POKER AGENT USING ENHANCED SELF-PLAY TECHNIQUES

INTRODUCTION

Poker is a partially observable stochastic game. It is a partially observable game because, unlike the games like chess where the entire information related to the game is available to each of the players at any given point, each player in a game of poker only has the knowledge of their own pair of cards and the opened community cards, they do not have knowledge of the private cards dealt to other players. It is a stochastic game as the cards are randomly shuffled at the beginning of each game before each player is dealt with a pair of cards and community cards are laid out. Poker can be usually played with 2-10 players, and as the number of players increases, the complexity of the game increases. This inherent partially observable stochastic nature of Poker has piqued the interest of AI research community and ours as well, as an AI agent would need to be successful at assessing the winning chances of the combination of pair of cards in hand and the opened community cards while inferring the confidence levels and winning chances of other players solely based on their actions before making a decision.

In a single game of Texas no limit hold 'em Poker, in each round there are four betting stages known as streets, they are as follows: **Preflop**, **Flop, Turn, River**. At the beginning of each round, each of the players is dealt with a pair of cards and 5 community cards are closed and placed from a randomly shuffled deck of cards. In each of the betting stages, the players can choose an action (**Fold**, **Call** or **Raise** by any value from the amount they have with them). In **Preflop** street all the community cards are hidden, in the **Flop** street the first 3 community cards from left are opened, in **Turn** street the 4^{th} community card is opened and finally in the **River** street, the 5^{th} community card is opened. The **small blind** tag in a game of poker is given to the player who would start the round by placing the mandatory minimum bet amount. In case there are multiple rounds in a game of poker then this **small blind** tag keeps shifting to the left in the clockwise direction after each round. In a 6-player variant of Poker, based on all the combinations in which the cards could be dealt and the number of possible actions each player could take at each state would make the number of possible discrete states across the four streets be in trillions. An agent naively exploring all these states through self-play reinforcement learning to come up with optimal action policies would not be feasible. This was what motivated us. We wanted to explore, try out and come up with a set of algorithms that could be employed while training an Agent to come up with close to optimal action policies in the minimum possible time to play a 6-player Poker game decently well. For this we needed algorithms to make the most out of the experiences from each of the games during the training phase.

DETOUR: Before we proceed any further since the term **action policies** would be extensively used throughout this report, we would like to define what action policies in our perspective are. A distinct action policy would be present for each of the following combinations: player $\mathbf{p_i}$ + pair of cards held by player $\mathbf{p_i}$ + who the small blind is. Having action policies for each of these combinations is essential so that the agent would be able to play for any of the card combinations, for any of the players starting as the small blind and as any of the 6 players. Also, having separated out action policies would ensure that, for any player, only the relevant action policy is loaded onto the RAM, reducing the amount of data to be held in memory significantly. In our case, if we had all the action policies in a single data structure then we would have had to load the entire data structure of roughly 600MB for every player for every round into the memory, instead, since we separated out the action policies based on the above-mentioned combinations, we had to load data structures of size under 2MB per player per round. Each of these action policies is a nested dictionary that could go up to a depth of 24 since there are 6 players and 4 stages of betting. The nesting of dictionaries would terminate either when the game ends or when $\mathbf{p_i}$ reaches a terminal state like **FOLD**. Any given action policy would start with the small blind (**sb**) player's id as the key and the value would, in turn, be a nested dictionary whose key would be the action taken by **sb** and the value would be another nested dictionary with the id of the player who would play next as the key and the value would be constructed in a similar fashion. In an action policy related to player $\mathbf{p_i}$, the transitions based on actions of all the players are captured, in addition to these, for the states of $\mathbf{p_i}$, the information known to $\mathbf{p_i}$ along with the regrets (a measure of the desirability of an action) corresponding to the actions are

