CS182 Final Project: The Effectiveness of Reinforcement Learning Techniques in Texas Hold'em Poker

Noah Houghton, Matthew Mandel, Kyle Sanok, and Philip Thomsen

December 18, 2018

1 Abstract

This paper details our implementation of a variety of artificial intelligence approaches to playing a variant of poker called Texas Hold'em. We constructed agents that make use of different algorithms in reinforcement learning ("RL") and then tested them to determine the best RL approach to the game.

2 Introduction

At its core, artificial intelligence is the development of algorithmic ways to represent and find solutions to problems which involve some amount of decision-making. One of the interesting ways in which this technology can be applied and advanced is the playing of games. Specifically, we chose to apply a variety of artificial intelligence strategies to the game of poker to see if agents could learn to win the game. Poker is an interesting game for the field of A.I. because it is a game which hinges on imperfect information and managing risk.

We are not interested in creating the greatest poker bot in the world: other, more experienced researchers, such as those working on Deepstack, have already set their minds to the task.[5] Instead, we are interested in using the game of poker as a canvas on which to paint the relative strengths and weaknesses of different algorithms in RL.

3 Background and Related Work

3.1 The Game

Texas hold 'em is a variation of the card game of poker. Two cards, known as hole cards, are dealt face down to each player, and then five community cards are dealt face up in three stages. The stages consist of a series of three cards ("the flop"), later an additional single card ("the turn" or "fourth street"), and a final card ("the river" or "fifth street").

Each player seeks the best five card poker hand from any combination of the seven cards of the five community cards and their two hole cards. Players have betting options to check, call, raise, or fold. Rounds of betting take place before the flop is dealt and after each subsequent deal. The

player who has the best hand and has not folded by the end of all betting rounds wins all of the money bet for the hand, known as the pot.[1]

3.2 Poker and AI

Because of the enormous complexity of poker as a game given the lack of information and vast number of possibilities, most research in the area has focused on more complicated AI techniques such as deep learning. In 2017, DeepStack became the first A.I. to outperform professional poker players in heads-up no-limit Texas hold'em poker.[5] Similar success using the technique of solving subgames has been replicated through bots like Claudico [2] and Libratus [3].

The algorithms behind these bots are well beyond the scope of this course. They do illustrate, however, both the usefulness of games – and specifically poker – in artificial intelligence research, as well as the tractability of creating an A.I. poker agent which can perform at very high levels of play.

Using their success as inspiration, we set out to achieve a much more manageable task: to understand how different RL techniques from our course can be used to play against an opponent that replicates the ability of an amateur poker player.

4 Problem Specification

Our goal with this project is to understand how different RL algorithms can be used to play poker. In order to compare different approaches, we will determine an algorithm's performance (at least in part) by its ability to outperform an agent that approximates how actual amateur poker players play. [4] The way we decided to model such an opponent was by hardcoding the agent's response (fold, call, raise) to hand strengths. In creating an agent that can outperform this opponent, we are by definition creating agents that are able to learn mappings of hand-strengths to actions which are better than the ones we hardcoded. Beyond the agent's win rate as a metric that will influence our understanding of how good an algorithm is at playing poker, we are also interested in its time and space complexity: good poker agents should win and do so at reasonable computing costs.

5 Approach

We are interested in comparing how the application of different algorithms results in an impact on the performance of a poker agent. Broadly, we were interested in two kinds of RL agents: simulation based [5] and Q-learning variants. Though we expected the simulation based algorithms to perform better than the Q-learning based algorithms, we were interested in getting more specific information on how both kinds of RL agents actually perform and thereby more thoroughly exploring the different RL approaches to poker.

[1] This description of the game was taken from Wikipedia.org on December 18, 2018.

[2] https://www.riverscasino.com/pittsburgh/BrainsVsAI

[3] https://www.cs.cmu.edu/ noamb/papers/17-IJCAI-Libratus.pdf

[4] see section 6.2.2 for further discussion of the agent we are playing against

[5] By "simulation based agents," we mean those agents that learn what to do by simulating games before making a decision.

5.1 State Spaces

We also experimented with the state space for our bots. We initially thought we would store a tuple of (Hand Strength, Size of Pot, Stack) as the state as that is all of the information a real poker player has access to when making decision. Because of the massive number of possibilities of combinations of hand strength, pot size, and stack we were somewhat hesitant about how we modeled the problem. A further concern was the redundancy of states whose only difference was 1 dollar in the pot or stack, which would have to each be learned independently even though the best action in each of those states is probably the same. After some testing and discussion, we found that the pot size and stack did not have a discernible effect on how the player should act, and so we decided to remove them from the state space. Hand strength seemed to us the single most important metric about their situation that a poker player has access to for evaluating how they should act, and therefore we focused on our state space on just hand strength. For all our tests in this paper, we define the state space simply as the strength of the agent's hand.

