Monte-Carlo Tree Search in Limit Texas Hold'em

Zachary Dawson

Problem

Limit Hold'em is a hold'em variant that allows for bets and raises of only one size. This creates a relatively simple game compared to the more popular variant of no-limit hold'em, which allows for any size bet up to the size of the player's stack. Each hand in limit hold'em allows for the player to either check, call, raise, or fold depending on the state of the game. The object of any poker playing algorithm is to choose actions that improve expected value. This value is normally quantified in big blinds (BB) but could also be the magnitude of money lost, which would not take into account the size of the blinds. Hold'em is an incredibly interesting and complex game, allowing for many strategies that can all be profitable.

Poker is played with each player receiving two hole cards that are hidden from opponents. One player must post a small blind and the other a big blind. After this, the small blind has the opportunity to raise one big blind, call the big blind, or fold. Generally you raise with good hands and fold with poor hands. Calling can be used if you want to draw for a better hand. As long as no player folds, 3 cards are dealt face up which become the communal cards. The betting phase is repeated again followed by another communal card. This is repeated one more time, leading to 5 total communal cards and 4 betting rounds. These betting rounds are generally referred to preflop, postflop, turn, and river. Whoever has the best hand at the end of the 5 betting rounds wins the pot, or you win the pot if the opponent decides to fold.

The state of a game is relatively simple, as we only know our hole cards, the communal cards, and what are our current possible actions. There are many things that we keep track of however in an attempt to ascertain how likely we are to win in any given scenario. That includes the opponents previous actions, the strength of our hand in relation to all hands, and some board characteristics. These features will allow us to predict how likely we are to win the hand, which will help inform what action to take next. I will attempt to use this information along with Monte Carlo Tree Search (MCTS) to create a poker bot that can beat a rules based agent.

References

The original idea for a project creating an opponent model for poker came from AlphaGo, a perfect information game. They started their training process by training a supervised machine learning model using match histories of high amateur players. They then improved upon their initial predictions using partial solving and Monte Carlo Tree Search to create their final agent. Extending this to an imperfect information game would require a similar process, although much more rudimentary.

Opponent modelling has been done many times before in poker. In Learning Strategies for Opponent Modeling in Poker supervised learning models were trained successfully to predict specific players actions given hand histories from that players games. There were also two books Computers and Games and Algorithms and Assessment in Computer Poker that went through various metrics and techniques for creating poker AI. Many of the metrics used for features are calculated using algorithms from these works.

Various poker bots have been made using MCTS and various other methods. I was informed mostly by a dissertation of a previous Cambridge student and a research group from the Netherlands. I made sure to avoid reading any code available and chose to use a different simulation algorithm, which is the meat of the MCTS algorithm. The Cambridge student's hand histories were actually used to create the supervised learning models that will be described in the methods. There were various other resources that aided my dive into poker algorithms including Computers and Games and Algorithms and Assessment in Computer Poker. Both of these informed what metrics would be worth extracting from the game state to inform our algorithm.

The two libraries that enable the training and evaluation of this bot are called eval7 and RLCard. Eval7 is a hand equity calculator and hand strength evaluator. It is used in the simulation to evaluate equity versus various ranges as well as generating new states that require random cards. RLCard is a reinforcement learning gym that can be used for various card games.

