No Limit Texas Hold'em Poker Learning AI

Authors: Erik Chan (ejc233), Gary Gao (gg392)

Project Report Paper for CS 4701
Artificial Intelligence Practicum
Spring 2017

Abstract

"Poker is a scholastic card game involving a large degree of chance so it may be difficult to evaluate how good a Poker Bot can perform. For this project, we created an AI that could infer on average how likely it would fare and also infer the personality of the player it was playing against."

Keywords

Artificial Intelligence, Training, Poker

General Work

Proposal

We planned to implement a game playing program in Java for the game no-limit Texas Hold'em Poker. Among the core functionalities of the system would be the distribution of playing cards, ability to bet a certain amount of currency, opportunity to either call, fold or raise, and the determination of the highest poker card ranking. We would build classes for the game itself, each poker hand, each card, each ranking, and each deck. We would support human playability and implement a set of A.I. players with different levels of intelligence and varied strategies.

We originally planned to have human players test our A.I. for evaluation; a smarter A.I. would be able to win more often. We later decided against this because Poker comes with unpredictability and randomness that cannot be thoroughly tested given the inconsistency of human play. We found it more effective to have our A.I.'s evaluate each other. We also tried to include more tree searching and looking at game states, but we could not figure out a way to make tree searches work effectively, so we shifted our project toward a learning approach.

Our A.I. players would be smart about their plays. Each A.I. would look at the hand it has and determine a value based on its starting hand. It would look at how many of the other players have folded and how much money it had relative to the other players. The A.I. would also look at the size of the bets the other players had bet and the possible cards that are available based on the starting sequence of cards on the table.

Every hand would have a score inherent in the quality of the hand. Each A.I. would analyze its surroundings to modify its score. Based on the expected scores of its opponents to its own hand, it would make a decision between calling, folding, or raising the bet.

Goals Accomplished

We first formulated the problem so that it could be approached computationally. We next developed graphics so that the game could be visualized. Finally, we designed a set of computer players whose intelligence could later be evaluated. Our work reflected our goals quite closely.

Backend of Gameplay

We initially picked to develop a Poker program because neither of us had ample experience playing Poker. We figured that the project would help us learn the machinations of Poker while also tying into Artificial Intelligence. We quickly realized that Poker would be more challenging than we anticipated. Unlike smaller games like Checkers or Othello, Poker has a large set of states represented by variable money, bet, and pot amounts. Even though there is a set number of 52 cards, not all of the

cards are at play at one time; even those in play may be hidden from all but one player. It was difficult to build a truly optimal artificial intelligence.

Nevertheless, we opted to continue because we felt there was much to learn about making rational decisions on incomplete data. We settled on the Texas Hold'em variant of Poker because it is the most popular Poker variant, and the inclusion of community cards allows computer players to obtain about half of the information needed to infer opponents' cards. We also chose the no-limit variation because we felt a very confident player should be allowed to bet their entire hand.

We wrote five main Java classes: Game, Player, Deck, Card, and Hand. The Game class, also our top-level class, includes the information introduced at the start of every game: the pot, the blinds, the current bet, and the deck. The Player class includes information specific to a single player, such as the cards in their hand, their amount of money, whether they have folded or called, whether they are bankrupt, and their last decision. This class can be extended in subclasses with personality traits. The Deck class includes the cards not yet drawn or on the table. The Card class includes information about a card's suit and rank. The Hand class includes the set of cards in a single player's hand along with the list of community cards on the table.

Each player's hand also has a specific score calculation tied to the hand's list of cards. A player's hand's score plays an important role in determining his decision. The score is calculated such that a stronger poker hand will always have a higher score than a weaker poker hand. We break ties from poker hands with the same hand type by the ranks of the cards in them. Ties beyond that point have identical scores. The scores dynamically update with the arrival of community cards. Each player can view only his score until the round has concluded. The active player with the highest score for his hand wins the round and the pot (shared in the event of a tied score).

