker, is that the average strategy takes too long 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.

6 Multithreading

I implemented multithreading in the MCCFR algorithm. Multithreading is where computer can (essentially) do multiple things at the same time. I created a lock for every pair of infoset and matching regret sum, strategy, and strategy sum so different threads can alter the information of different infosets simultaneously. I altered the original MCCFR program to create a new thread each time the external_cfr function is called if there are more than 3 possible actions and it is preflop or flop. 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. If the bot is on the flop round, then there are not many regret calculations left. So sacrificing computing resources to create the thread only to barely do any work is wasteful. Also, the locks are the main bottle neck of the multithreading algorithm anyway. If you are unfamiliar with multithreading and imagine that you are doing a 100 question test, and you can have n amount of students that are randomly assigned questions and can work simultaneously. But each question can only be worked on by a single student, and each student costs the school x dollars. In the beginning, each additional student is worth the x dollars because the speed of finishing the test significantly increases. But as you approach 100 students, the marginal benefit does not outweigh the marginal cost. There ends up being a lot of students that sit idle.

7 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:

- Train multiple models and average the result. So for a given infoset, I run the infoset through x different models and then sample randomly from the outputs. I want to try this because the performance of the poker bot has extremely high variance. For example, sometimes the bot to be exploitable from tight players, and sometimes the bot will be exploitable from loose

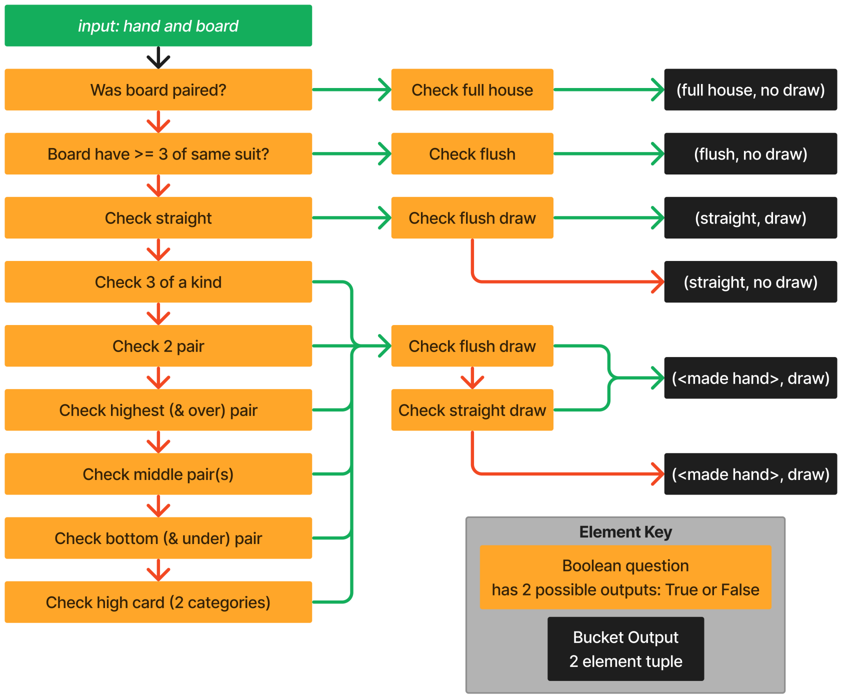

Figure 1: Postflop Bucketing Algorithm.

players. And the bot/weights were both trained from scratch in both situations.

- Continue playing with deep learning and try to implement stuff I learned in CS155 (a theoretical machine learning course at Caltech) and see if performance improves.