Data

Raw hand histories from Poker Academy Pro are the unprocessed data of this project. A single hand in this form looks something like this:

```
                                                    Not in Raw Data
******************************
   Poker Academy Pro #22,679
   Limit Texas Holdem ($1/$2)                       Preliminary Hand Info
   Table SimpleBot Logger
   November 24, 2010 - 13:23:10 (GMT)

    1} David              (sitting out)
    5) LoggingSimpleBot      $996  5c 5d            Big Blind
    6) SimpleBot *           $1,004  4s Ts          Dealer

   SimpleBot posts small blind $0.50
   LoggingSimpleBot posts big blind $1              Preflop Action
   SimpleBot calls $0.50
   LoggingSimpleBot checks

   FLOP:  Th 4c 6h                                  Flop Cards
   LoggingSimpleBot checks
   SimpleBot bets $1                                Flop Action
   LoggingSimpleBot calls $1

   TURN:  Th 4c 6h 5s                               Turn Cards
   LoggingSimpleBot bets $2
   SimpleBot raises $2
   LoggingSimpleBot raises $2                       Turn Action
   SimpleBot raises $2
   LoggingSimpleBot calls $2

   RIVER:  Th 4c 6h 5s Jc                           River Cards
   LoggingSimpleBot bets $2
   SimpleBot raises $2
   LoggingSimpleBot raises $2
   SimpleBot raises $2                              River Action
   LoggingSimpleBot calls $2
   SimpleBot shows 4s Ts
   LoggingSimpleBot shows 5c 5d

   LoggingSimpleBot wins $36 with Three of a Kind, Fives   Conclusion
******************************
```

These histories contain all the necessary information to extract any feature, as all of the information is available, even the opponents hole cards. Various python scripts were used to extract relevant features from each hand and write each decision point to a dataframe. In an attempt to emulate previous work, The features in Table 1 were intended to be extracted. Unfortunately, many of these features could not be extracted as they were too computationally expensive. PPOT and NPOT required iterating over all possible combinations of run-outs of communal cards as well as opponent possible hands, which ends up being extraordinarily large, so these features could not feasibly be extracted or computed in real time. As such, winning probability is a much simpler approximation than the formula laid out by Billings, which used both PPOT and NPOT in the calculation. Hand strength and rank are determined using eval7, a python hand evaluator, which quickly determines how strong your hand is relative to all other hands. All other features are made simply by saving the previous actions from the hand and determining raise amounts and stack committed for your opponent and yourself. All feature extraction was run using python scripts that dumped into a csv that can be seen below. The final list of features extracted can be seen in Table 2. The features extracted include various attributes about hand strength, the actions of each player, and some attributes about the common

Table 1: Candidate features

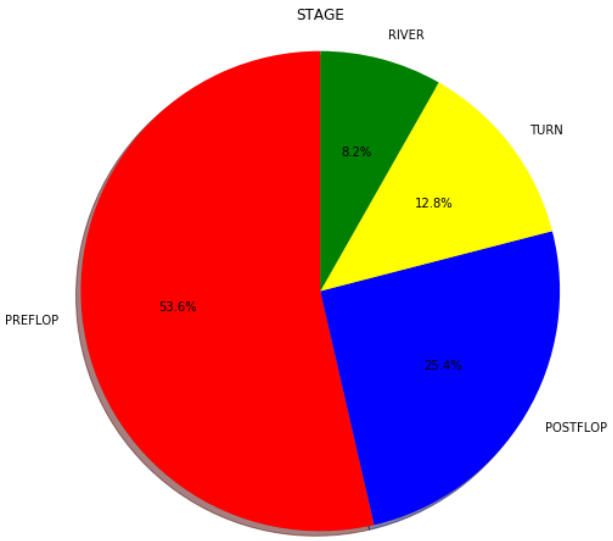
Id	Explanation
1	Hand strength
2	PPot
3	NPot
4	Whether the player is dealer or not.
5	Last action of the opponent (null, call or raise)
6	Last action of the opponent in context (null, check, call, bet or raise)
7	Stack (money) committed by the player in this phase
8	Stack committed by the opponent in this phase
9	Number of raises by the the player in this phase
10	Number of raises by the opponent in this phase
11	Hand rank
12	Winning probability
13	Hand outs
14	Number of raises by the player in previous phases
15	Number of raises by the opponent in previous phases
16	Highest valued card on the board
17	Number of queens on the board
18	Number of kings on the board
19	Number of aces on the board

