Applications of Computational Intelligence to Pineapple OFC Poker

Author: Peter Scott Williamson
Advisor: James Glenn

May 1, 2019
## Contents

1 Abstract 1

2 Introduction 2
   2.1 Gameplay 2
      2.1.1 Points and Scoring 2
      2.1.2 Fantasyland 3

3 Pineapple Game Architecture 3
   3.1 Pineapple Environment 4
      3.1.1 Pineapple Board and Hand 4
      3.1.2 Pineapple Agent 4
   3.2 Fantasyland Calculator 5

4 Agents and Agent Strategies 5
   4.1 Random Agent 6
   4.2 Heuristic Agents 6
      4.2.1 MCTS Determinization 6
      4.2.2 MCTS with Randomization 7
   4.3 Neural Network Agent 7
      4.3.1 Training Data 7
      4.3.2 Gamestate Encoding 7
      4.3.3 Performance 9

5 Results 9
   5.1 Random Agent 9
   5.2 MCTS Determinization Agent 9
   5.3 MCTS with Randomization Agent 9
   5.4 Neural Network Agent 10
   5.5 Observations 10

6 Further Work 10
   6.1 Reinforcement Learning Agent 11
1 Abstract

Applications of computational decision-making strategies are often tested by their performance in playing games. In general, games are hard for computers to solve when either the number of moves for a computer to make is large, there are hidden information and unknowns, or there are multiple players in a game. The game of Pineapple OFC Poker (Pineapple) provides a challenge to computers due to both the uncertainty of information on which future moves will be available, as well as the large branching factor the game sees after each move. This project aimed to apply several computational techniques to develop agents that are capable of playing Pineapple at a high level.

With the goal of creating multiple AI agents to competitively play Pineapple against others, a game framework for Pineapple from which both AI agents as well as humans could play through was developed. A random move agent was built to be a benchmark for the other agents, followed by a probabilistic agent using two variants on a Monte Carlo tree search algorithm (MCTS) that could incorporate the incomplete information and randomness of Pineapple to an otherwise deterministic process. Using sampled positions from the MCTS agents, a neural network was trained. It was found that the random agent scored 0.04 royalty points per hand (RPH), that in the solitaire variant the deterministic MCTS agent performed at a rate of about 4.35 RPH, that the random MCTS agent scored about 4.76 RPH, and that the neural net agent (trained by the random MCTS data) scored about 4.24 RPH. An expert human player is able to achieve slightly more than 5 RPH in solitaire.
2 Introduction

This project aims to create AI agents to competitively play the game of Pineapple Open Face Chinese Poker (Pineapple OFC/Pineapple) using a variety of computational intelligence techniques. Pineapple is a card game played by up to three players, and though this work focused on the solitaire variant, minor alterations to the techniques used may extend the results to both the two- and three-player variants.

2.1 Gameplay

Pineapple is a variant of open face Chinese poker that originated in Finland during the mid 2000s, and was popularized in Russia several years later. The game comprises three different hands (also referred to as rows) called the front, middle, and back, in which players place cards down in each to create three poker hands as strong as possible determined by traditional poker hand rankings. The front hand has three spots for cards, and the middle and back both have five spots for cards.

Players initially receive and place five cards face up across the three hands, and then take turns playing four sets of three cards of which two are placed face up (with the third discarded face down). Once a player places a card on a hand (or discards the card), its position may not be changed. Each hand must have an absolute strength less than or equal to the row below it (i.e. the front hand may not be stronger than the middle hand which may not be stronger than the back hand). Should a player’s board have a hand whose strength is greater than the one below it, the player’s board fouls and each hand is forfeited. Once all players have placed thirteen cards across the three hands, scoring is conducted.

2.1.1 Points and Scoring

In each round of Pineapple, players compete against one another for points (in the solitaire version of the game, players simply track the number of royalty points earned) which can be earned in one of two ways. Firstly, players may earn one point for each hand that beats their opponents corresponding hand. For example, if player \( P_1 \) has a back hand of a flush, and player
Williamson Applications of Computational Intelligence to Pineapple OFC Poker

$P_2$ has a back hand of three of a kind, then $P_1$ would receive one point for winning that hand. Secondly, players may earn “royalty points” based solely on their own board. These royalty points are given to hands which are challenging to make, with values as shown below. Royalty points are not earned should a player’s board foul.