5.2 Agents and Algorithms Used

Agent	Description
Random	Selects a random valid action.
Naive	Bets proportionally to the strength of its hand. This approximates the way that most human players would approach the game.
Monte-Carlo[4] [6, p. 530]	This agent uses Monte Carlo simulations to learn how it should act (call, fold, raise) given its hand strength. These simulations consist of finding the future potential outcomes (its future hand strength, its opponent's hand, the hand winner) the bot could get given its current cards and the community cards and determines its chance to win given those simulated games. Then, it decides whether or not to bet based on its calculated chance to win the current hand.
Look Ahead	This agent attempts to predict the potential strength of future hands and the chance that the next card drawn will create them. Then, it makes a decision about how to bet based on the chance that it will have a good hand. This bot uses the MonteCarlo algorithm but only focuses on hand strength, rather than also accounting for the strength of the opponent's hand. This results in a faster runtime.
Q-Learning [6, p. 844]	This agent records the results of playing games and uses that information to make more informed decisions as it learns the game. With no training it behaves similarly to a random bot; however, as it trains and learns the state space it makes increasingly more efficient decisions.
Small Q-Learning	For this agent, we modified the state space of our Q-Learning agent to allow for a more compact and easily learn-able state space. It is otherwise the same as Q-Learning, but the state space is reduced to the natural log of the hand strength.
Approximate Q-Learning [6, p. 845]	A modified version of Q-Learning wherein the agent learns how much to weight the values of different features when calculating the Q-value of a state, action pair.
SARSA [6, p. 844]	A modified version of the Q-Learning algorithm which waits until the agent has made two actions before updating the value of its previous state-action pair. This algorithm tends to result in a more conservative Q-value space.[6]

Table 1: Agents

6 Experiments

6.1 Framework

We initially tried to convert the Pacman code from the semester's problem sets into a working poker simulator. This involved a deep dive into how the Pacman game code integrated with the parts of the AI agents we implemented during the semester, and many hours poring over the code, trying to dissect its lower-level functions. As we discovered, the framework of the Pacman game was simply not flexible enough to accept a new game type without more extensive rewriting.

We ultimately decided it would be more beneficial to the project to switch to an existing poker framework. We settled on PyPokerEngine for its power and quick setup, although a lack of documentation meant that, once more, we had to dive into the source code to see what made the engine tick in order to create our agents and test them.

6.2 Methodology

6.2.1 Scope of Testing

We ran at least 1000 games to calculate the victory[7] percentage for each bot against the NaiveBot. Agents that needed training were tested on progressively larger amounts of games to see how training might impact performance.[8] Learning parameters were held static or changed one at a time isolated from other game changes to maintain causality in the data. All win rates discussed in this paper are the result of 5 tests of a thousand games each run independently and averaged.

6.2.2 Dataset

We trained our agents against NaiveBot, a choice we made for a variety of reasons: most importantly, we believe that it is our best approximation for how non-professional humans play. Ultimately, we wanted to see if it was possible to create learning agents without the use of deep learning which could defeat human players. We decided that a good heuristic for how close to that goal we could come would be to see how the agent performed against an opponent that acted how its potential human opponents might. Given that most people play poker by trying to evaluate their hand, and then deciding how to act given how good their hand is, we thought NaiveBot would be a good representation of a human player.

We also chose NaiveBot because it is relatively efficient in terms of its demands on computing power when testing. This consideration also led us to our decision to only test our bots against individual opponents rather than groups. Although a large table would have been more realistic, it would have also required much more computing power and made our large scope of testing less feasible.

We also found that the game configuration could have some impact on agent performance. After doing several rounds of testing, we noticed that the learning agents were doing much better when given more rounds and more starting money. We think that fewer rounds and higher blinds initially biased our data against learning approaches that would either run out of money quickly because of the high blinds and therefore lose out on training or not see enough hands because of the paucity of rounds and so again not be able to train enough to improve their actions. When we changed the game configuration, we saw improvement in the learning agents' performance, which confirmed our suspicion that the structure of the tests was biased against the learning approach.

We gathered the data presented below under the following game configuration:

Ante Amount	Beginning Stack	Max. Number of Rounds	Small Blind Amount
5	500	50	4

[7] see 6.2.4 for discussion of what counts as a victory

[8] refer to fig.5

These parameters were selected after initial tests revealed the effect of the starting configuration on agent performance:

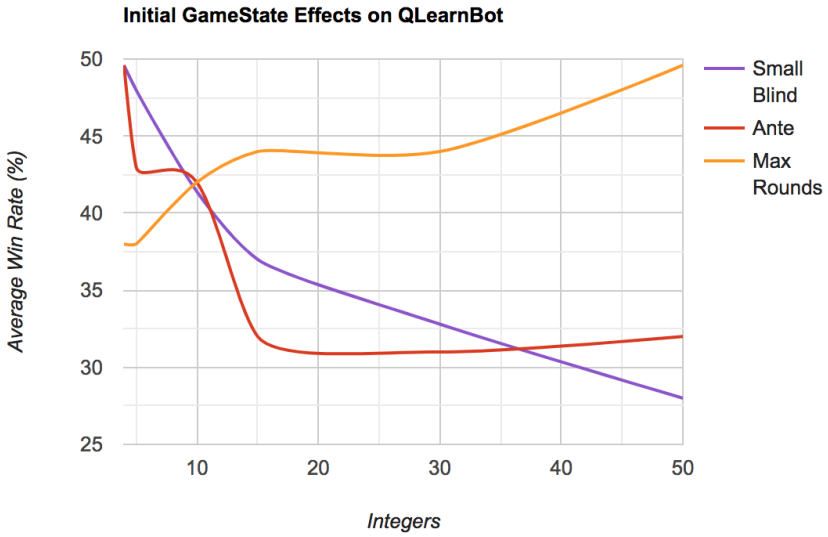

Figure 1: This graph shows how QLearnBot performs when the configuration is changed based on ante amount, small blind amount, and the amount of rounds in a game. Performance falls dramatically as ante and small blind increase and performance increases as the amount of rounds played increases.