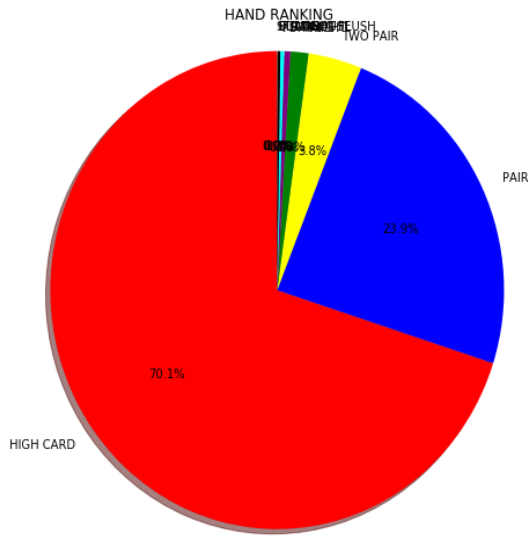
Table 2: Features

action	int32
stage	int32
dealer	int64
hand_strength	float64
hand_rank	int64
opp_last_action	int32
my_last_action	int32
my_stack_committed_curr_phase	float64
opp_stack_committed_curr_phase	float64
my_num_raises_curr_phase	float64
opp_num_raises_curr_phase	float64
my_num_raises_total	float64
opp_num_raises_total	float64
num_outs	int32
winning_prob	float64
highest_card	int32
num_aces	int32
num_kings	int32
num_queens	int32
prev_action	int32
prev_stage	int32
prev_dealer	int64
prev_hand_strength	float64
prev_hand_rank	int64
prev_opp_last_action	int32
prev_my_last_action	int32
prev_my_stack_committed_curr_phase	float64
prev_opp_stack_committed_curr_phase	float64
prev_my_num_raises_curr_phase	float64
prev_opp_num_raises_curr_phase	float64
prev_my_num_raises_total	float64
prev_opp_num_raises_total	float64
prev_num_outs	int32
prev_winning_prob	float64
prev_highest_card	int32
prev_num_aces	int32
prev_num_kings	int32
prev_num_queens	int32

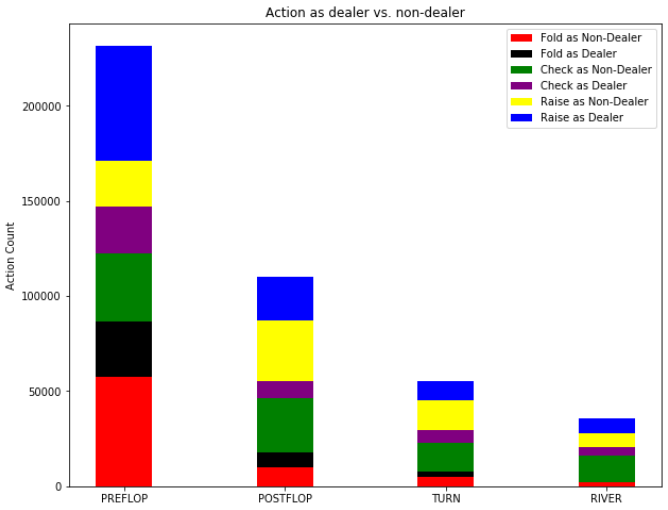
cards. For each feature, I also included the value from the previous decision, to allow for some temporal analysis.