![Table showing royalty points](image)

To score the hand, each player earns the value of the difference of their total points and their opponent’s total points (with solitaire solely concerned with royalty points earned per hand). For instance, if player $P_1$ wins the back and front rows, and $P_2$ wins the middle row (with no royalties earned by either player), then $P_1$ would win one point, and $P_2$ would lose one point. If $P_1$’s back row were instead a flush, and $P_2$’s back row were a straight, then $P_1$ would win three points in total, and $P_2$ would lose three points.

Finally, player might also “scoop” the round by winning all of the hands against an opponent. In this case, the player will earn an additional three points. Thus, when a player fouls (and forfeit every hand), her opponent will scoop the round with any board that does not also foul. As a result, players must balance going for high value hands to score royalties against the likelihood of fouling.

2.1.2 Fantasyland

Should a player have a pair of queens or stronger in the front hand and have a board that does not foul, then the player will enter “Fantasyland” in the next hand. During Fantasyland, a player is dealt fourteen cards all at once (while other players not in Fantasyland must play out the hand as normal). This is highly advantageous, as a player is able to guarantee a board that does not foul while also maximizing the number of royalty points possible with those fourteen cards. To remain in Fantasyland, the player must achieve four of a kind or stronger in the back row, or three of a kind in the front row.

3 Pineapple Game Architecture

Before computational intelligence techniques could be applied to Pineapple OFC Poker, an environment from which the agents could play in needed to be developed. The environment
takes in some number of agents, generates a board for each, and deals new sets of cards to the agents after the agents place previously dealt cards on their respective boards.

3.1 Pineapple Environment

A generic pineapple environment class was implemented expecting only logic for how to execute one round at a time so that a variety of environments (such as solitaire or two-player) could be used with similar baseline logic. This environment extended the work of Dex Groves, who started work on an implementation for an environment for OFC Poker (a variant of Chinese Poker in which players are dealt one card at a time). After checking for if an agent is in Fantasyland (and if so, dealing the first fourteen remaining cards in the deck to each agent in Fantasyland), the environment deals cards to the agents sequentially such that an agent is given knowledge of where the cards given to the previous agent were played.

3.1.1 Pineapple Board and Hand

The pineapple board class holds logic for placing cards on each of the various hands, as well as keeping track of discarded cards. Additionally, logic for finding legal moves as well as determining royalty points earned, fouling, and if the game is over was implemented. To calculate both royalty points and absolute hand strength, a library called “deuces” was used to efficiently calculate a hand’s ranking. An example of an environment an the end of a game can be seen below.

3.1.2 Pineapple Agent

A generic class was made for agents so that each agent could be passed into the pineapple environment without having the internal logic of the environment depend on the agent. The
generic agent class simply defines how to play the starting five cards, how to play each set of three cards, and how to play Fantasyland cards. Additionally, each agent keeps track of their own Fantasyland state.

### 3.2 Fantasyland Calculator

While the playout of pineapple for an agent not in Fantasyland is computationally challenging, one is able to solve for the correct card placements for when an agent is in Fantasyland through brute force with pruning. Though agent may use the Fantasyland calculator when playing one another, the primary purpose of the calculator was to determine the expected RPH earned from Fantasyland, as well as the probability an agent will remain in Fantasyland. It was found that Fantasyland achieves approximately \(12\) RPH, and that there is approximately a 10\% probability of staying in Fantasyland given optimal play. When later solving for an agent’s RPH in solitaire, these values were used in favor of random Fantasyland playouts, as Fantasyland sees high RPH variance.

### 4 Agents and Agent Strategies

While some work by others has been made into learning Pineapple OFC Poker, published literature on the topic is essentially nonexistent. In this project, we attempt to apply established techniques from deterministic games with small branching factors to Pineapple, which comprises both randomness and a large branching factor. When eventually implementing a neural network to play Pineapple, we encoded the gamestate in a variety of ways to reduce the number of features that the network needed to independently learn.

For the MCTS agents and the neural net agent, the initial five cards were played using a heuristic based model to aggressively play for Fantasyland when possible. A heuristic was used for the starting hand as the breadth of playing five cards over three boards required computational power beyond the scope of this project.\(^1\) Strong hands identified for the starting heuristic are:

1. 4 of a kind
2. full house
3. 5 cards to a flush (or straight flush)
4. 4 cards to a flush
5. 5 cards to a straight
6. 3 of a kind
7. 3 cards to a flush with an outside pair
8. 4 cards to a straight
9. 2 pair
10. pair
11. 3 cards to a flush
12. gutshot to a straight
13. 3 cards to a straight
14. high cards only