The game itself can be customized to be played either by a human or by a simulated computer player. One can specify the number of players, the number of rounds in a game, and the number of games played. One can also choose to rapidly simulate a set of games or to display a game with paced, real-time animations.

Graphical User Interface

We built the GUI less for evaluation of the strength of our artificial intelligence than to test the playability of the game we created. Both of us had never played poker before this and we figured that if we could visualize the game, we could better check that it was working appropriately with the official Texas-Hold'em rules.

The interface, as seen in Fig. 1, represents the current state of the game, including all players' money and bets, the community cards, the pot, the current bet, the small blind, and the big blind. Each active player is only able to view his own hand type and hand score. At each player's turn, he has three operators that he can call to change the state: call, fold, or raise. At the end of every round, the set of cards and score of each player are revealed to everybody before the pot is allocated to the winners.

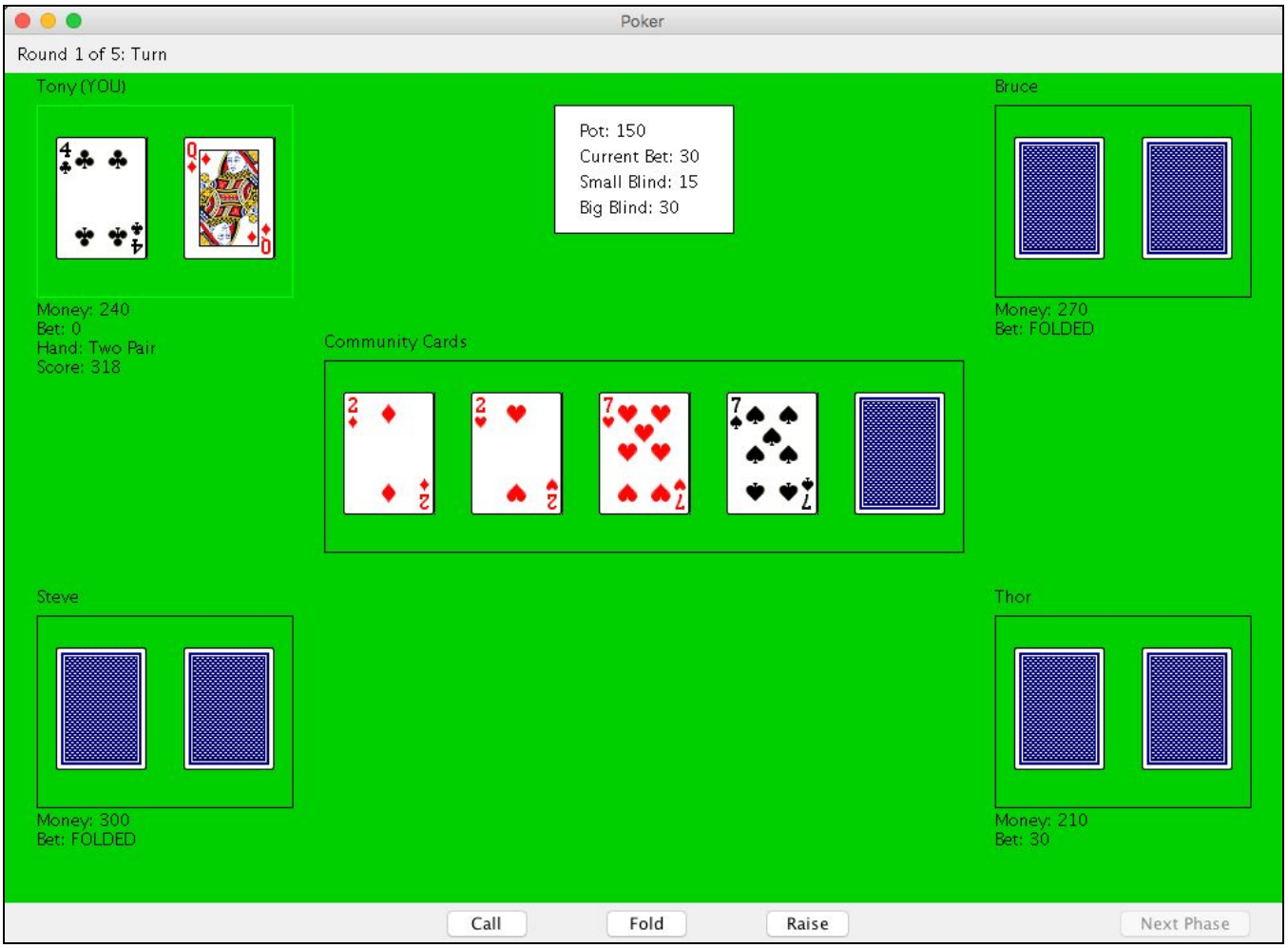

Fig. 1. The graphical user interface of an active poker game

Computer Players

Although we stuck to our goal of building a version of Poker, we scaled down our project from having a smart computer player, hereafter referred to as CPU, adapt based on its plays to having a smart CPU make inferences about its opponent. We designed the game so that each CPU makes one of three decisions: call, fold, or raise the standing bet by a certain amount. Ideally, the CPU should respond to the actions of 4 players; however, we found that addressing multiple opponents was an unreasonable first step. We reduced our program to a game between two players so that we found it better to evaluate our CPU's against each other.

We hoped to create a CPU that could read different personality traits, such as those in Table 1, from its opponent after learning from a small set of games. We created multiple CPU's to emulate these varied quirks, as seen in Table 2. We tested our smart CPU against personality traits such as playing more protectively, playing more aggressively when the chance of winning is higher, playing more cautiously when the other person plays more aggressively, and taking more chances to stay active even if it requires bluffing. Our smart CPU tried to infer personality traits of its opponent while trying to find an optimal strategy to handle those traits.