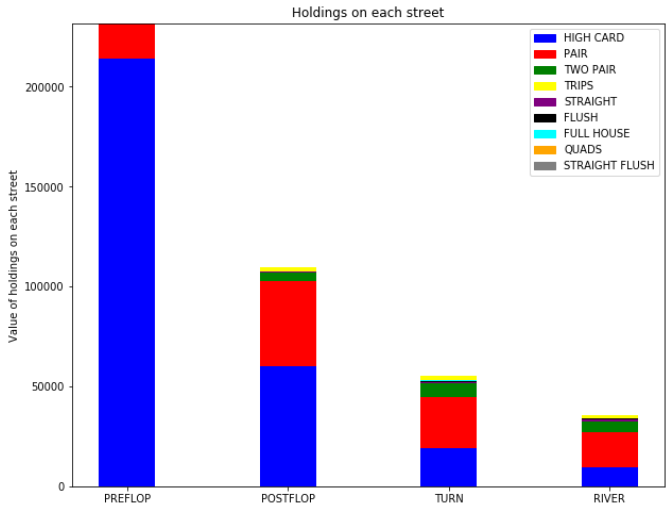
The structure of this data made prediction very challenging. The data was imbalanced between stages of the game as well as the hand rank for each player. In figure one we see that over 50% of decisions are for preflop actions and a slow taper to 8% of river actions. Hand ranks are even more imbalanced, with 73% of examples being in the top 2 of 9 classes. Of over 400,000 decisions, only 8 of them included players with the highest rank, Straight Flush.





Action distribution is well dispersed among each option for the dealer and non-dealer but the distribution of hand ranks are very imbalanced with very few instances of players improving beyond two pairs.





Methods

Opponent Model

In order to mitigate the imbalanced hand rank distribution, I resampled the original data to include equal numbers of each hand rank and also created a new hand rank that includes Quads, Full House, and Straight Flush as they were so rare. This resampled data is only used for the hand rank model, not the action model.