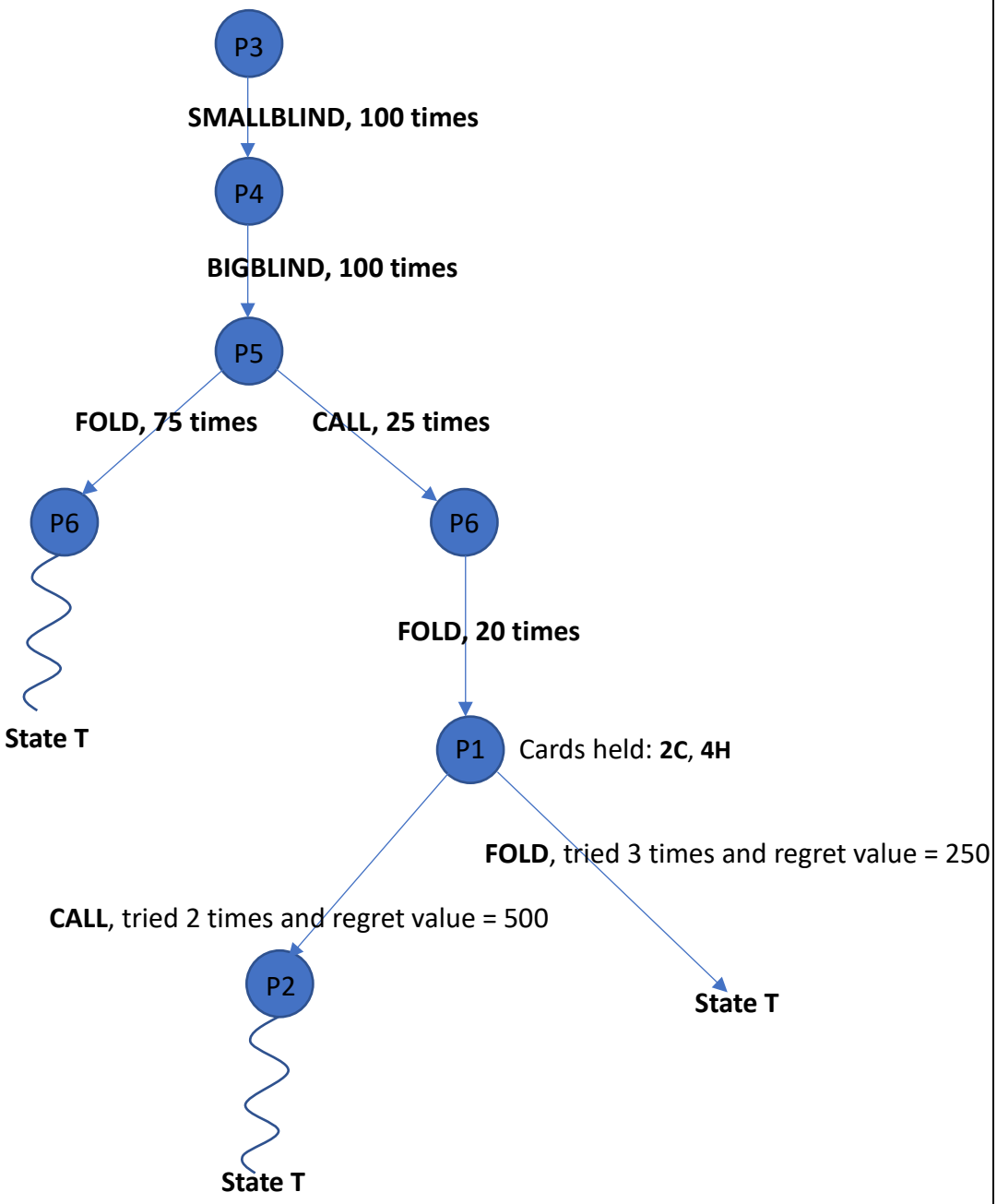
captured as well. Let us take the following example for better understanding, the action policy for player $\mathbf{p_1}$ for the card pair **2C**, **4H** (2 of Clubs and 4 of Hearts) with player $\mathbf{p_3}$ as small blind (**sb**) before using any abstraction algorithms would be something like this:

{“p3”: {“SMALLBLIND”: [100, {“p4”: {“BIGBLIND”: [100, {“p5”: {“CALL”: [25, {“p6”: {“FOLD”: [20, {“p1”: [[(2C,4H), (“There would be community cards here in the subsequent rounds”)], [[((3, 250), “FOLD”), {“FOLD”: {}}], [((2, 500), “CALL”), {“CALL”: {“p2”: ……..}}]]]}}]}}], “FOLD”: [75, {“p6”: [30, {……..}]]}}}]}}]}}

In the above example **p5** played **CALL** and **FOLD**, these are captured by the policy and are highlighted by the arrows.

Below is the tree representation of the above-mentioned sample action policy.

State T is some terminal state for **P1** or end of the game. **BIGBLIND** is a mandatory bet that the player following the small blind needs to put in.



As it can be observed from the example action policy for **p1**, for players other than **p1,** only their actions are captured, whereas for **p1** all the information essential for **p1** to make a decision at that given state is captured. This way for each of the 6 players $\mathbf{p_i}$ and each of the small blind (**sb**) value and for each of $^{52}\mathbf{C_2}$ card combinations which the player $\mathbf{p_i}$ could hold we would have 47,736 discrete action policies. With our **Abstraction** and **Rotation** Algorithms, we brought this number down to 546 discrete action policies.

OVERVIEW

As a part of this project, we were able to build a decently playing **Poker Agent** which learned the action policies from tabula-rasa (no prior knowledge or clean slate) state through self-play techniques alone. For achieving this we built an agent whose architecture mainly comprised of two components:

1.) The training algorithms
2.) The real-time gameplay algorithms

The training algorithms comprised of **Information abstraction**, **Action abstraction** and **Rotation** algorithms which aided in drastically reducing the total number of different action policies to be generated, and the number of possible discrete states in each of these action policies. They also included the **Action picker** algorithm for choosing an action at each state and **Regret updating** algorithm which was a form of regret minimization algorithm which was essential in coming up with the optimal action policies. These action policies were then used by the real-time gameplay algorithms during the evaluation of our agent. The real-time gameplay algorithms reused **Rotation** algorithm and **Action picker** algorithms from the training component along with **Pseudo Harmonic Action Mapping** algorithm (introduced by Ganzfried, S., & Sandholm, T. (2013)) which was essential in mapping the opponents' actions which were not present in the action policies to closest actions present in the policies.

This agent, using its action policies built through 500,000 self-play games was profitable and consistently won against various other poker agents in multiple 6-player poker games. These other poker agents were the ones we found available online, its previous version after 300,000 games and also the agents built by another team in the class. Details related to these games would be discussed in the evaluations section.

