COMPSCI 683 Project: Quality Diversity for Texas Hold’Em

John Raisbeck
jraisbeck@umass.edu

Peter Phan
pkphan@umass.edu

Ruiqi Hu
ruiqihu@umass.edu

Giang Nguyen
gnnguyen@umass.edu

Abstract

Limit Texas Hold’em is a popular variant of Poker in which there are both community cards, which are common knowledge, and private cards, which are not known to other players, each with numerical values. We train agents to play one-on-one matches of Limit Texas Hold’em in a tournament setting, where each player is randomly matched with an opponent, drawn from a set containing itself, all other learning models, and a number of hand-coded Texas Hold’em agents. The learning agents are trained using a Reinforcement Learning Algorithm, PPO, which is an actor-critic method. The reward function is created by combining the gain of the player during a round and a diversity bonus at each state, which rewards learning players for being different from one another, inspired by work on Novelty Search[1]. We provide agents with several abstractions on the state-space to simplify the game and improve training speed. By running all of the models simultaneously in a large training tournament with diversity bonuses, we are able to maintain a high-quality and diverse training regime for each player.

1 Introduction

In this project, we aimed to improve the strategic decision-making of poker-playing AI by incorporating diversity strategies into the Proximal Policy Optimization (PPO) agent. By embracing quality diversity, we developed a range of robust strategies. We simplified the complex decision-making in poker by using game state and action abstractions. Our tests against both hand-crafted and algorithmically trained players showed that our PPO agents consistently outperformed many opponents, highlighting the benefits of diversity strategies in creating adaptable and competitive AI for strategic games like poker.

1.1 Our Contributions

We developed an AI poker bot capable of competing effectively against various hand-coded poker agents. Our bot harnesses self play and play against other learning and handwritten policies to learn competitive strategies (agent learning by Peter Phan and tournament code/multithreading by John Raisbeck), and employs sophisticated card and action abstraction techniques to optimize decision-making and computational efficiency (implemented by Peter Phan). To ensure robust performance across various computing platforms and enhance the bot’s adaptability from Mac, Linux, and Windows systems, further system-specific optimizations were implemented (implemented by John Raisbeck). Additionally, we validated our bot’s capabilities against a series of trivial models that served as baseline opponents to fine-tune its strategies under controlled conditions (testing by Giang Nguyen).

1.2 Related Work

Limit Texas Hold'em is a variant of Poker with over 10^{14} information sets [2]. The version we consider is played between two players and consists of four betting rounds: pre-flop, flop, turn, and river. At the start of each game, each player is chosen randomly to be either the small or big blind and is required to bet a corresponding fixed bet. During a player's turn, they can choose to act by folding, calling, or raising. A game ends either when a player folds, forfeiting the pot (sum of wagers), or the game goes to a showdown where the player with the better hand wins the entire pot. In this variant, raise amounts are fixed and episodes are played for a maximum of 100 games or until one player loses their entire initial stack.

Our project builds on established methodologies for Poker and augments them with a diversity bonus during policy learning. Inspired by prior work in AI poker strategies, we have developed and fine-tuned an approach based on the Reinforcement Learning technique of Proximal Policy Optimization (PPO)[3]. The diversity component[1] implemented in our agents help them to generate, as a set, a wide array of competitive approaches to poker, providing a rich training bed for all of the agents being trained.

2 Model and Preliminaries

2.1 Abstractions

We represented the game state using a vector representing the street, agent's hole, community cards, a compressed version of both agents' actions during the episode, and the agent's stack. The street and stack are encoded as normalized scalars. The agent's hole and community cards are encoded as one-hot vectors. We abstracted the action history of both players into ratios of calls to raises. We used a 2-dimensional vector to represent the ratio of calls to raises counts played by the agent and another similar vector to represent the respective ratio for the opponent's actions. Importantly, this abstraction achieves its smaller size by ignoring temporal information, which could be valuable for certain strategies. Once concatenated, the game state is represented with a $5 + 52 + 52 + 4 + 1 = 114$ dimensional vector.