This graph, calculated by averaging 5000 games for each set of parameters, demonstrates the initial game state's effect on our learning agents. It shows that a high max number of rounds, low ante, and low small blind lead to better learning. Thus, we used the above configuration as the standard configuration for our comparative algorithmic testing.

6.2.3 Tuning Q-Learning

Unlike the deterministic models, we had to tune the learning parameters of our Q-Learning algorithm. These parameters affect how likely the agent is to explore new state-action pairs, how much it will weight new rewards over old, and how much it cares about when a reward is earned in its lifetime.

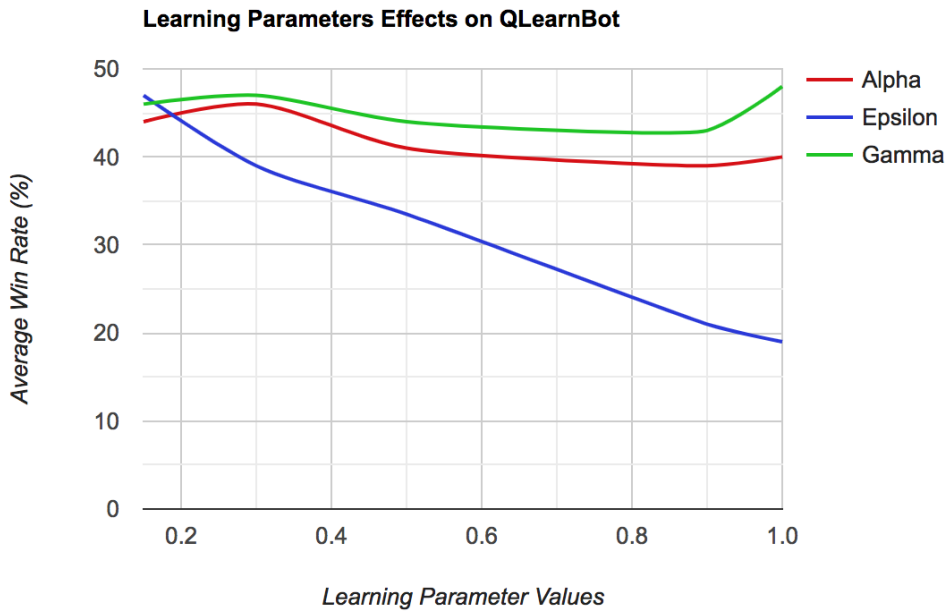

Figure 2: This graph shows how QLearnBot's performance was affected by changes in parameter values. We found that the optimal parameterization to be epsilon = .15, alpha = .3, gamma = .9.

Our agent worked best with low values of alpha and epsilon, and changes in gamma seemingly had no real long term effect. However, we expect that with more time and training cases we would be able to find gamma to have more of an effect. Using this data, we selected $\alpha = .3$, $\gamma = .9$, and $\epsilon = .15$ for our Q-Learning and Q-Learning based agents. We used these parameters when gathering the data discussed in this paper.

6.2.4 Evaluation

We spent a fair amount of time deciding what should constitute agent success. There was an obvious factor — number of games where the agent has the most money at the end — but we were also interested in seeing how many rounds the agent would win. Initially, we thought that round wins and game wins should rise together. We quickly realized that this incorporated an incorrect belief of ours as bad poker players, which was that the best players will win the most rounds. In fact, the best poker players know when to fold, and then win big — which means that even though they might not win the most rounds, they have the most money at the end of the game. As our results show, our agents learn this strategy over time, as the number of round victories go down as game victory rates climb.[9] We made sure to run tests of at least 1000 games at a time to minimize the effects of the volatility of individual game results and get more accurate results as a whole.

[9]Refer to tables in Appendix C, particularly Table 3

6.3 Results

Our overall results can be seen in Figure 3 below, which compares the percentage of games won against the NaiveBot. Our MonteCarlo and LookAhead Agents performed very well against the NaiveBot, much better than our Q-learning agents. Within our learning agents, our Q-Learning bot performed the worst, possibly due to the large state space, and our Small Q-Learning bot, which operates under a compressed state space, performed best. More detailed comparative analysis of algorithm performance can be found in the next section (6.4).