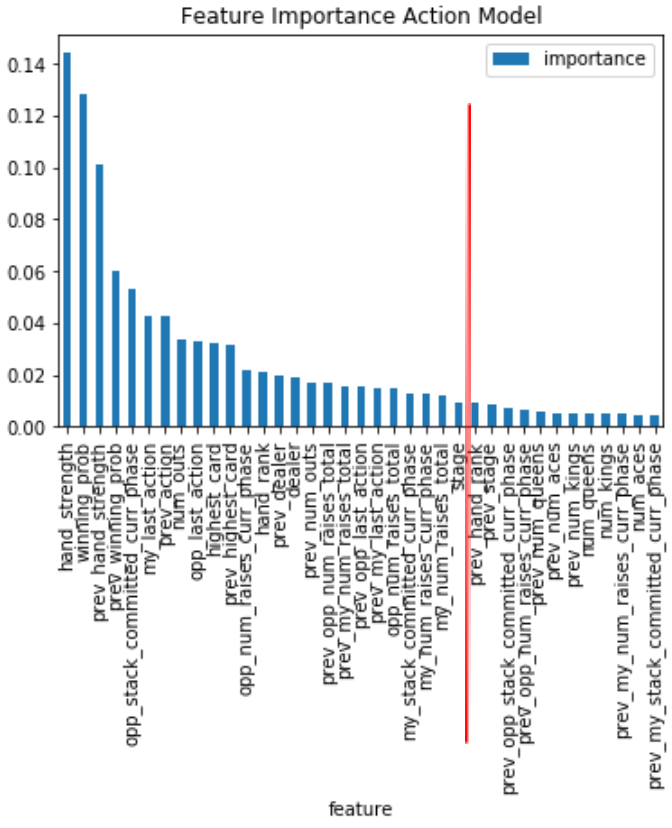
Hand Ranks before Resampling

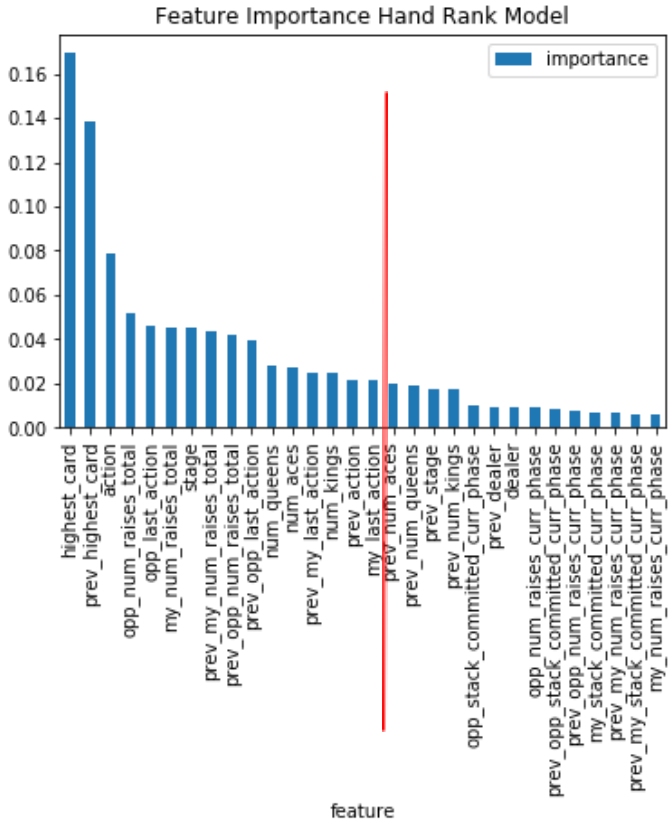
0	302796	High Card
1	103422	Pair
2	16455	Two Pair
3	5531	Trips
4	1883	Straight
5	1205	Flush
6	956	BEST

Hand Ranks after Resampling

6	10000	BEST
5	10000	Flush
4	10000	Straight
3	10000	Trips
2	10000	Two Pair
1	10000	Pair
0	10000	High Card

After resampling, I created an Extra Tree Classifier, which is an ensemble model within sklearn to determine feature importance for predicting both action and hand ranks, selecting the top 20 and 17 features respectively.





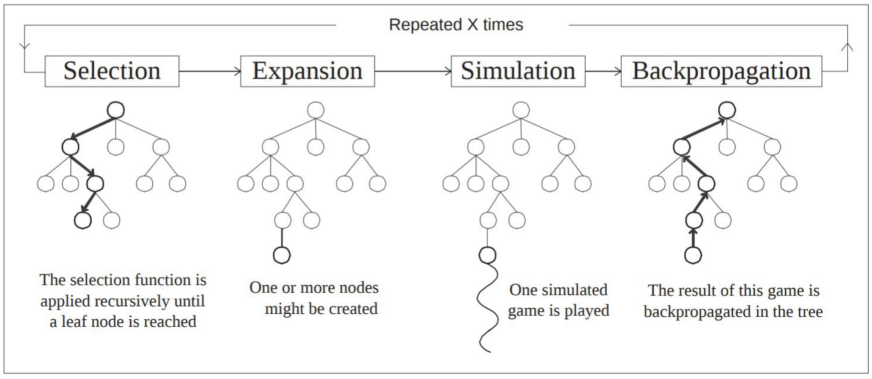
Using these selected features, I train classifiers for both actions and hand ranks utilizing a random forest classifier, extra tree classifier, gradient boosting classifier, and multi-layer perceptron classifier.

Monte Carlo Tree Search

In order to make a profitable poker bot, we must develop some decision making framework that, given the current game state, will make a decision that when repeated, will lead to a profit. The repeated piece is key since poker in an imperfect information ame and any decision could always be unprofitable in a given instance but profitable if repeated many times in a row.

A reasonable method to solve this problem would be MCTS. This allows for exploration time of the game tree and multiple repetitions to be made. This can help to mitigate any problem that might come from running only one simulation or some other probabilistic model.

MCTS is an algorithm with 4 main components. A tree starts from the root and runs a selection algorithm until an unexplored node is reached. Then a new random new node is expanded and simulation is run from that node to completion. Win counts are then updated up the tree in a process called backpropagation.