\(^1\)Though there are only 232 possible placements of 5 starting cards across the 3 hands, determining which of the placements is best without running several thousands of a MCTS (or other approach) per placement is challenging. Using a computational approach such as MCTS to place the first five cards would increase the total time of playing each round by a factor of over 100, as even if strictly bad hands are pruned before searching, the remaining contenders will take an extensive amount of time to filter through.
4.1 Random Agent

Initially, a completely random agent is implemented in order to test expected output of a poorly played hand, as well as to benchmark the speed at which computations might be made on a local machine. Given a set of cards, the random agent selects one move among all possible legal moves at random. As expected, this agent performed quite poorly, and fouled 69% of rounds it played. This random agent was primarily used to benchmark other agents in a computationally non-intensive way, as well as to see what improvement the other agents made over simply choosing random moves.

4.2 Heuristic Agents

As noted by Browne et al., the Monte Carlo Tree Search methodology is typically used to train AI on deterministic games with perfect information. In an attempt to adapt MCTS to Pineapple, a stochastic game with incomplete information, two different techniques were used on the four sets of three cards to be played.

4.2.1 MCTS Determinization

Though Pineapple is a stochastic game with imperfect information, it can be mapped to a deterministic game by fixing the order of the deck of cards and then running the standard MCTS algorithm. The method known as “Determinization” is when several instances of a deterministic version of the game are sampled in order to gain information on what should be the correct move for the stochastic version of the game. This method has been applied to other games with hidden information, such as “Dou Di Zhu” or “Phantom Go,” with various degrees of success.

Though MCTS with determinization did improve performance over the random agent, there was a significant drawback to how sampling was performed. Consider the following scenario in which there are two moves remaining in the round. Suppose that the agent can make move $m_1$ in which the agent guarantees a score of 2 RPH for the round, or that the agent can make move $m_2$ that sees the agent get a score of 20 RPH 25% of the time, and sees the agent foul the other 75% of the time. Strictly sampling the deterministic processes will have the agent make move $m_1$ for 75% of deterministic samples (as the agent will not foul these rounds), and it will have the agent make move $m_2$ for the 25% of samples in which the agent would not foul.

While move $m_1$ is not bad per se, it is clearly not as strong of a move as $m_2$ as expected values of both moves are $E[m_1] = 2 < 5 = E[m_2]$. Determinization is a strong technique when one is concerned with simply winning or losing (values of 1 and -1), as this sees the most likely to win moves selected more often. In comparison, games that have variable point values between 10 and 200 were used as the total number of deterministic games to be played out, though it was found that after around 50 games the move selections did not change greatly.
outcomes that have potential moves giving unlikely outcomes with high returns are not usually selected for.

4.2.2 MCTS with Randomization

To identify high value boards that infrequently occur, randomization was instead built directly into the Monte Carlo tree search algorithm itself (as opposed to randomizing the deck then running the MCTS algorithm on a determinized version of each position). To limit the size of the branching factor, which would greatly diminish the effectiveness of a MCTS, only the moves for cards were recorded in nodes and not the cards themselves. While this seems as though it might lose information as to what sorts of moves are good (e.g. placing cards that increase the number of cards to a flush would be a good play, whereas making the same move with cards of a different suit would be a bad play as it would ruin the possibility of making a flush), when aggregated over several thousand iterations the best moves often filtered to the top. Thus, it was seen that introducing randomness directly into the MCTS algorithm greatly outperformed the determinization model even when fewer total iterations were used.

4.3 Neural Network Agent

With the goal of eventually creating an reinforcement learning agent, a simple neural network agent was first created. The purpose of this agent was twofold. Firstly, how various gamestates of positions in Pineapple are encoded lead to vastly different results over a fixed training data size. Secondly, an agent’s unsupervised learning might be accelerated if it is first supervised over a data set to learn the basic features of the game it is trying to solve.

4.3.1 Training Data

When attempting to train the neural network on optimal moves, initially 1000 positions were sampled and labeled by hand. However, this process was highly time consuming and did not lead to a large enough data set to train the neural net sufficiently. To gather a larger sample size, 140000 games were played out by the MCTS heuristic agent with randomization, and from each game a position was sampled and fully encoded.\(^3\) A disadvantage to this approach is that the MCTS agent with randomization does not necessarily play the optimal move for each position, which in turn may lead the neural network to learn sub-optimal features.