Table 1. Personality traits common to Poker Players

Personality Trait	**Description**
Playing more protectively	Making smaller raises or less calls when not confident of own chances to win.
Playing more aggressively	Making higher bets with better hands and higher confidence of winning. Taking advantage of when own hand is extremely good.
Playing more cautiously	Taking less risks when the opponent displays more confidence: i.e. when opponent makes higher and more consistent raises. Takes less risks when more confident of opponent's chance to win.
Playing more deceptively	Making moves to stay in although there is a chance of losing given there is some chance to win, and scaring opponent into folding or making bad judgements. Bluffing.
Playing randomly	Making moves not entirely based on strategy.

Table 2. CPU's we built and their hard-coded personality traits

Name	**Strategy**
CallingPlayer	This computer player plays normally, but risks calling in with a hand that CallingPlayer doesn't think is too good to gain money. If CallingPlayer's hand is decent then CallingPlayer might stay in knowing that he would be able to gain money if the opponent's hand is also not as good.
ConfidentPlayer	This computer player plays relatively safely with bad hands and knows when to fold. With better hands, ConfidentPlayer plays with more confidence and goes more on the offense to win more money. Depending on how good its hands is, ConfidentPlayer plays more confidently and aggressively.
DefensivePlayer	This computer player plays safe for the entire game without taking too many risks and doesn't try to play too aggressively.
RationalPlayer	This computer player plays heavily basing its choices on its hand strength and how strong the opponent's hand strength likely is. It can choose to fold, call, and raise more rationally.
RandomPlayer	This computer player sometimes plays moves that are not entirely based on strategy and more on random number generation.
ScaredPlayer	This computer player plays normally, but it feels threatened on how confident the opponent player is. If the opponent player plays really aggressively and confidently, then the ScaredPlayer will increasingly decide to fold before it loses too much. ScaredPlayer will likely call less when its opponent is more aggressive.
SmartPlayer	This computer player starts playing normally, but learns to infer the strategy of the other computer players in a number of training games before it can start to counter the opponent effectively.