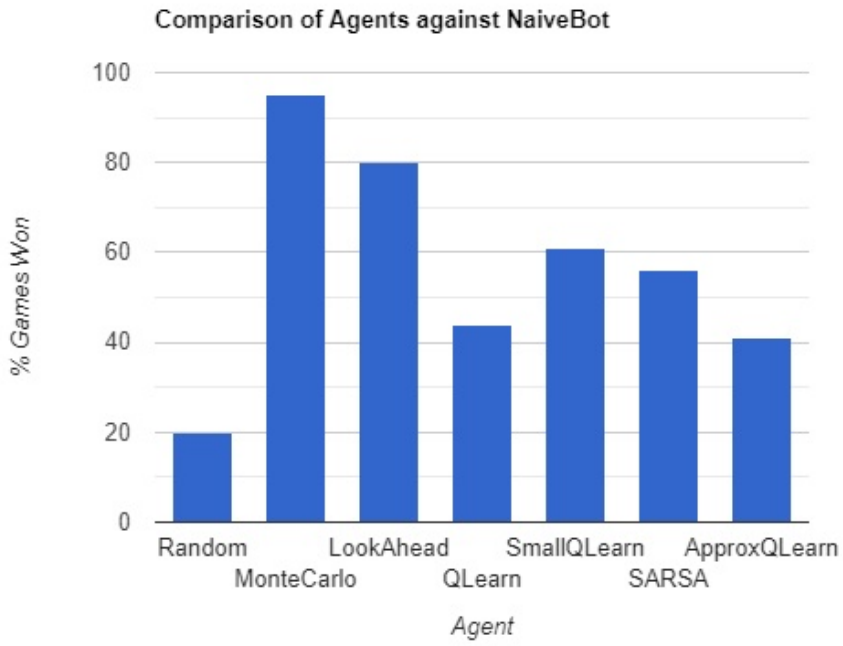

Figure 3: This graph shows how each agent performs under our standard configuration against NaiveBot. Though it looks like MonteCarloBot is far and away the best, this graph does not take into account other factors like time complexity. Furthermore, the graph shows that all of our RL approaches outperformed an agent that randomly decided how to act.

6.4 Algorithm Performance Analysis

6.4.1 Monte-Carlo

The MonteCarloBot performed the best of the agents we tested against the NaiveBot by a significant margin. This result is not surprising because the MonteCarloBot is able to amass sufficient information to make an informed, but still probabilistic, decision about the best path forward by simulating 1000 rounds every time it decides what to do, it. Though this strategy allows the MonteCarloBot to win the vast majority of games it plays, it does so only at tremendous time cost. Indeed, it is on average about 100 times slower than Q-Learning.

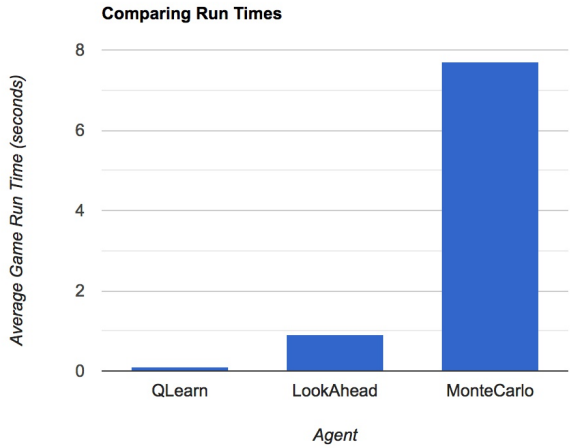

Figure 4: This graph represents the time to play a single poker game for MonteCarloBot, LookAhead, and QLearning. Though we could have also compared all of the different agents, this graph is just meant to show the dramatic time inefficiencies of the MonteCarlo approach when compared to the QLearning variants. QLearnBot moves 99% faster than MonteCarloBot.

6.4.2 Look-Ahead

To reduce the time complexity of MonteCarloBot which tries to learn the opponents' hand in addition to learning what to do given the strength of its own hand via simulaitons, LookAhead only focuses on the player's own hand. This reduced amount of information and a reduction in the amount of simulations runs reduced the time complexity by about a factor of 8 (see fig. 4). While making a less well informed decision, we see that LookAhead still performs around 1.5-2x better than the Q-learning algorithm variants, and about 4x better than the random player. Though LookAhead runs much faster than MonteCarloBot, it is still about 10x slower than the Q-Learning variants.

6.5 Q-Learning

Leaving behind simulation based algorithms, we turned our attention to other RL algorithms that had far less demanding time costs. (see fig. 4) The first of which was a Q-Learning agent. This algorithm used Q-Learning to learn better moves to perform in a given state, represented by the strength of the player's hand at a given point in time. This Q-LearningBot had fairly significant space costs, as the worst case state space was the product of the number of hand-strengths in poker and the number of actions one could take in each of those states. We hypothesize that it is the size of this state space that in part accounts for Q-LearningBot's poor performance at lower training levels (it has not visited enough states) and then slower improvement as it trains.