2.2 Process Model

Consider the following notation for the Poker process: we represent Texas Hold'em as a discrete time decision process, represented by a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \eta, r, \gamma)$ that we will hope to solve. The goal of our approach is to learn an optimal policy, π^*, that maximizes the expected cumulative reward.

We define the reward function r in the process as follows: there is no reward for changes in the agent's winnings from the game except in the final step, and at each step there is a small reward based on the distance between the agent's action-distribution on the current state and the action distribution which the other learning policies would have exhibited in the same state. Specifically, we define the return of a round to be the net change to the agent stack since the last round in the last state, denoting the stack after the game $\texttt{stack}'$ and that at the beginning of the game $\texttt{stack}$. For instance, if the agent ends up betting and losing \$60 after a round, their reward for the last state-action transition of that round would be -60. If the agent were to have won this round, the reward would be 60. The diversity bonus helps to encourage the learned policies to differentiate themselves and helps to maintain a diverse training environment for all of the policies. The diversity score is computed by taking the average of the three smallest Euclidean distances between action distributions (encoded as vectors in $\mathbb{R}^3$) of different training policies on the current state and added to the stack-based reward. The set of training policies is denoted Π in the equation below.

$$r(s,a) = \begin{cases} \texttt{stack}'(\pi) - \texttt{stack}(\pi) + \texttt{diversity\_bonus}(s,\pi,\Pi), & \text{if } (s,a) \text{ is terminal} \\ \texttt{diversity\_bonus}(s,\pi,\Pi), & \text{otherwise} \end{cases} \tag{1}$$

2.2.1 PPO

Proximal Policy Optimization (PPO) is a class of on-policy Reinforcement Learning algorithms introduced by Schulman et al. [3] that aims to take the biggest possible improvement step on a policy without stepping too far to cause performance collapse. In our implementation we used the PPO-Clip variant of PPO which clips the objective function to reduce the learner taking excessively large steps. We follow an actor-critic structure where we learn the optimal policy with an actor network and the value function with a separate critic network. We denote the parameters of a policy model θ and that of a value function network ϕ. Then we update our policy from π_{θ_k} using PPO-Clip as follows:

$$\theta_{k+1} = \arg\max_{\theta} \mathbb{E}_{s,a\sim\pi_{\theta_k}}[L(s,a,\theta_k,\theta)]. \tag{2}$$

Where L is the objective function which is defined as:

$$L(s,a,\theta_k,\theta) = \min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}A^{\pi_{\theta_k}}(s,a), \text{clip}\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, 1-\epsilon, 1+\epsilon\right)A^{\pi_{\theta_k}}(s,a)\right) \tag{3}$$

Where the advantage function $A^{\pi_{\theta_k}}$ is a function of the difference between the reward collected by π_{θ_k} and the value given by our critic model for a given state-action pair. We will use this difference as the loss function for optimizing our critic network. Altogether, our procedure for training a singular PPO agent goes as follows:

Algorithm 1 PPO-Clip

```
1: Initialize policy parameters $\theta_0$ and value function parameters $\phi_0$
2: for $k = 0, 1, 2, \dots$ do
3:     Play a round of poker with $\pi_{\theta_k}$ and collect a set of trajectories $\mathcal{D}_k = \{\tau_i\}$
4:     Compute rewards according to $(1)$
5:     Compute advantage based on $V_{\phi_k}$
6:     Update the policy to $\pi_{\theta_{k+1}}$ by maximizing $(3)$ via gradient ascent
7:     Update the value function to $V_{\phi_{k+1}}$ by regression via gradient descent
8: end for
```

2.3 Tournament Play

We trained each of 14 different PPO Players in thousands of rounds against the RaisedPlayer, MCPlayer, CallPlayer, RiskyPlayer, SafePlayer, RandomPlayer, HonestPlayer, and BluffPlayer, as well as the 14 learned PPO players (including self-play), to produce many diverse policies. When agents fell too far behind (for instance, occasionally, we found that agents would simply collapse into always folding), we culled them if their performance during validation rounds fell below a preassigned standard.