PRIOR RELATED WORK

When we decided on pursuing this project, we went through multiple prior research works done in building poker agents using different techniques to understand how various researchers have tackled this problem. There were a set of works that focused on coming up with heuristic functions based on domain knowledge in order to evaluate a state encountered based on the cards in hand and the information about the community cards. These algorithms took decisions based on the values returned by the heuristic function. These works required extensive domain knowledge in order to be able to design a function that would be capable of assessing a state based on the information available at that state and also the knowledge necessary for coming up with a decision ladder based on the values from the heuristic function. Another set of works focused on exploring various reinforcement learning techniques in order to enable the agents to come up with their own optimal policies. Of these, the ones we found interesting were the two-player and six-player Poker Agents built by Noam Brown and Toumas Sandholm. They came up with and leveraged various abstraction algorithms and used them during self-play with a linear variant of Monte Carlo Counterfactual Regret Minimization algorithm to let the agent come up with baseline strategies which they called blueprint strategy. These baseline strategies were then combined with real-time depth limited search algorithms and **Psuedo Harmonic Action Mapping** algorithm to choose optimal actions at a given state in the game. The generation of baseline strategies for both these agents required a lot of computation power, millions of games during the training phase and also the baseline strategy for the 6-player poker agent occupied close to 500GB of memory. Even though we were inspired by these works and leveraged abstraction, self-play and regret minimization algorithms at a very high level, the way in which we used them to handle the computational and time limitations we had was very much different. As a part of information abstraction, in subsequent betting stages after **Flop**, in this work, they relied on bucketing strategically similar information states. For figuring out strategically similar information states they used K-Means clustering on a large dataset with millions of prior poker games data. We, on the other hand, used **expected hand strength** value as

a metric for bucketing together strategically similar information states. Another key aspect in which our work differs from these works is that these baseline strategies held the belief distributions (Probability distribution across all the card combinations) of opponents' card pairs based on their actions. Our work, on the other hand, with an intuition which we would discuss in the approach section safely omits holding these belief distributions.

APPROACH

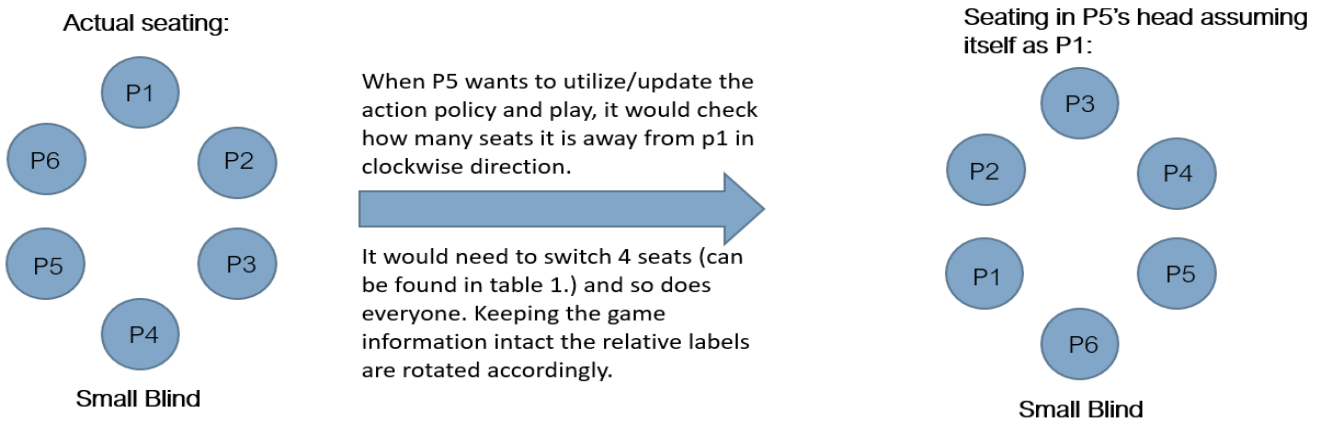
In order to spend the maximum amount of time on the algorithms necessary for building our agent, we chose to rely on utilizing an existing poker engine (**PyPokerEngine**) available for Python. This poker engine has all the components necessary for conducting a 6-player game of poker and the methods for accessing the game results, which are essential in updating the action policies. During each game, PyPokerEngine holds all the game-related information and conducts the game. In each round of the game, the poker engine requests an action from each of the players $\mathbf{p_i}$ and updates the round related information accordingly. This request for action is done by passing the following three parameters: (i) hole cards, (ii) valid actions and (iii) round state. The **hole cards** are the pair of cards dealt to the player $\mathbf{p_i}$, **valid actions** are all the possible actions that the player $\mathbf{p_i}$ can take at that given state based on the game rules and **round state** is a dictionary consisting of the following relevant information: (i) street of the game, (ii) who the small blind is, (ii) pot value (consists of both the main pot value which is the cumulative of the bets placed by players so far and the split pot value, if any), (iii) seating arrangement of the players along with amount of money they have, (iv) community cards opened so far and (v) the action histories so far in the game (These contain all the actions taken by each of the players since the game has begun).

Before utilizing this poker engine for setting up self-play environment, we focused on the algorithms which would be essential for the self-play process. Details of each of those algorithms are as follows:

ROTATION ALGORITHM: As mentioned in the detour section earlier, for the agent to play successfully irrespective of which player the agent is playing as, who the small blind is and what card pair the agent is holding, we require 47,736 discrete action policies for each of these combinations. With an intuition that we would be able to reduce the total number of policies by a factor of 6 if we could come up with a way to enable the agent to use/update action policies associated with a single player **p1** irrespective of which player out of the six players (**p1, p2, p3, p4, p5, p6**) the agent is playing as, we started working on designing this rotation algorithm. The main idea is that, instead of having action policies for all the six players which the agent could play as, we maintain, and update action policies associated with a single player **p1** alone and provide a way so that the agents playing as **p2, p3, p4, p5, p6** could use and update these same action policies. The simple technique we came up with for enabling this rotation is that while an agent is playing as a player other than **p1,** it would access the seating arrangement provided by the game engine and find the number of seats away it is from **p1** in clockwise direction and store it as **rotation_value**. Since all the players in a game of poker are seated in a cyclic fashion, all the player labels (**p1, p2, p3, p4, p5, p6**) associated with the game information are shifted to the right in the clockwise direction by this **rotation_value.** Once the game information is rotated as mentioned above, the agent which performed this rotation would be able to successfully utilize and update the action policies associated with **p1.** This happens because, once the game information is rotated, the agent which performed this rotation would consider itself as **p1** based on this updated information, in which the agent would be seated as **p1** after the rotation. The below table consists of the rotation values for each of the players the agent could play as.