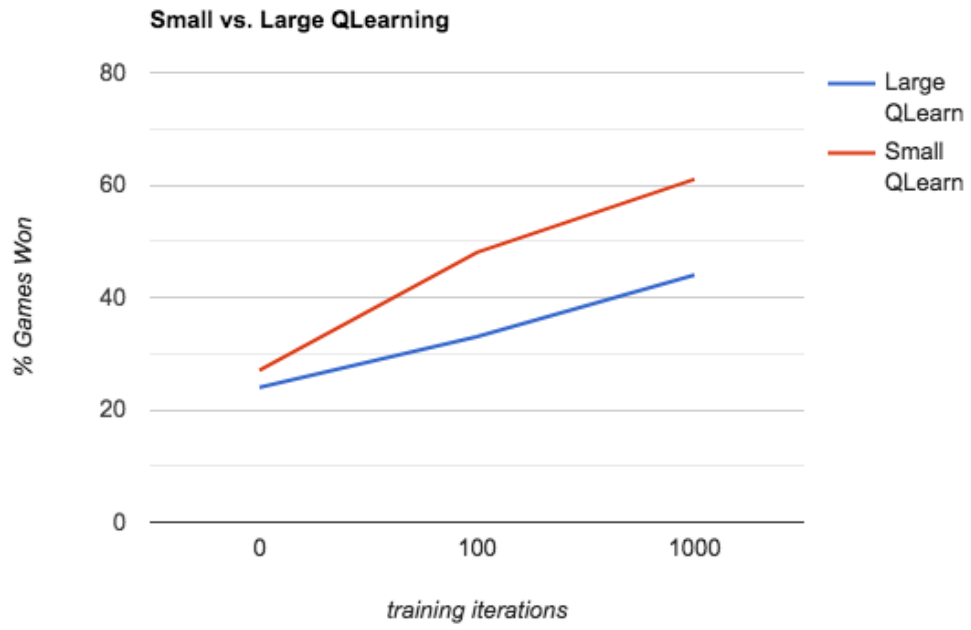

Figure 5: This graph compares how SmallQBot and QLearnBot perform as training increases (from t = 10 to t = 1000). In the graph, we see that though SmallQBot consistently performs better than QLearnBot, it improves more slowly after 100 training games than QLearnBot. Analysis of SmallQBot can be found in 6.5.1

Though due to computational power restrictions we were unable to extensively test Q-LearningBot's performance beyond 1000 training games, we would expect to see some more improvement as training time is increased and the agent is able to visit and revisit more states in the vast state space, thereby refining its understanding of the total state space.

6.5.1 Small Q-Learning

As a modification to QLearningBot, we made an agent we call SmallQBot. Because it appeared to us that one of QLearningBot's key weaknesses was its inability to learn a large state space in a relatively short amount of time, we made a modification that shrunk the state space by defining a state as the truncated natural log of the current hand strength. This operation had the effect of allowing the agent to visit every state (147) after only a few thousand actions, whereas QLearnBot was still finding new states after a comparable number of actions.

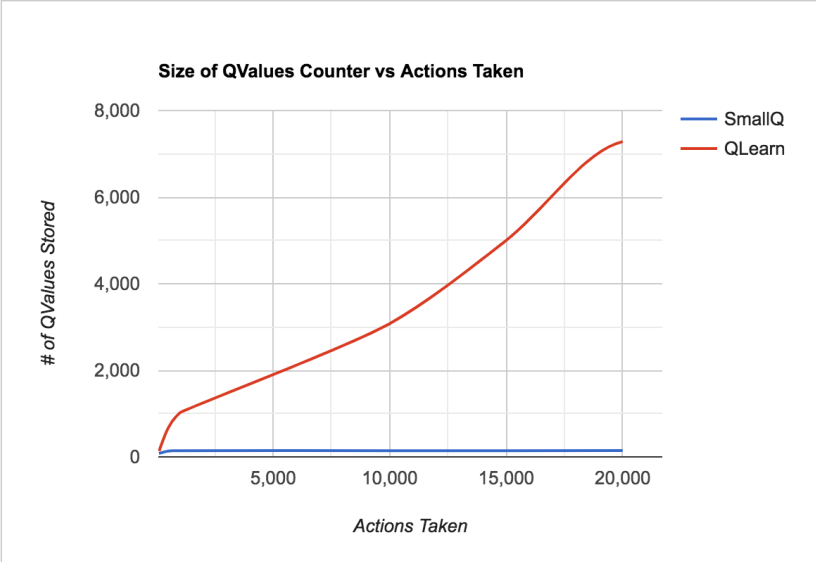

Figure 6: This graph shows the size of the Counter storing Q-values as the agent acts more. Though the size of SmallQBot's Counter quickly converges to 147, the QLearnBot's Counter size continues to increase even after 20,000 actions

The SmallQBot agent does do better than QLearnBot with less training (see earlier graph), which we expected and indeed designed for; however, over time this improvement becomes less noticeable. We hypothesize that SmallQBot stops improving as much because the policy that it learns for the smaller state space is a less accurate one because each state encodes many different hand-strengths, which most likely have different optimal actions. In this way, our improvement in state space from QLearnBot to SmallQBot demanded a sacrifice of accuracy.