UCT is a selection strategy that attempts to optimize exploration as well as exploitation. This means that it prioritizes unexplored paths as well as those it believes to be the most profitable. The UCT formula is defined below.

$$\frac{w_i}{s_i} + c\sqrt{\frac{\ln s_p}{s_i}}$$

- w_i : this node's number of simulations that resulted in a win
- s_i : this node's total number of simulations
- s : parent node's total number of simulations
- c : exploration parameter

Normal simulation strategies employ very basic strategies in order to simulate quickly. Simulation in an imperfect information game is even more challenging because you cannot predict opponent actions particularly well without knowing their hand. Using opponent models, we can predict the probabilities of actions and hand ranks for the simulation.

Results

Opponent Modelling

Initial Model Evaluation

In order to determine a baseline for various types of classifiers, I trained 4 models: an extra trees classifier, random forest classifier, gradient boosting classifier, and a simple multi-layer perceptron classifier. I trained each model with KFold cross-validation with k=10. The extra tree classifier and random forest classifier exhibit the most promise for both the action and hand rank model. The neural networks did not initially exhibit much promise, but I used a simple architecture with very few hidden nodes in each layer. I decided to try to improve the neural networks performance for the action model by increasing its complexity. In table 4 you can see that after increasing the number of parameters in the neural network by a factor of 2 in each subsequent model, the accuracy does improve, but not to a level that is comparable to the other classifiers, especially considering those models have not been tuned at all.

Table 3

```
Training Action Model:
Extra Tree Classifier:
        Mean Accuracy: 73.57%
        Standard Deviation: 0.55%
Random Forest Classifier:
        Mean Accuracy: 74.71%
        Standard Deviation: 0.54%
Gradient Boost Classifier:
        Mean Accuracy: 65.75%
        Standard Deviation: 0.54%
Neural Network:
        Mean Accuracy: 49.36%
        Standard Deviation: 0.68%

Training Hand Rank Model:
Extra Tree Classifier:
        Mean Accuracy: 46.99%
        Standard Deviation: 0.67%
Random Forest Classifier:
        Mean Accuracy: 47.13%
        Standard Deviation: 0.64%
Gradient Boost Classifier:
        Mean Accuracy: 34.06%
        Standard Deviation: 0.47%
Neural Network:
        Mean Accuracy: 29.38%
        Standard Deviation: 0.69%
```

Table 4

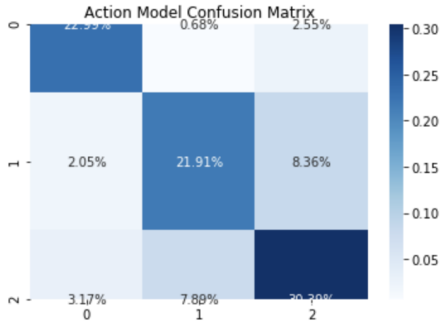
```
Training Action Model:
Neural Network:
        Mean Accuracy: 49.36%
        Standard Deviation: 0.68%
Neural Network1:
        Mean Accuracy: 50.10%
        Standard Deviation: 4.75%
Neural Network2:
        Mean Accuracy: 59.85%
        Standard Deviation: 1.02%
```

Hyperparameter Tuning

In an attempt to squeeze any further performance from the classifiers, I tuned the random forest classifier and extra tree classifier. Using various depths and number of estimators, the accuracy for the action based model improved by about 0.75%. Ideally, that number would be higher but in a game with so much variation in play with so little information, 75% accuracy is reasonable.

The same process was repeated for the hand rank model using a random forest, as well as another model for both action and hand using an extra trees classifier. The extra trees classifier did not perform better than random forest for either model, and the random forest did not improve the hand rank model by any significant measures.

```
Action Model Performance
Accuracy= 0.7529714285714286
Error= 0.24702857142857138
              precision    recall  f1-score   support

        fold       0.81      0.88      0.84      4590
       check       0.72      0.68      0.70      5657
       raise       0.74      0.73      0.73      7253

    accuracy                           0.75     17500
   macro avg       0.76      0.76      0.76     17500
weighted avg       0.75      0.75      0.75     17500
```