Table 1.

Player	Rotation Value
P1	0
P2	1
P3	2
P4	3
P5	4
P6	5



The same process of rotation as in the above diagram is followed for any of the 6 players who want to use/update action policies. Post the rotation, the player would be able to use/update action policies as if the player is **p1**.

Let us take the following tabular examples of rotation for better understanding. The game information here isn't an actual representation of the game information but a basic sample being used for illustration purposes.

Table 2.

Agent as	**Actual Game Information**	**Rotated Game Information**	**Policy chosen using Actual Game Information**	**Policy chosen using Rotated Game Information**
P1	{"**current player**": "p1", "**cards**": "2H, 3C", "**smallblind**": "p3", "**seating**": ["p1", "p2", "p3", "p4", "p5", "p6"], "**action history**": [{"p3": "SMALLBLIND", "p4": "BIGBLIND", "p5": "CALL", "p6": "RAISE 300"}] "**player chips**": [{"p1": 1000, "p2": 1000, "p3": 980, "p4":960, "p5": 960, "p6": 660}]	{"**current player**": "p1", "**cards**": "2H, 3C", "**smallblind**": "p3", "**seating**": ["p1", "p2", "p3", "p4", "p5", "p6"], "**action history**": [{"p3": "SMALLBLIND", "p4": "BIGBLIND", "p5": "CALL", "p6": "RAISE 300"}] "**player chips**": [{"p1": 1000, "p2": 1000, "p3": 980, "p4":960, "p5": 960, "p6": 660}]	Action policy for **p1** + **2H, 3C** card pair + **p3** as small blind	Action policy for **p1** + **2H, 3C** card pair + **p3** as small blind
P2	{"**current player**": "p2", "**cards**": "5C, 6S", "**smallblind**": "p3", "**seating**": ["p1", "p2", "p3", "p4", "p5", "p6"], "**action history**": [{"p3": "SMALLBLIND", "p4": "BIGBLIND", "p5": "CALL", "p6": "RAISE 300", "p1": "CALL"}] "**player chips**": [{"p1": 660, "p2": 1000, "p3": 980, "p4":960, "p5": 960, "p6": 660}]	{"**current player**": "p1", "**cards**": "5C, 6S", "**smallblind**": "p2", "**seating**": ["p6", "p1", "p2", "p3", "p4", "p5"], "**action history**": [{"p2": "SMALLBLIND", "p3": "BIGBLIND", "p4": "CALL", "p5": "RAISE 300", "p6": "CALL"}] "**player chips**": [{"p6": 660, "p1": 1000, "p2": 980, "p3":960, "p4": 960, "p5": 660}]	Action policy for **p2** + **5C, 6S** card pair + **p3** as small blind	Action policy for **p1** + **5C, 6S** card pair + **p2** as small blind