6.5.2 SARSA

As a modification to QLearningBot, we also made SARSABot. This algorithm is very similar to Q-Learning, with the exception that rather than base its evaluation of a state-action pair on the best possible action that the agent will take in the next state, it updates based on the action that the agent does take from that next state. This agent improves on the performance of the Q-Learning algorithm. We can see this by comparing the data in fig. 7 to the data in fig. 5. Whereas Q-Learning reaches a win rate of just over 40% after 1000 training games, SARSA makes it to 56%. We believe this difference emerges because SARSA does not take chances as often as Q-Learning, which means that in the game of poker it is more likely to have learned the negative value of gambling on a risky hand, whereas Q-Learning may take longer to learn the same.

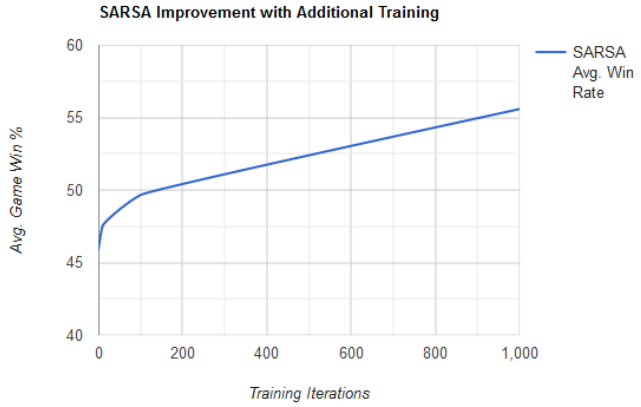

Figure 7: This graph shows how the SARSA agent's performance improves as it learns how to play across more training games.

6.5.3 Approximate Q-Learning

The last Q-Learning variant we looked at was an agent that used Approximate Q-Learning to learn how to act given a hand-strength.

Because the only feature that the Approximate Q-Learning agent learned a weight for was hand-strength, its performance depended on whether or not it could learn to positively weight hand-strength. As you can see in the graph below, the variance of the agent's performance was very high at low training values. We ascribe this variance to the agent not converging on a weight for hand-strength – it could get lucky and happen to learn to positively weight hand-strength, in which case it might perform fairly well. The benefit of more training was that the agent much more consistently learned to weight hand-strength positively, which meant that its mean win rate increased significantly.

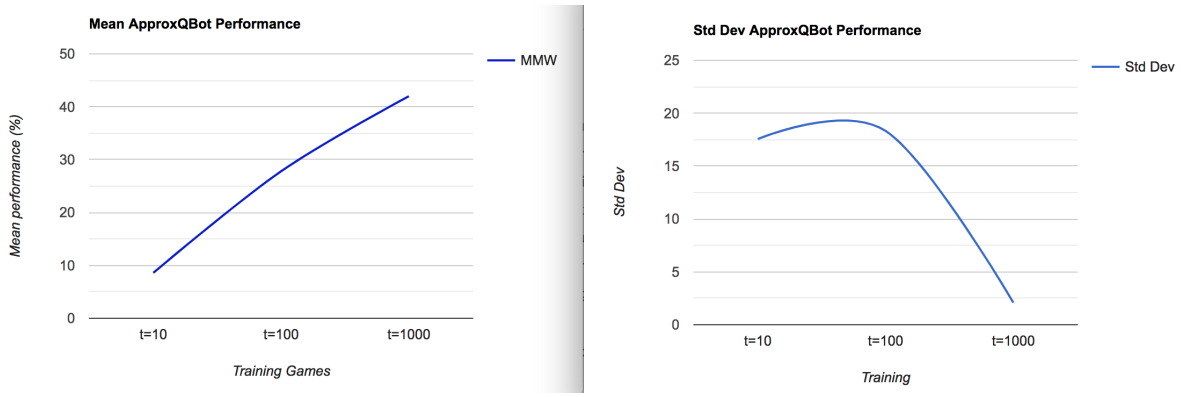

Figure 8: The right graph tracks the standard deviation of the ApproxQBot's performance across 5 different simulations of 1000 games after t number of training games, and the left graph tracks its mean win rate. As it trained more, ApproxQBot's variance fell and mean increased.

This variation addresses the Q-Learning agent's state space problem: by only learning how to weight features of states, the Approximate Q-Learning agent can generalize the information it learns across states. Though the agent does not have a state size problem, it performed slightly worse than the Q-Learning agent. We hypothesize that this inability to improve over the Q-Learning agent came from the Approximate Q-Learning agent's loss of accuracy that came with its increased generality. We also did not exploit the fact that ApproxQBot could extract more features and thereby make a more informed decision without space tradeoffs. Adding more features may have helped make up for the accuracy loss from the generality. This would be an interesting next step to pursue.[10]

7 Conclusion

7.1 Takeaways

In brief, our project demonstrates that simulation-based approaches will on average play the game of Texas Hold'em better than any Q-learning variant. However, our endeavor also proves that those methods, which are more optimal, are also much slower than the Q-learning algorithms. We found that although the Q-learning techniques do not appear to converge to an optimal policy for poker (which we expected, because poker is not a solved problem), dynamic artificial intelligence algorithms can still outperform an agent that approximates an amateur poker player

We demonstrate in a practical way the tradeoffs between space and time complexity and accuracy with these algorithms, and show that poker as a game presents a compelling space for artificial intelligence research and development.