Our smart CPU, hereafter referred to as SmartPlayer, spends a number of games just playing games safely while collecting information about its opponent. Our algorithm for SmartPlayer is based off concepts from the Kriegspiel chess game in that, in Poker, one player knows only his own hand and the cards on the river. From this knowledge, the player knows all of the cards remaining in the deck that are not from his own hand or from the river. Therefore, this player can calculate how well the opponent could theoretically perform if the player acted like he knew the opponent's hand by guessing every possibility of pairs possible remaining in the deck.

$$\frac{1}{N} * \sum_{i=1}^{N} s_i$$

Given that the player knows on average how the opponent is going to do, he can decide how he plans to proceed given how strong the difference is between their scores. If the player has a much higher score than the average of the opponent, then he will raise the current bet by more money. If the player has a much worse score than the average of the opponent's scores, the player will likely fold.

SmartPlayer learns the patterns of an opponent by playing games against the unknown player and keeping track of data on the opponent's decisions. Collected information includes how the opponent responds to raises, what kind of decisions the opponent made given the score of their hand, and how often the opponent folds.

SmartPlayer infers strategies related to three specific players: CallingPlayer, ConfidentPlayer, and ScaredPlayer, as described in Table 2, from suspicious actions. After the set amount of training games, SmartPlayer will play a new strategy specifically targeted to exploit flaws in its opponent's strategy. In theory, the SmartPlayer could be expanded to infer even more personality traits and employ even more ways to engage them.

Failed Attempts

We explored other approaches we learned in this class such as a game tree search approach with a minimax algorithm. The first difficulty we encountered was our starting with a four person game. Because we had only explored the two-player variant of minimax, we scaled down the game to start with two players. With reference to the textbook, we tried to implement an averaging over clairvoyance strategy. We assumed that the opponent was able to play every possible hand remaining in the deck. Then, we tried to get the max of the values weighted according to their probabilities.

$$argmax_a \frac{1}{N} \sum_{i=1}^{N} MINIMAX(\, RESULT(s_i, a)\,)$$

After looking at the Kuhn Poker example with minimax, we found it really hard to use minimax on a full deck version of no-limit Texas Hold'em Poker. Not only does Texas Hold'em with a full deck have more missing and incomplete information, but it also has more possible states and more considerations for minimax. Even with a Monte Carlo approximation, this seemed needlessly strenuous to implement to create an effective CPU.

Given that the cards in the hands of the players are not revealed, it became increasingly difficult to implement minimax. Even though we thought found out a way to represent Poker as a zero-sum game, we could not fully flesh out the CPU to be effective for our evaluation. The averaging over clairvoyance strategy also failed for several reasons. It assumed the opponent could see its cards, and it never hid information because it assumed that the information was already available. This strategy was missing an entire portion essential to the game of poker in interpreting facial expressions, body movements, and sending [sometimes false] signals. A computer program cannot see this, but it can still see some weird regularities that help predict the opponent's hand and their personality.

We considered using Nash Equilibriums to help the CPU with bluffs and other personality traits or strategies to help it with winning. However, there were too many poker strategies for us to determine a suitable strategy to compute a Nash Equilibrium in this undergraduate level project.

We also initially underestimated the difficulty of poker because we were both not well experienced with everything involved in the game. Given the odd nature of poker, we should have expected that some of our attempts would have failed. However, we were able to inch towards more favorable results with a more machine-learning-based approach with several weeks of working on the project. If we had more time on this project, we would have liked to explore the topics we failed with more, but we feel confident with the direction our project actually headed.

Artificial Intelligence Concepts

Games

We studied core concepts in gameplay such as state spaces, evaluation functions, and stochastic domains. We aimed for our CPU’s to simulate rational agents.

An agent can perceive the game environment through its sensors and act on the environment through its actuators. We may formulate a problem such that a computer can read its environment, or the state of a game, to make a set of actions to reach a desired end state satisfying a specific goal condition. A computer is also able to assign values to its current state and states it can land on. Some games can have stochastic domains where the same set of parameter values and initial conditions could lead to a set of different outputs. A computer might have to determine the expected value of an operator that has a stochastic element or to run a Monte Carlo simulation to estimate the value of a certain state given the states resulting from each operator.