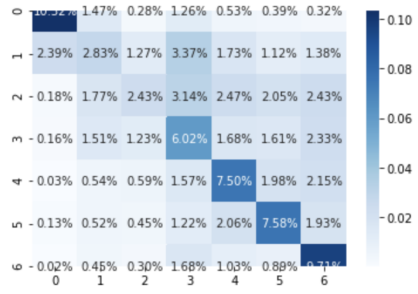


Monte Carlo Tree Search

In order to evaluate the MCTS agent, I created a series of experiments that would be run between the MCTS agent and a premade rules based bot that is included in the RLCard library. Its decisions are informed by a massive block of if statements that are meant to create a reasonable strategy. A strategy like this should be easily exploitable with a strategy that learns as it plays, but the MCTS agent is built to play against a player with a completely different strategy, so results could have been improved had hand histories of this rules based agent had been used instead of a different player.

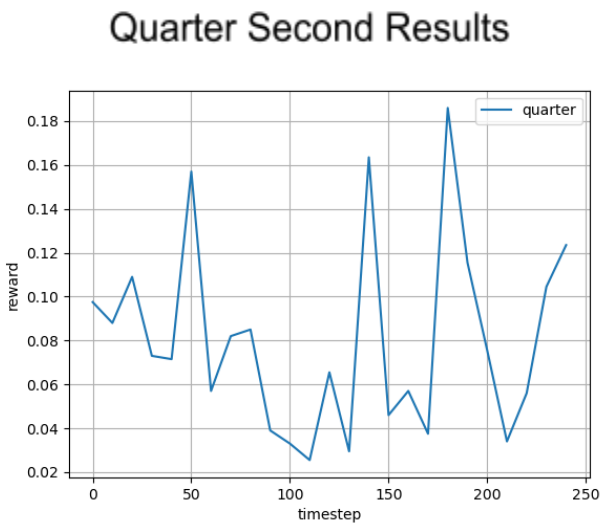
```
Hand Rank Model Performance
Accuracy= 0.4638857142857143
Error= 0.5361142857142858
              precision    recall  f1-score   support

   High Card       0.78      0.71      0.74      2550
        Pair       0.31      0.20      0.24      2466
    Two Pair       0.37      0.17      0.23      2534
       Trips       0.33      0.41      0.37      2543
    Straight       0.44      0.52      0.48      2512
  Full House       0.49      0.55      0.51      2430
        Nuts       0.48      0.69      0.57      2465

    accuracy                           0.46     17500
   macro avg       0.46      0.46      0.45     17500
weighted avg       0.46      0.46      0.45     17500
```

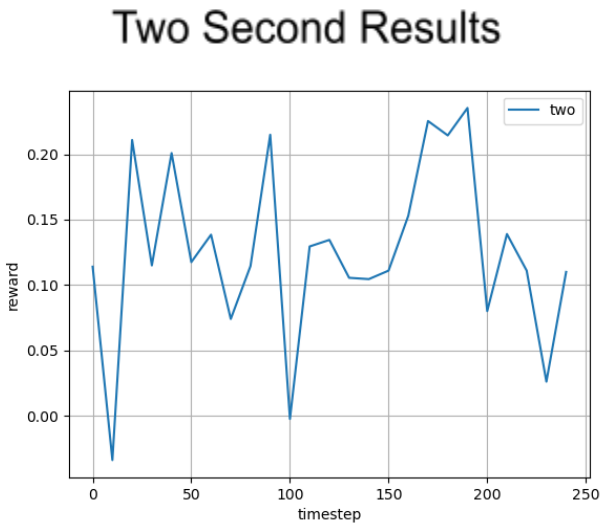
```
<matplotlib.axes._subplots.AxesSubplot at 0x2ab99057d310>
```