P3	{“**current player**”: “p3”, “**cards**”: “AC, 8S”, “**smallblind**”: “p3”, “**seating**”: [“p1”, “p2”, “p3”, “p4”, “p5”, “p6”], “**action history**”: [{“p3”: “SMALLBLIND”, “p4”: “BIGBLIND”, “p5”: “CALL”, “p6”: “RAISE 300”, “p1”: “CALL”, “p2”: “FOLD”}] “**player chips**”: [{“p1”: 660, “p2”: 1000, “p3”: 980, “p4”:960, “p5”: 960, “p6”: 660}]	{“**current player**”: “p1”, “**cards**”: “AC, 8S”, “**smallblind**”: “p1”, “**seating**”: [“p5”, “p6”, “p1”, “p2”, “p3”, “p4”], “**action history**”: [{“p1”: “SMALLBLIND”, “p2”: “BIGBLIND”, “p3”: “CALL”, “p4”: “RAISE 300”, “p5”: “CALL”, “p6”: “FOLD”}] “**player chips**”: [{“p5”: 660, “p6”: 1000, “p1”: 980, “p2”:960, “p3”: 960, “p4”: 660}]	Action policy for **p3** + **AC, 8S** card pair + **p3** as small blind	Action policy for **p1** + **AC, 8S** card pair + **p1** as small blind
P4	{“**current player**”: “p4”, “**cards**”: “TH, 2S”, “**smallblind**”: “p3”, “**seating**”: [“p1”, “p2”, “p3”, “p4”, “p5”, “p6”], “**action history**”: [{“p3”: “SMALLBLIND”, “p4”: “BIGBLIND”, “p5”: “CALL”, “p6”: “RAISE 300”, “p1”: “CALL”, “p2”: “FOLD”, “p3”: “RAISE 100”}] “**player chips**”: [{“p1”: 660, “p2”: 1000, “p3”: 880, “p4”:960, “p5”: 960, “p6”: 660}]	{“**current player**”: “p1”, “**cards**”: “TH, 2S”, “**smallblind**”: “p6”, “**seating**”: [“p4”, “p5”, “p6”, “p1”, “p2”, “p3”], “**action history**”: [{“p6”: “SMALLBLIND”, “p1”: “BIGBLIND”, “p2”: “CALL”, “p3”: “RAISE 300”, “p4”: “CALL”, “p5”: “FOLD”, “p6”: “RAISE 100”}] “**player chips**”: [{“p4”: 660, “p5”: 1000, “p6”: 880, “p1”:960, “p2”: 960, “p3”: 660}]	Action policy for **p4** + **TH, 2S** card pair + **p3** as small blind	Action policy for **p1** + **TH, 2S** card pair + **p6** as small blind
P5	{“**current player**”: “p5”, “**cards**”: “TC, QS”, “**smallblind**”: “p3”, “**seating**”: [“p1”, “p2”, “p3”, “p4”, “p5”, “p6”], “**action history**”: [{“p3”: “SMALLBLIND”, “p4”: “BIGBLIND”, “p5”: “CALL”, “p6”: “RAISE 300”, “p1”: “CALL”, “p2”: “FOLD”, “p3”: “RAISE 100”, “p4”: “FOLD”}] “**player chips**”: [{“p1”: 700, “p2”: 1000, “p3”: 880, “p4”:960, “p5”: 960, “p6”: 700}]	{“**current player**”: “p1”, “**cards**”: “TC, QS”, “**smallblind**”: “p5”, “**seating**”: [“p3”, “p4”, “p5”, “p6”, “p1”, “p2”], “**action history**”: [{“p5”: “SMALLBLIND”, “p6”: “BIGBLIND”, “p1”: “CALL”, “p2”: “RAISE 300”, “p3”: “CALL”, “p4”: “FOLD”, “p5”: “RAISE 100”, “p6”: “FOLD”}] “**player chips**”: [{“p3”: 700, “p4”: 1000, “p5”: 880, “p6”:960, “p1”: 960, “p2”: 700}]	Action policy for **p5** + **TC, QS** card pair + **p3** as small blind	Action policy for **p1** + **TC, QS** card pair + **p5** as small blind
P6	{“**current player**”: “p6”, “**cards**”: “JC, KS”, “**smallblind**”: “p3”, “**seating**”: [“p1”, “p2”, “p3”, “p4”, “p5”, “p6”], “**action history**”: [{“p3”: “SMALLBLIND”, “p4”: “BIGBLIND”, “p5”: “CALL”, “p6”: “RAISE 300”, “p1”: “CALL”, “p2”: “FOLD”, “p3”: “RAISE 100”, “p4”: “FOLD”, “p5”: “CALL”}]	{“**current player**”: “p1”, “**cards**”: “JC, KS”, “**smallblind**”: “p4”, “**seating**”: [“p2”, “p3”, “p4”, “p5”, “p6”, “p1”], “**action history**”: [{“p4”: “SMALLBLIND”, “p5”: “BIGBLIND”, “p6”: “CALL”, “p1”: “RAISE 300”, “p2”: “CALL”, “p3”: “FOLD”, “p4”: “RAISE 100”, “p5”: “FOLD”, “p6”: “CALL”}] “**player chips**”: [{“p2”: 660, “p3”: 1000, “p4”: 880, “p5”:960, “p6”: 560, “p1”: 660}]	Action policy for **p6** + **JC, KS** card pair + **p3** as small blind	Action policy for **p1** + **JC, KS** card pair + **p4** as small blind

	"**player chips**": [{"p1": 660, "p2": 1000, "p3": 880, "p4":960, "p5": 560, "p6": 660}]			

As it can be observed from the above table of examples, after rotating the information, each of the agents playing as each of the players accesses the policy associated with **p1** alone. Thus, the rotation algorithm helped in reducing the number of action policies by a factor of 6 from 47,736 to 7956, since the action policies for **p1** alone are required. Adding to this, the rotation algorithm also enables faster learning since all 6 agents update the action policies associated with **p1**.

INFORMATION ABSTRACTION ALGORITHM: After reducing the number of action policies by a factor of 6, we wanted to look for possibilities to reduce them even more, since exploring 7956 different start states and their corresponding children enough number of times to come up with optimal action policies would take too long. Inspired by prior works in poker which focused on information abstraction and with a belief that in the first betting stage value of the card alone is sufficient to make decent decisions, we abstracted the distinct start states by ignoring the suit of the pair of cards held. This way the total combinations of pairs of cards came down from $^{52}C_2$ to $^{13}C_2 + 13$. $^{13}C_2$ for all the card value combinations and the extra 13 for the 13 same value pairs that could be held (eg. Pair of aces A, A). With this abstraction step we brought down the total number of discrete action policies from 7956 to 546. For the subsequent information states encountered in the **Flop, Turn and River** stages of betting, instead of having individual states for each of the combination of pair of cards held and the opened community cards, we utilized **expected hand strength value** of the combination as a metric for bucketing together strategically similar information states. This way we restricted all the possible information states in the subsequent betting stages to 190 buckets which are representative of the information states.

ACTION ABSTRACTION ALGORITHM: In Texas no limit hold 'em Poker, a player can **RAISE** any value between the minimum raise value and maximum raise value with single dollar increments. Apart from these raise values, there are **FOLD** and **CALL** actions as well. For instance, if the minimum raise value is $200 and maximum raise value is $9600, then there are 9400 possible raise values. With these many possible actions, as the round progresses the number of child nodes possible from these transitions would increase drastically and exploring all of them enough number of times wouldn't be viable. In order to avoid such a scenario, we have abstracted the raise values possible at any given state as a proportion of the pot value into 9 values for the **preflop** betting stage, 7 for **flop** and 5 for **turn** and **river** stages. For Preflop the raise values are a proportion of pot starting from 0.25 to 2.0 with 0.25 increments and **all-in** (betting the entire amount a player has). For Flop the raise values are a proportion of pot starting from 0.75 to 2.0 with 0.25 increments and **all-in.** For Turn and River the raise values are a proportion of pot starting from 0.5 to 2.0 with 0.5 increments and **all-in.** This way we limited the possible raise values at any given state to under 9, thus limiting the number of child nodes possible based on the actions chosen.