4.3.2 Gamestate Encoding

The set of inputs into the neural network greatly influenced the accuracy of its outputs. As a first attempt, input features with the following structure were fed into the neural network:

\(^3\)A fully encoded gamestate comprises all of the cards in the front hand, middle hand, and back hand based on rank from 1 through 52. A value of zero was used to indicate an empty spot in that hand.
While these inputs captured a large majority of the information in the game, this structure abstracted away many of the features that the neural net would like to know (such as if a card is a pair or not) and as a result the neural network barely performed better than the random agent. Due to the structure of these inputs, the neural network likely could not distinguishing between ranks, as a partial hand of 4/5/6 would be interpreted very similarly to a hand of 5/5/5 even though this are vastly different hands. Finally, this input dimension’s size was simply too large as a 146 dimensional vector was needed to represent these features.

A second input set was then attempted with the goal of minimizing the number of features that the neural net would have to independently learn as well as reduce the dimension of the input space. These input features had the following structure:

[front features, mid features, back features, card set pairs, first card features, second card features, third card features]

This feature set approach allowed the neural network to learn the features of the game more easily. In this approach, front features had information on if there were pairs or three of a kind, as well as how strong they were. This was done by having one bit indicate if there were three of a kind or not (with a higher value indicating a stronger three of a kind), and using three additional bits to indicate pair strength. These three bits dispersed different pairs over “low pairs,” “mid pairs,” and “high pairs.” Pairs would be combinations of values of these bits to capture both strength as well as point value. For example, a pair of jacks would see a value of 0.4 in the “mid pairs” indicator, and a value of 0.6 in the “high pairs” indicator.

Both mid features and back features captured similar information to front features, but also had indicators for number of cards to either a straight or flush, as well as the number of outs that remained for both. Additionally, to capture hands ranging from two-pair to four of a kind, seven additional bits were used to represent these hands. The first bit represented the highest $n$-of-a-kind (0.5 for a pair, 0.75 for three of a kind, and 1.0 for four of a kind) with the following two sets of three bits indicating card values in the same manner that the front features did. Finally, card features comprises how each card interacts with each hand on the board.

By abstracting away the actual details of the board to features that are important (e.g. four cards to a flush vs. actual card values), the neural network was able to learn to play like the MCTS agent somewhat well. A large reason for the improvement of the neural network on this encoded compared to prior inputs is that some of the features are essentially “prelearned” for the neural network (such as if two cards are a pair or not). This in turn allow for the network to understand how cards to be played would interact with a board with a far smaller training data set that would otherwise be required.
4.3.3 Performance

Though the neural network was able to play Pineapple somewhat well, it struggled to achieve higher than a 43% accuracy rate when training on the MCTS agent data. As the output space of the neural network is one of twenty seven different moves (each move corresponding to placing each card on either a hand or discarding), a 43% accuracy is still quite good compared to random moves. The low accuracy is likely due to the neural net finding several moves all as strong contenders, and does not have the ability to distinguish between two somewhat “identical” moves.

5 Results

For each agent, results are taken over 25000 games (not including Fantasyland). The true agent RPH is then calculated by taking:

\[
\text{True RPH} = \frac{\text{Natural RPH} + (1.1 \cdot \text{FL \%}) \cdot 12.2}{1 + (1.1 \cdot \text{FL \%})}
\]

5.1 Random Agent

Over 25000 games, the random agent achieved:

<table>
<thead>
<tr>
<th></th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average Natural RPH</td>
<td>0.04 RPH</td>
</tr>
<tr>
<td>Fantasyland Probability</td>
<td>0.00</td>
</tr>
<tr>
<td>Foul Probability</td>
<td>0.69</td>
</tr>
<tr>
<td>Average True RPH</td>
<td>0.04 RPH</td>
</tr>
</tbody>
</table>

5.2 MCTS Determinization Agent

Over 25000 games, the MCTS determination agent achieved:

<table>
<thead>
<tr>
<th></th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average Natural RPH</td>
<td>3.44 RPH</td>
</tr>
<tr>
<td>Fantasyland Probability</td>
<td>0.10</td>
</tr>
<tr>
<td>Foul Probability</td>
<td>0.19</td>
</tr>
<tr>
<td>Average True RPH</td>
<td>4.35 RPH</td>
</tr>
</tbody>
</table>

5.3 MCTS with Randomization Agent

Over 25000 games, the MCTS with randomization agent achieved:

<table>
<thead>
<tr>
<th></th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average Natural RPH</td>
<td>3.62 RPH</td>
</tr>
<tr>
<td>Fantasyland Probability</td>
<td>0.14</td>
</tr>
<tr>
<td>Foul Probability</td>
<td>0.25</td>
</tr>
<tr>
<td>Average True RPH</td>
<td>4.76 RPH</td>
</tr>
</tbody>
</table>
5.4 Neural Network Agent