We bore these concepts into our system by treating the core components of Poker, such as the money, the cards, and the players, as a collective state. Each agent, or player, has a set of three operators, call, fold, or raise, that can change the state of the game. The final goal state is to have the highest score of the active players. A CPU uses as its evaluation function the score of its hand and the amount of money it still has remaining compared to its opponents. The stochastic nature of the game makes the inference of the full state challenging; the opponent’s hands and scores are hidden. Thus,

the CPU may determine the expected value of his score compared to its opponent by calculating the average score of its opponent based on the remaining set of hands its opponent can have.

Machine Learning

As stated in class, “An agent is learning if it improves its performance on future tasks after making observations about the world.” Our smart CPU relies on learning to make its decisions.

Any component of an agent can be improved by learning from data. The improvements depend on the component itself, the prior knowledge of the agent, the representation used for the data and the component, and the feedback available to learn from. In our project, our agent was our smart CPU, SmartPlayer. We tried to improve its ability to play against opponents better by inferring the strategy of its opponent.

We created an agent that can infer relevant properties of its opponent after many simulations of playing the opponent. This allowed the agent to desire more actions than others to counter the opponent’s strategy. Our agent can start with no prior knowledge of its opponent's playstyle and work out the playstyle of the opponent after the simulations. Our agent partakes in supervised learning where, given input variables and output variables, we use an algorithm for the agent to learn the mapping function from the inputs to the outputs. We learned to map the decisions to the strategy of the CPU in what basically amounted to classification learning.

We heavily relied on Machine Learning to help our CPU learn how to infer the personality types of its opponents. Our CPU learned through training set with the results of a certain number of games, a process we honed through exploration.

Exploration

Questions Asked

We posed a series of questions tied to two specific aspects: the formulation of the problem and the capability of the smart CPU.

First, we wondered if our game successfully modeled a simulated poker game. Did we formulate a problem to approach it computationally? Did we create an environment where a smart CPU could make rational decisions?

Second, we questioned how we could determine that our smart CPU, SmartPlayer was actually intelligent. How could we make a Poker AI that can play effectively against any kind of player? Given that poker is stochastic and largely based on random number generation and chance, how could we evaluate if our CPU was doing well? What were the formal rules to draw conclusions?

Evaluation

The first step we took after completing the coding portion of the project was reflecting on the game itself. We still feel that our system suitably emulates no-limit Texas Hold'em Poker. We created an environment where players can intelligently read its surroundings and then make rational decisions that can truly impact the rest of the game. We also set the game so that a player could make guesses based on a specific player's actions at different states of the game. CPUs can adapt despite the random nature of Poker hands and CPU decisions to win more often than not.

Before we could even begin determining the strength of our CPU's, we needed to figure out what settled for intelligence. Would our determination of success be based on number of rounds won, number of games won, or amount of money left at the end of the game?

We ultimately settled on attempting to optimize SmartPlayer's ability to win games. We defined a game so that it had five rounds, each of which had the five traditional phases: Pre-Flop, Flop, Turn, River, and Showdown. Each CPU's success would be determined by how many games of 1000 that it won against its opponent.

Training

SmartPlayer works by reading its opponent's behavior in a set of games before choosing to employ a specific strategy. To determine the number of training games, we had SmartPlayer play sets of 1000 games against each of CallingPlayer, ConfidentPlayer, ScaredPlayer, and RandomPlayer with training game numbers 10, 50, 100, and 150, as seen in Table 3.

While certain traits, like ScaredPlayer, could be read and exploited quickly, as seen in Fig. 2, others were harder to interpret and then address. We decided to calculate the average number of wins for the four opponent strategies.