7.2 Next Steps

One next step we would have liked to developed further is developing our FancyApproxBot[11], which uses an Approximate Q-learning algorithm but with more features than just hand-strength. We think our Approximate Q-Learning agent's performance could be improved greatly by taking into account, for example, "opponent-confidence", which tracks how many times your opponent has raised. Learning to respond to opponent's actions in addition to your own hand strength should help the agent win.

One limitation of the framework we chose was that it does not include functions which make it easy to smartly choose an amount to raise. Because we wanted to focus more on the algorithms themselves than gameplay, in the event that an agent chooses to raise it randomly chooses an amount between the minimum and maximum possible amounts. All agents use this same logic to decide how much to raise when applicable. We believe that while this may lower the overall effectiveness of agents, every agent is subject to this same deflating factor. Therefore we do not believe that this decision unfairly biases the data towards any individual bot. In later implementations of these bots, it would be useful to add an algorithm to decide how much the bot should bet in the case of a raise, or perhaps even encode it somehow into the state space.

[10] cf Section 7.2

[11] FancyApproxBot's code is in the code base. We do not discuss any tests on it in this paper.

A System Description

https://github.com/Noah-Houghton/cs182-poker-player

Dependencies

```
pip install PyPokerEngine
```

```
pip install pypokergui
```

To Run

Single game with GUI

First, run this command to automatically generate a config file (you can edit this later if you wish or re-run the command).

```
python setupgame.py -a <ante> -b <blind_structure> -s <initial_stack> -r <max_round> -m <small_blind> -p <agentType>.py -n <numAgents>
```

Sample command to setup default game, no args `python setupgame.py`

Sample command to set game with some args `python setupgame.py -a 5 -s 250 -r 5 -m 10 -p MonteCarloBot.py`

Now, run this command to start the server and play!

```
pypokergui serve poker_conf.yaml --port 8000 --speed moderate
```

Command Line, Many Games (no GUI)

This command simulates many games running.

Example command line to run with default config `python simulate.py -p MonteCarloBot -o FishBot -n 3 -g 10`

Example command line to run with custom config `python simulate.py -p MonteCarloBot -o RandomBot -n 3 -g 10 -a 5 -s 500 -r 15 -m 15`

```
simulate.py -p <agentType> -o <opponentType> -n <numOpponents> -g <numGames> -a <ante> -b <blind_structure> -s <initial_stack> -r <max_round> -m <small_blind> -t <numTraining> -w <writeToLog> -l <alpha> -e <epsilon> -d <discount>
```

Note that the latter three values `-l`, `-e`, and `-d` are only used in QLearning agents. If they are accidentally left in, the simulator will ignore them. The default values for these variables are not defined by `simulate.py`, but rather by the agents themselves in their `__init__` function.

To run automated tests and record the data generated, simply populate **tests.txt** with the flags you wish for **simulate.py** to run. Then run **python automatetests.py** in the command line.

B Group Makeup

- Noah Houghton
    1. Programmed initial Pacman-based engine
    2. Set up PyPokerEngine and helper scripts
    3. Bot developer
    4. Data gathering and analysis
- Matt Mandel
    1. Integrated PyPokerEngine with RL Agent Logic
    2. Bot developer
    3. Data gathering and analysis
- Philip Thomsen
    1. Data gathering and analysis
    2. Bot developer
- Kyle Sanok
    1. Data gathering and analysis
    2. Bot developer

C Sample Data Tables

Agent	Average Run Time (seconds)	Average Stack	Most-Money Victory (MMV) Rate	Round Victory (RV) Rate
RandomBot	.024	192	193/1000 (19.3%)	12.9%
NaiveBot	.024	192	193/1000 (19.3%)	12.9%
LookAheadBot	0.9156	796	797/1000 (79.9%)	43.14%

Table 2: Simulation Results for Non-Learning Agents
***State Space**: Hand Strength*
***Opponents**: 1 NaiveBot*

Value	Alpha Success (MMV%)	Epsilon Success (MMV%)	Gamma Success (MMV%)
0.15	44	47	45
0.3	46	39	47
0.5	41	33.5	44
0.9	39	21	43
1.0	40	19	48

Table 3: QLearnBot Data for Changing Learning Parameters (tested at 2000 training games. Standard setup is $\alpha = .3$, $\gamma = .9$, and $\epsilon = .15$)
***State Space**: Hand Strength*
***Opponents**: 1 NaiveBot*