3 Results

In addition to the given randomized player and the player that always raises given in the starter code, we developed 8 other hand-coded players, some trivial, and others much more complex, including a player which ran Monte Carlo simulations to approximate the best action and a hand-model-quality estimator based player which raised only if the quality of the hand was above a certain threshold. See the Appendix for detailed descriptions of each player.

3.1 Experiments

We created and trained 14 policies in a single 3290-task tournament run (involving roughly 50,000 episodes of poker), and aimed to validate the performance of each policy. Testing was conducted by simulating 500 games, each with a maximum of 100 rounds. We look at the performance by PPO08 and PPO07 as they showed very contrasting policies but led the pack in terms of performance by consistently beating the random player and exhibiting impressive win rates against other players, notable the raise player. Figure 1 and 2 illustrate the percentage of actions taken by two different policies generated during our tournament run. Both policies exhibited a 100% win rate against the random player, with the bluff player posing the greatest challenge. Both agents performed significantly better against the Honest player than the Monte Carlo (MCE) Player, despite the Honest player being a variation of the MCE Player. PPO08 tended to be more aggressive, preferring to raise over calling, while PPO07 was more conservative, always choosing to call. Neither agent chose to fold anytime during the games.

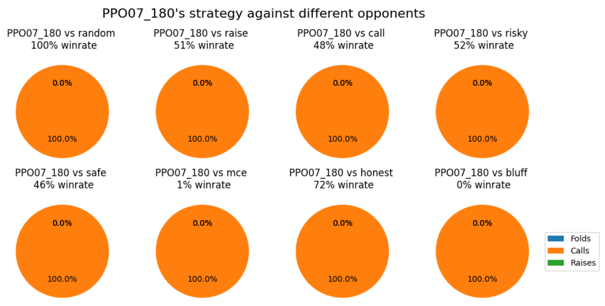

Figure 1: This is the action distribution of the PPO07 player after being trained for 180 episodes.

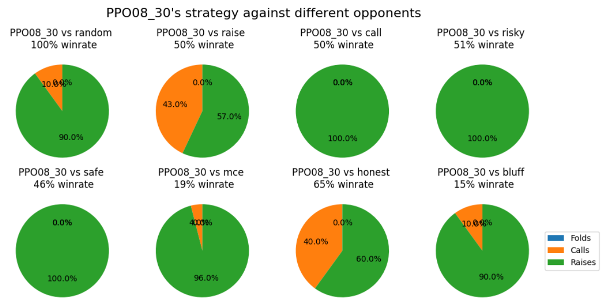

Figure 2: This is the action distribution of the PPO08 player after being trained for 30 episodes.

Opponent	PPO7_180		PPO8_30	
	Win%	ROI	Win%	ROI
RandomPlayer	1.0	99	1.0	99
RaisePlayer	**0.51**	2.8	0.502	**0.6**
CallPlayer	0.48	**0.5**	**0.504**	2.8
RiskyPlayer	**0.524**	1.6	0.516	**3.4**
SafePlayer	0.46	**0.1**	**0.462**	-5.6
MCE Player	0.018	-69	**0.19**	$\mathbf{-62}$
HonestPlayer	**0.72**	**40**	0.654	31
BluffPlayer	0.0	-98	**0.156**	$\mathbf{-69}$

Table 1: Win rate percentages and return on investment (ROI) statistics for PPO07_180 and PPO8_30

4 Discussion

The results presented above are only a tiny sample of the total set of results that we obtained during our experiments. We found that in the beginning of a training run, agents typically achieved significant advances in performance and used diverse strategies, only later coming to collapse as can be seen above into constant (or near-constant) policies. While some policies ultimately achieve reasonable performance, many of these degenerate policies begin to fail. The reason for these failings remains undetermined, though future work may seek to determine whether there was an issue with our procedure for determining quality hyperparameters–because these hyperparameters were determined during testing over a much smaller number of episodes, it is possible that they produce an excessively aggressive update regime which works well early in training but leads to lower-quality collapsed policies towards the end of the training process. We note that it may be difficult to learn an agent that is adverse enough to be able to recognize and exploit their opponent's strategy within the 100 rounds a single game lasts for. However, we also note that it should be possible to learn a dominating strategy just based on the strength of one's hand, as evident by the MCE player.