Table 3. Results of training our CPU with varying lengths

	Training Games					
Situation	**10**	**50**	**100**	**150**	**200**	**250**
Smart vs Calling	636	637	661	651	624	631
Smart vs Confident	593	606	653	604	598	626
Smart vs Scared	993	969	928	902	871	831
Smart vs Random	847	875	873	885	865	874
Average	767.3	771.8	778.8	760.5	739.5	740.5

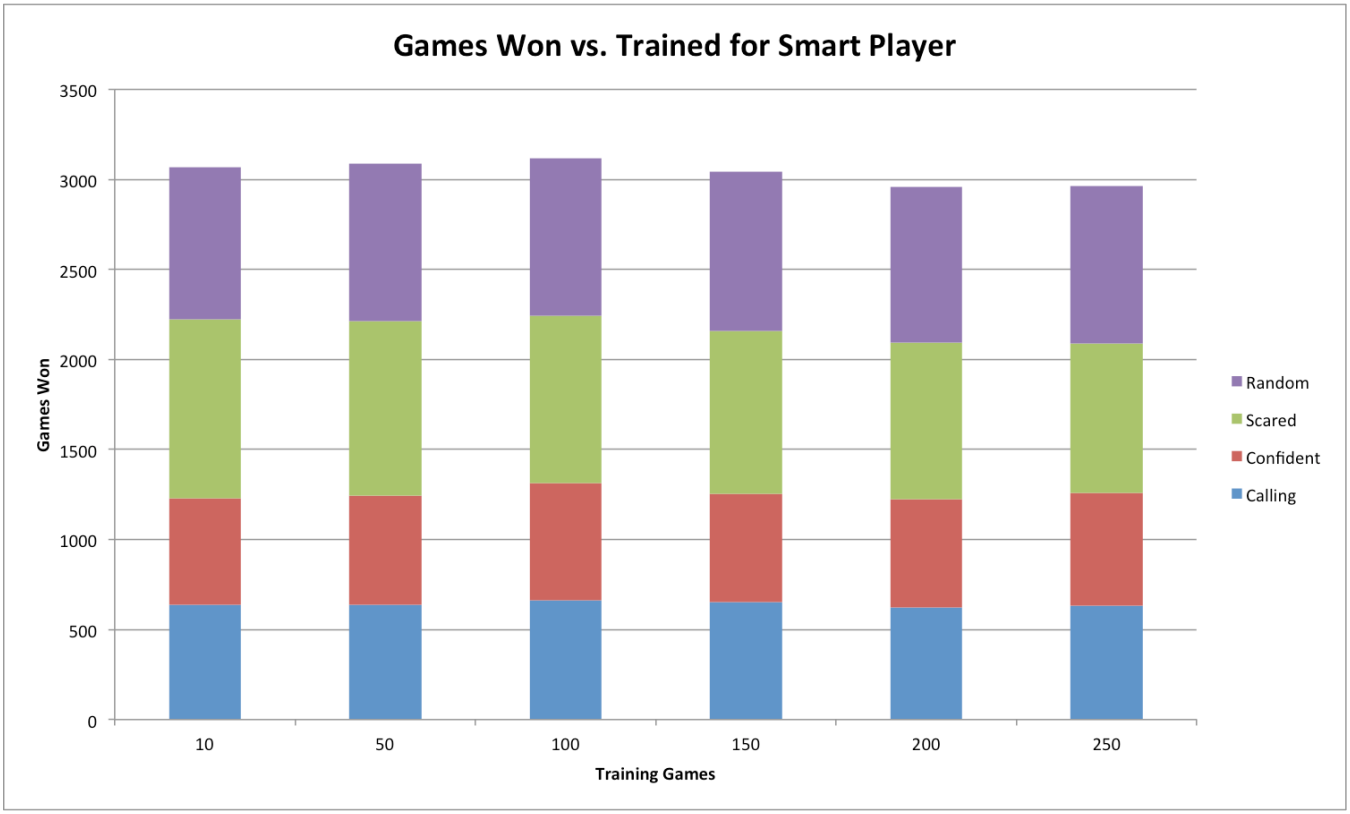

Fig. 2. A comparison of the number of games won compared to the number of training games

Ultimately, we determined that, of the five numbers of training games we investigated, 100 training games out of 1000 total games was the most suitable. We decided to train 100 games for the duration of our evaluation.