Using the above algorithms for abstractions at various levels, the self-play set up was done using the poker engine. For each of the 6 players an instance of the Poker Agent was initialized. To monitor the progress, we ran each training phase with 30,000 games and 6 rounds in each of the game. Each game began with all the 6 players having $10,000. At the beginning of each round, for each of the players the relevant action policy is loaded onto the memory based on the rotated game information, if the action policy is not present, a new empty action policy is initialized. When a player has to choose an action, this action policy (which is a nested dictionary) is traversed based on actions history of prior actions taken by players in the rotated game information. If any of the actions in the actions' history of the rotated game information are not present in the action policy, then they are added to the action policy and the subsequent actions in the action history are added as children to this new entry. Once the action policy is traversed based on the actions' history from rotated game information and a state of **p1** is reached, the valid legal actions at that given state and already explored actions, if any, at that specific state are taken into consideration and used with the **Action Picker** algorithm to decide on an action at that state. For the action choosing process in the **Action Picker** algorithm, we have

employed the famous UCT (Upper Confidence Bound 1 for Trees) formula to pick the best action. The rewards for the

$$bestAction \leftarrow argmax_i[r_i + (c \times \sqrt{\frac{\log N_i}{n_i}})]$$

r_i = reward for choosing action i

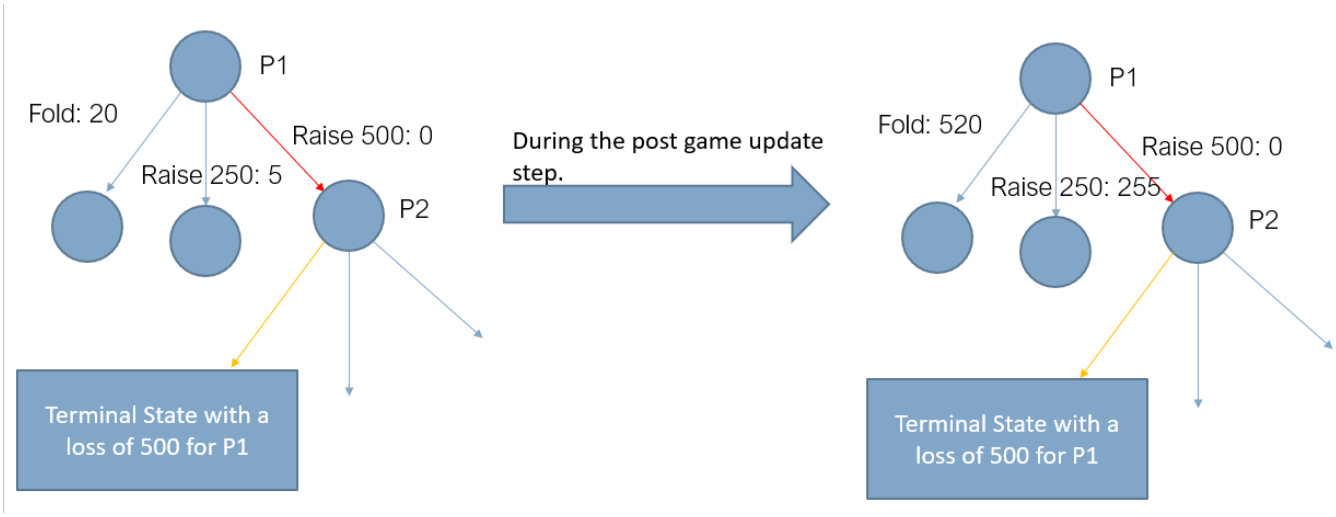
c = Balancing factor between exploration and exploitation

N_i = Number of times the Parent state of Action i has been explored

n_i = Number of times Action i has been explored

actions are the regret values associated with that action from the prior experiences. If an action hasn't been explored yet the reward value would be 0, and if a state is being explored for the first time the ($\log N_i/n_i$) value in the formula is set to infinity. We have also chosen a very high c value of 1,000,000 so that less explored actions are given more preference. If there are multiple actions which that have the same value being returned by this formula, we chose to break the tie by picking randomly from those actions, instead of the first action from them. The states are traversed, updated and explored in a similar manner for the subsequent betting stages in a round of poker. Keeping the memory limitations in mind, a new state is initialized only when it is explored for the first time and an action is initialized with a regret value of 0 for the player **p1** only when the action is explored for the first time. Once a round of poker is completed, the results are returned as the updated amounts which each of the player has. These results are used to compute net profit or loss for each of the players and that value is used with a form of regret minimization algorithm explained below to update the regret values for the explored states and corresponding actions in the round.

REGRET UPDATING ALGORITHM: For each of the players this net profit or loss referenced as **net value** from here on is used along with the actions' history from the rotated game information for updating the regrets. The main idea behind any regret minimization algorithm is that eventually after enough exploration, lesser rewarding actions would have lesser positive regrets and promising actions would have higher positive regrets and based on these values best possible actions could be chosen. Using the actions' history from the rotated game information the action policy is traversed and at each state of **p1** where an action has been chosen, the expected net values for all the other available actions at that state are computed using an approximate net value function and the expected net values are added to the existing regret values present for each of those actions. Below is a hypothetical example for a clearer understanding of the regret updating algorithm. The number of times the agent already explored FOLD, RAISE 250 and RAISE 500 in this given state are 2, 1 and 0 respectively.