Agent	Average Run Time (seconds)	Average Stack	Most-Money Victory (MMV) Rate	Round Victory (RV) Rate	MMV Rate Vs. Random	RV Rate Vs. Random	Num. Training Games	α, γ, ϵ
RandomBot	.037	92	62/1000 (6%)	138/7220 (2%)	-	-	-	-
QLearnBot	0.040	265	266/1000 (26.60%)	920/4854 (18.95%)	430%	950%	0	$\alpha = 0.3$ $\gamma = 0.9$ $\epsilon = 0.15$
QLearnBot	0.039	243	244/1000 (24.40%)	868/4766 (18.21%)	400%	900%	0	$\alpha = 0.3$ $\gamma = 0.9$ $\epsilon = 0.15$
QLearnBot	0.040	256	257/1000 (25.70%)	918/4835 (18.99%)	420%	950%	10	$\alpha = 0.3$ $\gamma = 0.9$ $\epsilon = 0.15$
QLearnBot	0.040	268	269/1000 (26.90%)	926/4828 (19.18%)	440%	960%	10	$\alpha = 0.3$ $\gamma = 0.9$ $\epsilon = 0.15$
QLearnBot	0.042	341	342/1000 (34.20%)	1252/5018 (24.95%)	560%	1300%	100	$\alpha = 0.3$ $\gamma = 0.9$ $\epsilon = 0.15$
QLearnBot	0.044	317	318/1000 (31.80%)	1242/5263 (23.60%)	525%	1250%	100	$\alpha = 0.3$ $\gamma = 0.9$ $\epsilon = 0.15$
QLearnBot	0.066	450	451/1000 (45.10%)	2190/8289 (26.42%)	760%	1300%	1000	$\alpha = 0.3$ $\gamma = 0.9$ $\epsilon = 0.15$
QLearnBot	0.065	446	447/1000 (44.70%)	2193/7935 (27.64%)	750%	1350%	1000	$\alpha = 0.3$ $\gamma = 0.9$ $\epsilon = 0.15$
QLearnBot	0.070	479	480/1000 (48.00%)	2502/8421 (29.71%)	800%	1490%	2500	$\alpha = 0.3$ $\gamma = 0.9$ $\epsilon = 0.15$
QLearnBot	0.068	473	474/1000 (47.40%)	2436/8758 (27.81%)	790%	1380 %	2500	$\alpha = 0.3$ $\gamma = 0.9$ $\epsilon = 0.15$

Table 4: Q-Learning Data

Agent	Average Run Time (seconds)	Average Stack	Most-Money Victory (MMV) Rate	Round Victory (RV) Rate	MMV Rate Vs. Random	RV Rate Vs. Random	Num. Training Games	α, γ, ϵ
RandomBot	.037	92	62/1000 (6%)	138/7220 (2%)	-	-	-	-
SARSABot	0.031	819	41/100 (41.00%)	95/269 (35.32%)	683.33%	2050.00%	0	$\alpha = 0.1$ $\gamma = 0.9$ $\epsilon = 0.15$
SARSABot	0.028	959	48/100 (48.00%)	115/288 (39.93%)	800%	2400.00%	0	$\alpha = 0.1$ $\gamma = 0.9$ $\epsilon = 0.15$
SARSABot	0.024	1039	52/100 (52.00%)	120/269 (44.61%)	866.67%	2200.30%	10	$\alpha = 0.1$ $\gamma = 0.9$ $\epsilon = 0.15$
SARSABot	0.021	979	49/100 (49.00%)	117/283 (41.34%)	816.67%	2067.00%	10	$\alpha = 0.1$ $\gamma = 0.9$ $\epsilon = 0.15$
SARSABot	0.020	959	48/100 (48.00%)	124/317 (39.12%)	800.00%	1956.00%	100	$\alpha = 0.1$ $\gamma = 0.9$ $\epsilon = 0.15$
SARSABot	0.018	959	48/100 (48.00%)	123/329 (37.39%)	800.00%	1869.50%	100	$\alpha = 0.1$ $\gamma = 0.9$ $\epsilon = 0.15$
SARSABot	0.049	1160	58/100 (58.00%)	221/1210 (18.26%)	966.67%	913.00%	1000	$\alpha = 0.1$ $\gamma = 0.9$ $\epsilon = 0.15$
SARSABot	0.033	1019	51/100 (51.00%)	182/1017 (17.90%)	850.00%	850.00%	1000	$\alpha = 0.1$ $\gamma = 0.9$ $\epsilon = 0.15$
SARSABot	0.045	1179	59/100 (59.00%)	264/1310 (20.15%)	983.33%	1007.50%	2500	$\alpha = 0.1$ $\gamma = 0.9$ $\epsilon = 0.15$
SARSABot	0.041	1059	53/100 (53.00%)	193/1107 (17.43%)	883.33%	871.50%	2500	$\alpha = 0.1$ $\gamma = 0.9$ $\epsilon = 0.15$

Table 5: SARSA Data
Note that even though these games are run with a starting stack of 1000 and are run only 100 times, we have verified that the data is still representative of the algorithm's performance.

References

[1] How to play texas hold'em poker - the official rules | PokerNews.

[2] Adesh Gautam. Introduction to reinforcement learning (coding SARSA) — part 4.

[3] ishikota. Poker engine for poker AI development in python. https://github.com/ishikota/pypokerengine.

[4] Kevin Jacobs. Introduction - PyPokerEngine.

[5] Matej Moravčík, Martin Schmid, Neil Burch, Viliam Lisý, Dustin Morrill, Nolan Bard, Trevor Davis, Kevin Waugh, Michael Johanson, and Michael Bowling. Deepstack: Expert-level artificial intelligence in heads-up no-limit poker. 356(6337):508–513.

[6] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Series in Artificial Intelligence. Prentice Hall, Upper Saddle River, NJ, third edition, 2010.