Learning

To formally draw conclusions, we chose to analyze how a SmartPlayer fared in its first 100 games compared to its next 900 games. If we observed a noticeable improvement, we could conclude that our SmartPlayer was indeed inferring accurately.

We collected data on how our CPU's fared against each other, as seen in Table 4. We focused particularly on how the SmartPlayer did against the CallingPlayer, ConfidentPlayer, and ScaredPlayer, CPUs whose strategies SmartPlayer was programmed to take on. SmartPlayer had win percentage improvements of 11.22%, 13.13%, and 71.72% respectively from its first 100 games compared to its next 900 games.

Table 4. Average Results and Improvement from Training the Smart A.I.

	First 100					First 1000					
Situation	**P1 Wins**	**P2 Wins**	**Ties**	**Percent P1**	**Percent P2**	**P1 Wins**	**P2 Wins**	**Ties**	**Percent P1**	**Percent P2**	**Improvement**
Calling vs Calling	44	42	14	51.16%	48.84%	421	410	169	50.66%	49.34%	
Confident vs Confident	49	42	9	53.85%	46.15%	468	465	67	50.16%	49.84%	
Defensive vs Defensive	31	56	13	35.63%	64.37%	260	577	163	31.06%	68.94%	
Random vs Random	52	48	0	52.00%	48.00%	558	440	2	55.91%	44.09%	
Rational vs Calling	55	45	0	55.00%	45.00%	621	379	0	62.10%	37.90%	
Rational vs Confident	55	39	6	58.51%	41.49%	544	407	49	57.20%	42.80%	
Rational vs Defensive	58	37	5	61.05%	38.95%	547	364	89	60.04%	39.96%	
Rational vs Random	49	39	12	55.68%	44.32%	546	352	102	60.80%	39.20%	
Rational vs Rational	81	18	1	81.82%	18.18%	817	166	17	83.11%	16.89%	
Rational vs Scared	32	68	0	32.00%	68.00%	361	636	3	36.21%	63.79%	
Scared vs Scared	16	84	0	16.00%	84.00%	150	840	10	15.15%	84.85%	
Smart vs Calling	56	44	0	56.00%	44.00%	661	335	4	66.37%	33.63%	11.22%
Smart vs Confident	53	46	1	53.54%	46.46%	653	308	39	67.95%	32.05%	13.13%
Smart vs Scared	28	71	1	28.28%	71.72%	928	71	1	92.89%	7.11%	71.72%
Smart vs Random	69	27	4	71.88%	28.13%	873	119	8	88.00%	12.00%	17.46%

SmartPlayer's most noticeable improvement came from its performance against ScaredPlayer. While SmartPlayer won just 28.28% of games against ScaredPlayer's 71.88% of wins in the first 100 games, SmartPlayer quickly adapted and won 92.89% of the total 1000 games. In fact, SmartPlayer actually won all 900 games after the first 100. This comes from ScaredPlayer's easily exploitable strategy: in detail, ScaredPlayer always folds when SmartPlayer raises, and otherwise raises the bet by 50 if his score is greater than 200 and calls if not. SmartPlayer can simply raise every round to guarantee a win.

Although SmartPlayer does increase his win percentage against the other opponents, the improvement is not as dramatic as that of ScaredPlayer. While ScaredPlayer can be easily taken advantage of, CallingPlayer and ConfidentPlayer have traits that are harder to address fully. Nevertheless, on average, SmartPlayer does attack them more effectively after learning.

Interestingly, SmartPlayer shows improvements even against players whose traits it is not programmed to infer, as seen in Fig. 3. SmartPlayer actually shows a 17.46% win increase in the latter 900 games against RandomPlayer. We wonder if, with additional coding, this CPU truly improve against any kind of player.