In the above example of a two-player game, for the player p1, the actions and their corresponding regret values are present. In the above case, p1 has already explored **FOLD** and **RAISE 250** actions and hence chose **RAISE 500** in the above state. Post that p2 chose another action and a terminal state is reached with p1 losing $500. When the actions'

history along with the net value of -500 is passed to our regret updating algorithm, the action policy is traversed to reach the given state of p1 using the actions' history from the rotated game information. At that state for the remaining actions, which are **FOLD** and **RAISE 250** in this case, the expected net values that p1 would have received had p1 chosen these actions instead of **RAISE 500** and p2 playing the same move is computed from the net value. These expected net values are then added to existing regret values. Had the terminal state resulted in a net value of +500, the regrets for FOLD and RAISE 250 would have been updated to -480, -245 respectively. The intuition behind this is that, with enough exploration of the state present in the above example, the action which is the best action at that given state based on the information which p1 has would end up with a high positive regret value and the player would eventually prefer it to other actions available at that state. As a first step we tried maintaining the moving averages of rewards for each of the actions as they were explored. However, even after 200,000 games, the agents didn't learn much when moving averages of rewards were being used. We explored other options which were available that would help in learning the optimal policies faster and found regret minimization algorithm to be promising for our use case and hence proceeded with it.

1 Self Play Training

Algorithm 1 High Level Overview Of The Training Algorithm

```
Begin
for k = 0, 1, 2, ....g do                                   ▷ g is number of games
    Start Poker Game
    for round = preflop, flop, turn, river do               ▷ rounds in one game
        for i = 0, 1, 2, 3, 4, 5 do                         ▷ for each of the players in the game
            rgd_i ←GetRotatedGameInfo(gd,i)                 ▷ get the rotated game data
            if round = preflop then
                sb_i ←GetSmallBlind(rgd_i)                  ▷ get small blind w.r.to player i
                p_i ←GetAbstractedPolicy(h_i, sb_i)         ▷ get policy for player i
            p_i ←AddNewStates(rgd_i, p_i)                   ▷ Add new states based on game so far
            e_i ←GetExploredActions(rgd_i, p_i)             ▷ Get the actions explored
            b_i ←BestAction(e_i)                            ▷ Choose the best action
            return b_i
    End Poker Game And Get Results gr
    for i = 0, 1, 2, 3, 4, 5 do                             ▷ for each of the players in the game
        p_i ←UpdateRegrets(gr,p_i)                          ▷ Update rewards based on game result
        SavePolicy(p_i)                                     ▷ Save the updated policy
End
```

2 Real Time Game Play

Algorithm 2 High Level Overview Of The Real Time Game Play Algorithm

```
Begin
Start Poker Game
i ←GetPlayerId(gd,name)                                     ▷ Get the player Id
for round = preflop, flop, turn, river do                   ▷ rounds in one game
    rgd_i ←GetRotatedGameInfo(gd,i)                         ▷ get the rotated game data
    if round = preflop then
        sb_i ←GetSmallBlind(rgd_i)                          ▷ get small blind w.r.to player i
        p_i ←GetAbstractedPolicy(h_i, sb_i)                 ▷ get policy for player i
    e_i ←GetExploredActions(rgd_i, p_i)                     ▷ Get the actions explored
    b_i ←BestAction(e_i)                                    ▷ Choose the best action
    return b_i
End
```

Using the above-mentioned self-play setup and algorithms, we generated two versions of the action policies. The first version is after 300,000 self-play games and the second version is after 500,000 games. The next component needed was the set of algorithms needed for utilizing these action policies in real-time gameplay. The rotation, information abstraction and action abstraction algorithms were reused for the agent to get the necessary action policy. Post that similar to the training stage the action policies were traversed using the actions' history from the rotated game information. During this traversal, if an opponent's action wasn't present in the action policy, we had to map it to the closest action present in the action policy and continue the traversal. For this, we made use of **Pseudo Harmonic Action Mapping Algorithm.**

Pseudo Harmonic Action Mapping Algorithm: This algorithm was first introduced in the paper 'Action Translation in Extensive-Form Games with Large Action Spaces: Axioms, Paradoxes, and the Pseudo-Harmonic Mapping' by Ganzfried, S., & Sandholm, T. Using the below formula this algorithm helps in mapping the opponents' actions not present in the action policy to the closest actions present in the policy which closely resembles the actions actually chosen by the opponents. This is achieved by using the value calculated using the below-mentioned formula.

$$f_{A,B}(x) = \frac{(B-x)(1+A)}{(B-A)(1+x)}.$$

A ≤ x ≤ B

In this formula, **x** is the actual value of the action chosen by the opponent. **A** is the largest value less than **x** present in the action policy and B is the smallest value greater than **x** present in the action policy. If the value of $f(x)$ in the above formula is less than or equal to 0.5 then action **A** is chosen and if the value is greater than 0.5 then action **B** is chosen.

For example, if the opponent chooses to **RAISE** by 450, and we don't have that action in the action policy. But we do have **RAISE** by 400 and **RAISE** by 500 actions which are in the closest proximity of **RAISE** by 450 of the opponent in the action policy. The $f(x)$ value is computed with **x** as 450, **A** as 400 and **B** as 500. For this example, we get 0.445 as the value after calculation and based on that value action **A** is chosen which is **RAISE** by 400 and the traversal is continued.

After traversing and reaching the current state of the player in the action policy the **Action Picker** algorithm was used with a very low c value (c=0.000001) to pick the most promising action from the actions that are available at that state.

During real-time gameplay, if a state which the player hasn't explored yet is reached, Monte Carlo simulation of 400 games with the current game information is run and based on the win rate from the simulations an action is chosen. If the win rate is less than 50% then the player chooses to **CALL** by 0 and if **CALL** by 0 is not available, then the player chooses to **FOLD**. If the win rate is greater than 85% **RAISE** by all-in is chosen and if it is between 75% to 85% **RAISE** by minimum possible amount is chosen and when it is between 50% and 75% **CALL** is chosen.