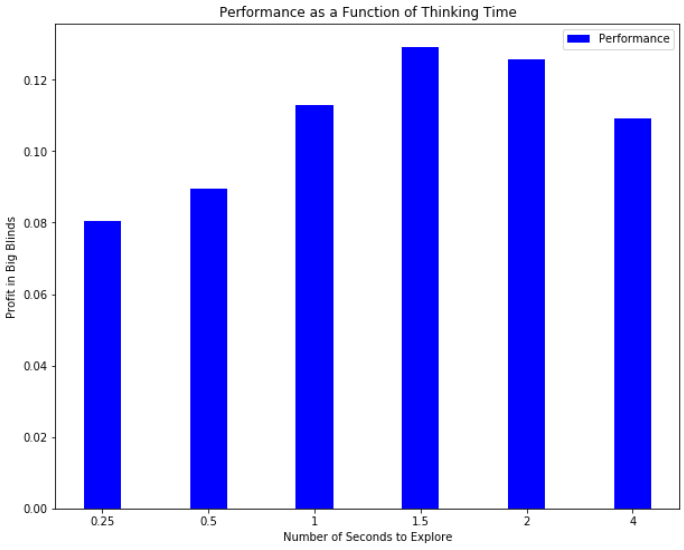
I ran 6 experiments with varying thinking times for the MCTS agent each time starting with 0.25 seconds and ending with 4 seconds. I expect that as the time given to for the algorithm to 'think' will lead to increased profitability. Eventually the profit should plateau as since there will be no improvements that can be made against this specific opponent. The output of these experiments come in the form of a graph that shows the reward as a function of time through the experiment. For example, below the results for 0.25 seconds and 2 seconds can be seen below.

These graphs show the average reward in each of the 25 tournaments between the MCTS agent and the rules based agent. Each tournament was 1000 hands totalling 25,000 hands for each exploration time. As this library was developed for reinforcement learning models, the graphs generally start poor and then stabilize at some average profitability. Since I used a more basic tree based algorithm, the results of each tournament are independent of the last, so there is





no improvement over time. This makes any generalizations from these graphs challenging, but you can see that the MCTS agent that explored for 2 seconds had generally better results than the MCTS agent that explored for only 0.25 seconds. The average profit in big blinds across all experiments can be seen below.



Discussion

The results of the MCTS agent were promising since it was able to beat the rules based bot with all exploration times. My first attempt was unable to beat an agent that chose actions at random, so there has been significant improvement from the beginning. Most of this improvement came from changing the simulation strategy. The original simulation strategy was far too simple to capture the complexity of the game. However, the simulation strategy is also the area where the most improvement can still be made. The agents do not actually play the hand to completion. Instead, a probability of winning is estimated for each player based on my hole cards and the predicted cards of the opponent. The expected value is determined to be the size of the pot times the probability of winning. Although this is still far too simple to understand the full nuance of the game, especially in the preflop and postflop stages of the game, it has managed to capture enough information to make a profitable poker bot.

The most interesting result, besides that the bot was profitable at all, was that its profits began to drop after 1.5 seconds of exploration. This was seen in other MCTS papers, but I am unsure of why it occurs. Perhaps we explore branches that are poor and the hand rank model underpredicts the strength of the opponents hands, so we believe we will win when we actually won't. Given the current tooling I have for examining decisions at the moment, there is not much I can do to investigate it.

Any improvements to the algorithm would require more tools to examine each decision. Unfortunately there is nothing built into the RLCard environment to see each decision. I attempted to create a DataFrame that would store the predictions of the MCTS agent vs. the supervised machine learning model that predicts actions based on previous actions. The model did not perform as it did in testing, likely because some metric that I was calculating or keeping track of was not correct. A tool like this would be invaluable in seeing which type of actions are leading us to lose value, whether that be overfolding or being too loose with poor hands. Without information like this it is nearly impossible to know what the next step to improving the model would be.

Any future attempt to create poker intelligent agents would not use MCTS. This is a relatively archaic method, as the literature I found was from the early 2000's. Using more

standard reinforcement learning based models stands to create far superior players while also requiring less thinking time for each action.