Over 25000 games, the neural net agent achieved:

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average Natural RPH</td>
<td>3.07 RPH</td>
</tr>
<tr>
<td>Fantasyland Probability</td>
<td>0.13</td>
</tr>
<tr>
<td>Foul Probability</td>
<td>0.41</td>
</tr>
<tr>
<td>Average True RPH</td>
<td>4.24 RPH</td>
</tr>
</tbody>
</table>

5.5 Observations

It is seen that the MCTS with randomization agent achieved the best score of the AI agents both in natural RPH as well as true RPH. This agent played more aggressively than the MCTS determinization agent, as natural RPH, Fantasyland probability, and foul probability all increased. This is likely due to the agent finding low probability but high reward events worth going for, achieving higher payoffs than its deterministic counterpart.

The neural net agent is an interesting mix of the two, as it played quite aggressively for Fantasyland (much as the MCTS with randomization agent did). However, this agent fouled more than twice as often as the MCTS determinization agent. This is likely due to fouls not being encoded in the gamestates that the neural network was trained on as there are no moves left to be made when a foul occurs. Another reason this might be the case is that the agent may not have learned that hands should not be stronger than the hands below them, but simply that it should make each hand as strong as possible.

Another major factor in determining the maximum score of each agent is how the initial five cards were set. These placements dictate how the rest of the hand will play out, and minor changes to the starting placement can drastically alter the expected value of the round. As the heuristic for placing the starting five cards aggressively places cards in order to achieve Fantasyland, it is likely that there are better initial placements that may allow the agents to achieve higher scores.

6 Further Work

With the goal of developing AI agents that are highly competitive with even expert human players, there are several steps that can be taken to further improve this project. The first major improvement would be to have an agent solve for the starting five cards using a probabilistic approach as opposed to a strict heuristic. This would allow for a more accurate placement of cards, which in turn would allow for a higher potential RPH for each round. Secondly, training the neural network on data encoded so that it is able to achieve a high accuracy rate would allow for both a higher RPH from the neural network. Several modifications to the attempted reinforcement learning agent could also be made so that it is able to further improve the network beyond what it is able to achieve from supervised learning alone.
Finally, an analysis of which moves a high-level AI agent chooses to make would allow for further improvement of human play. It would be interesting to see what sorts of features the agent sees as highly valuable, and which features the agent may not value as much as a human player (such as a starting hand with three cards to a flush).

6.1 Reinforcement Learning Agent

Q-learning is a popular method of having AI agents learn to play games or perform tasks independently of human supervision. Q-learning is when an agent learns a value $Q$ for each possible state-action pairs in the game. These values are initially randomized, but can be updated by setting $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha (R_{t+1} + \gamma \max_a\{Q(s_{t+1}, a') - Q(s_t, a_t)\})$, where $Q(s_t, a_t)$ is the quality score for the state-action pair $(s_t, a_t)$, $\alpha$ is the learning rate for new information, $R_{t+1}$ is the reward for taking action $a_t$, and $\gamma$ is the discount rate for taking an off-policy action from the target action.

As the state-action space of Pineapple is too large to feasibly compute, a neural network can be used to approximate quality score for different state-action pairs. To do so, a Q-network can be used to find quality scores $Q(s, a; \theta_i)$, and a separate target-network can be used to find $T := R + \gamma \max_a\{Q(s', a'; \theta_{i-1})\}$ for a future state $s'$. Using typical gradient descent algorithms in neural networks, these values can then be used to minimize the mean squared error between $T$ and $Q(s, a; \theta_i)$.

Unfortunately, when attempting to implement this algorithm for the neural net agent, it was found that the reinforcement learning caused the neural network to regress back to an essentially random state. Upon examination for why this might be the case, it was realized that the neural network agent predicts actions to be made as opposed to the actual value of the current game board. As a result, the neural network was not actually outputting a quality score for $Q(s_t, a_t)$ for a given state-action pair for a given game board.

Deep-Q learning sees the neural network updated based on a reward (e.g. the number of royalty points) and not on a literal move choice. Additional work to the neural network may be done to either incorporate the reward of each board into the gamestate encoding itself, or to train a separate neural network to output the value of a board (and then select the action which maximizes this value) instead of outputting an action directly. The latter would likely be most promising in executing reinforcement learning using a Q-network/target-network pair as described above.
References