EVALUATION

For evaluating the performance of our final Poker Agent (with action policies after training for 500,000 games), we tested it against:

1.) A previous version of itself after 300,000 games.
2.) Other Poker Agents available online which used **pypokerengine** as base platform. They were:
    - Caller Agent: An agent which always calls, irrespective of cards dealt.
    - MCTS Agent1: An Agent which performs 1400 Monte Carlo simulations before taking an action.
    - Genetic Agent1: Variant 1 of an Agent built using Genetic Algorithm.
    - Genetic Agent2: Variant 2 of an Agent built using Genetic Algorithm.
    - MCTS Agent2: An Agent which performs 3000 Monte Carlo simulations before taking an action.
3.) Poker Agents built by the other team comprising of Luke, Chujin and Robert.

Against the previous version of the Poker Agent, we ran 1000 games of 10 rounds each, so that the agents would effectively play 10,000 hands. At the beginning of each round each of the players started with $10,000. The final results after running these games are in the below table.

Table 3. Trial V2 vs V1

Agent	Our Agent V1	Our Agent V1	Our Agent V2	Our Agent V2	Our Agent V1	Our Agent V2
Profit/Loss ($)	-184,302	-397,265	+694,680	+196,737	-608,800	+298,950

The final version of our Poker Agent consistently outperformed the prior version and made significant profits over the 10,000 hands.

Against the other agents which we found online, to have concrete results about the profitability of our agent we ran 3 trials with 1000 games of 10 rounds each. The below tables contain the final results after each of those trials.

Table 4. Trial 1

Agent	Caller Agent	MCTS Agent1	Genetic Agent1	Genetic Agent2	Our Agent V2	MCTS Agent2
Profit/Loss ($)	+7,597	-65,000	-1,513,383	-2,252,414	+3,781,385	+41,815

Table 5. Trial 2

Agent	Our Agent V2	MCTS Agent1	Genetic Agent1	Genetic Agent2	Caller Agent	MCTS Agent2
Profit/Loss ($)	+2,022,660	-149,800	-824,240	-513,680	-482,475	-52,465

Table 6. Trial 3

Agent	MCTS Agent2	MCTS Agent1	Genetic Agent1	Genetic Agent2	Caller Agent	Our Agent V2
Profit/Loss ($)	-43,700	-146,200	-310,880	+194,800	-1,588,365	+1,894,345

In the above tables 4,5 and 6 our Agent has been highlighted in black and it can be observed that across the three trials conducted our agent ended up making the maximum profit. More importantly, it has been consistently profitable as well.

We also conducted a trial of 500 games with 5 rounds in each of the game with the poker agents built by the other team comprising of Luke, Chujin and Robert. The results of the trial are in the below table. Due to technical and time limitations, we were only able to run a 5-player poker game contrary to the 6-player poker game on which our agent was trained on.

Table 7. Trial vs Other Poker Agents built by Luke, Chujin and Robert

Agent	Our Agent1	Our Agent2	Our Agent3	Other Agent1	Other Agent2
Profit/Loss ($)	+105,800.0	+105,670.0	+85,850.0	-363,860.0	+66,540.0

As it can be observed from table 7, our agent ended up being profitable as well as made the maximum profit. To our surprise even though our agent wasn't trained explicitly for playing 5-player poker, it ended up using the action policies fairly well for playing the 5-player poker game. After going through the action policies to figure out how our agent was able to do so, we found out that, during the training since each game had multiple rounds, some players ended up losing all the money in the earlier rounds itself and this resulted in the agents playing 5, 4, 3 and 2 player rounds as well with the remaining active players. This was how it had policies for handling 5-player game as well.

CONCLUSIONS AND FUTURE WORK

In the limited amount of time we had, the computational resources available to us and the limited domain knowledge of Poker, we were successful in building a Poker Agent which came up with its action policies through self-play alone and was able to use them to its advantage during real-time gameplay.

The following points would summarize the unique aspects of our work:

- We were successful at leveraging various algorithms during the self-play to utilize all the experiences of each of the 6 agents to the fullest in building the common action policies.
- Action policies built are position agnostic, meaning the agent can sit in any position and utilize them.
- With an intuition that with enough number of games, strategically similar decisions of opponents based on their cards could be successfully captured in the action policies, we safely avoided maintaining belief distribution of possible cards which opponents might have.

The following areas are something which we could work on as extensions to our current work:

- Implementing real-time 1ply depth limited search while choosing an action rather than choosing based on the information in the current state alone.
- See how holding the belief distribution about cards of other players would alter the performance of our agent.
- Adding the total number of games to be played as a factor for to the agent's decision-making system.

ACKNOWLEDGEMENTS

We collaborated and worked on each of the components together.

PyPokerEngine was the library that we used in our project for setting up the poker game environment required for building our Poker Agent.

PyPokerEngine link: https://ishikota.github.io/PyPokerEngine/

REFERENCES

1.) Brown, Noam, and Tuomas Sandholm. 2018. "Superhuman AI for Heads-up No-Limit Poker: Libratus Beats Top Professionals." Science. American Association for the Advancement of Science. https://science.sciencemag.org/content/359/6374/418.
2.) Brown, Noam, and Tuomas Sandholm. 2019. "Superhuman AI for Multiplayer Poker." Science. American Association for the Advancement of Science. https://science.sciencemag.org/content/365/6456/885.
3.) Ganzfried, S., & Sandholm, T. (2013). Action Translation in Extensive-Form Games with Large Action Spaces: Axioms, Paradoxes, and the Pseudo-Harmonic Mapping. IJCAI. https://www.semanticscholar.org/paper/Action-Translation-in-Extensive-Form-Games-with-and-Ganzfried-Sandholm/cf3f57218fdeb247baef236e36455c708bab0463
4.) https://ishikota.github.io/PyPokerEngine/ Poker Engine used.

APPENDICES

Zip file of our code along with the action policies has been uploaded.