5 Conclusion

In conclusion, the results obtained from our training tournament produced PPO policies which demonstrated promising performance and learning potential, exhibiting strong win rates against most opponents. It is, however, essential to acknowledge that the effectiveness of PPO is influenced by various factors, some of which may not have been appropriately configured during our testing. Nonetheless, our findings reaffirm the potential of PPO as a viable approach for developing autonomous agents capable of strategic decision-making in complex environments, especially Poker, where we found that even short training runs could produce competent policies which can win against even advanced Monte-Carlo-based players.

6 Detailed Contributions:

6.1 Peter Phan

1. Wrote most of the hand-coded player code
2. Wrangled the orignal codebase (including investigating bugs in the startercode)
3. Wrote learning player (PPO) code
4. Worked on formatting code for submission
5. Worked on commenting code
6. Performed hyperparameter testing and tuning
7. Sifted through results, ran evaluations, and generated graphs and tables
8. Edited final report

6.2 John Raisbeck

1. Organized all team meetings
2. Wrote tournament code
3. Wrote remaining player code
4. Wrote multithreading code
5. Worked on commenting code
6. Wrote validation code
7. Fixed code for cross-platform operation
8. Edited final report

6.3 Giang Nguyen

1. Performed some final result testing to generate graphs
2. Drafted the report
3. Edited final report

6.4 Ruiqi Hu

1. Wrote drafts of the project's abstract and introduction
2. Tested the project to ensure functionality

References

[1] Joel Lehman and Kenneth O. Stanley. Abandoning objectives: Evolution through the search for novelty alone. *Evolutionary Computation*, 19:189–223, 2011. URL `https://api.semanticscholar.org/CorpusID:12129661`.

[2] Michael Johanson Oskari Tammelin, Neil Burch and Michael Bowling. Solving heads-up limit texas hold'em, 2015.

[3] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

A Appendix

A.1 Players

Call player: Similar to the raise player, this player will call whenever possible, otherwise they will fold.

Risky Player: This player model keeps track of the number of raises and calls their opponent plays in a round. At their turn, this player will raise if the ratio of their opponent's raise to call moves is more than one and call otherwise.

Safe Player: Similar to the risky player model this player will also act based on a record of their opponent's moves. This player will fold if their opponent's raise to call ratio is higher than one and call otherwise.

Monte Carlo Player: This player will compute a Monte Carlo Estimation of the probability that their hand is better than their opponent's given their hole and current community cards. If the probability of their hand is better is less than 0.1 then they will fold. If the probability of their hand is better is between 0.1 and 0.9 then they will call. If the probability of their hand is better higher than 0.9 then they will fold. Since the nature of this player is to play for the strength of their hand, we used this model to train our pretrain models in our experiments. The advantage of this model is that it is not as slow as the Emulator player which is good for training, and it was effective in teaching our base models about the strengths of hands.

Honest Player: The Honest player operates similarly to the Monte Carlo player but with different breakpoints for its actions. This player will fold if their estimate tells them they have a less than a 0.5 probability of having a better hand, and will raise otherwise.

Bluff Player: A less conservative version of the Monte Carlo Player.Bluff Player will call if the estimated winrate of their hand is less than 0.75 and raise otherwise.

Emulator Player This player will emulate games against their opponent to determine their action. For each valid action this player can play, it will emulate playing n games and aggregate the results. Each simulation starts at the current game state, uses the opponent's exact model, and runs until the round finishes. The Emulator player will pick the action that gives the highest average return from the simulations. In our testing, we found this model to be the most difficult to beat.

A.2 Markov Decision Process

An MDP is defined by the tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \eta, r, \gamma)$, where $\mathcal{S}$ is the state space, A is the action space, P is the transition dynamic $P : (\mathcal{S} \times \mathcal{A} \times \mathcal{S}) \to [0, 1]$, $\eta(\cdot)$ is the initial state distribution, r is the reward function $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, and $\gamma \in (0, 1)$ is the discount factor.