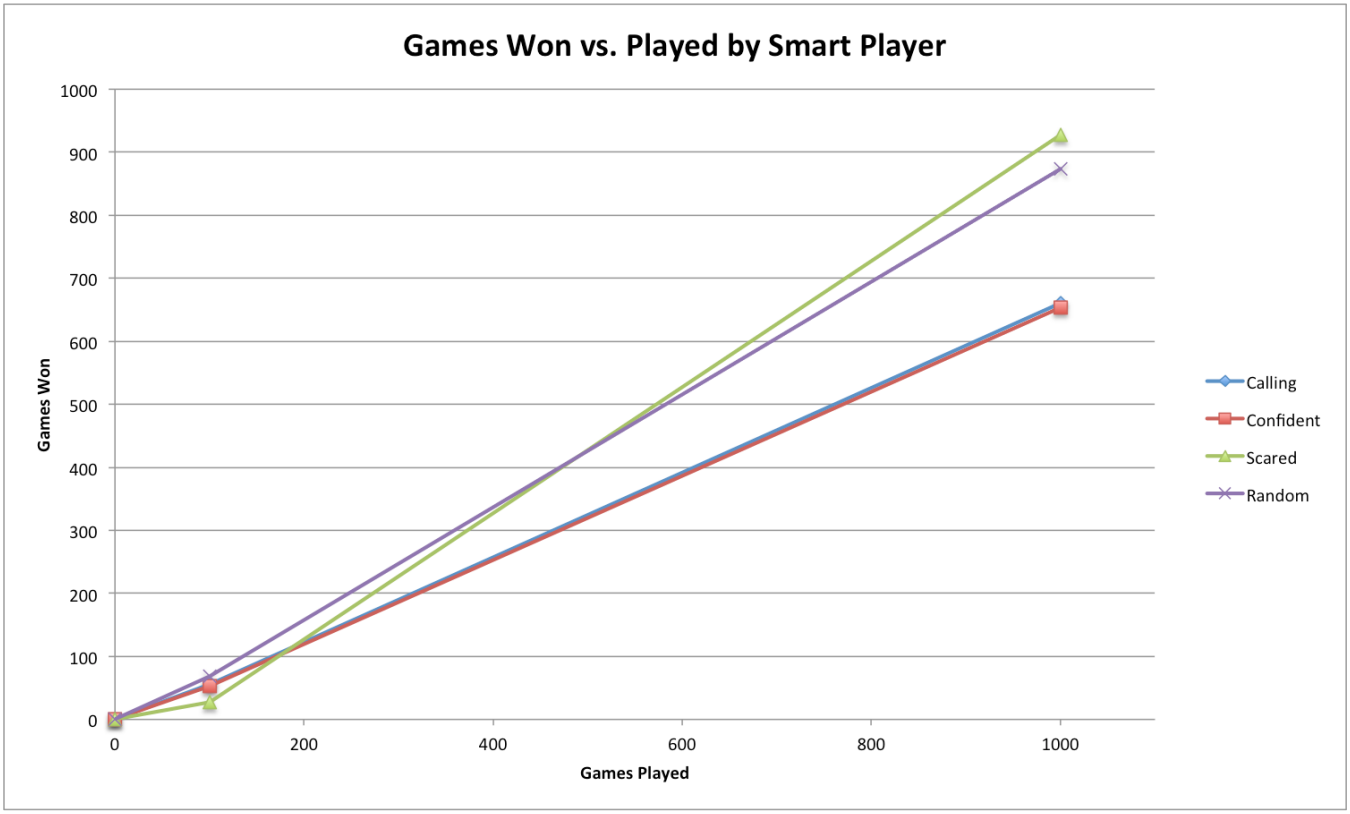

Fig. 3. The number of games won first without training and then with, for four computer opponents

Conclusion

We believe, with additional time, we could further hone our smart CPU to a point where it could read a substantial set of strategies and appropriately adapt to them. Even though our CPU touches on just a small part of the massive set of considerations for a smart computer player in Poker, we feel we have captured the core of a capable CPU given the amount of time we had this semester and a group of only two people; any additional developers would have been a helpful resource. Nevertheless, we accomplished our implementation goals and look forward to expanding the system even further.