# Mastering Open-face Chinese Poker by Self-play Reinforcement Learning

Andrew Tan, Jarry Xiao

September 2018

**Abstract**

The *AlphaZero* algorithm has proven to be extremely powerful in learning deterministic games with no domain knowledge minus the game rules. In this paper, we take a preliminary step into seeing if it can produce similar results with a more stochastic game. We found that a vanilla DQN is relatively ineffective when training an agent to play Open Face Chinese Poker through self-play reinforcement learning. However, we noticed that there were marked improvements when introducing a more greedy approach to action selection during the early stages of the game, and we developed a more robust model to fit the stochastic nature of the environment.

## 1 Problem Statement

### 1.1 Rules

Open-face Chinese Poker (OFCP) is a stochastic, perfect-information zero-sum game played between 2-4 players. For the purposes of our RL agent and this project, we aim to solve the two-player variation, which draws on more variance. The rules of OFCP are relatively simple. There are many variants of the game–a version known as "Pineapple" is particularly popular among the poker community. The goal of the game is to earn more points (also known as units) than your opponent by winning more hands (rows) and/or by collecting royalties on premium hands without fouling. During each turn, players will draw a card and place it in one of three locations on the board: front, middle, or back. The only exception is the first turn, where each player draws and places 5 cards at a time. Players will take turns drawing and placing a card until each have a board of 13 cards: 3 cards in the front, 5 cards in the middle, and 5 cards in the back. For a board to be valid, the back hand must be stronger than or equal to the middle and the middle hand must be stronger or equal to the front. Otherwise, the hand is not legal and is considered fouled. An invalid board means the player must forfeit each hand and is not eligible for royalties. In the event both players foul, neither will earn any points.

In our project, we decided to build a reinforcement learning agent for the following variant:

1. Each player takes turns placing cards on their front (3 cards), middle (5 cards), and back (5 cards) "streets."

2. When the game ends, for a player to have a valid board, the back street needs contain a poker hand that beats the hand in the middle street, and the middle street needs to beat the hand in the front street.

3. Points are computed by pairwise comparisons of streets. On the first turn, each player is dealt 5 cards and may place them in any valid configuration.

4. For subsequent turns, the players are dealt a single card and can place it and any street that is not fully occupied.

5. All actions are seen by both players.

6. If a player beats the opponent in all 3 streets, that player is granted 3 additional points for sweeping.

At the game's conclusion, if the winner scored $N$ points, the loser will have scored $-N$ points. Thus, the game is zero-sum. Compared to games such as Chess or Shogi, Open-face Chinese poker has an extremely small action space with a player having only 3 possible moves at any given point. However, the state space is enormous by comparison as a result of the stochastic nature of the problem. Each player has a board of 13 cards, and on each turn one of the players is dealt a new card to play on the board. Thus, the state space is upper bounded by 53 (52 cards plus a symbol to denote an empty space) raised to the power of 27 (26 slots plus a slot for the current action card). The setup of this problem as a 2 player zero-sum game makes it seem like a prime candidate for applying deep reinforcement learning.

## 1.2 Scoring

Unlike other variations of poker, Open-face Chinese poker uses a scoring system made up of points. The two-player version of OFCP is a zero sum game, so player 2's score is the negation of player 1's score. There are three ways to score points:

1. Winning each hand (street)

2. Scoop

3. Royalties

Players receives 1 point for each hand they win, and 3 bonus points if they win all three hands, known as a scoop. Likewise, players lose 1 point for each hand they lose and lose 3 bonus points if they lose all three hands, known as being scooped. If player 2's board is fouled, as long as player 1 has a valid board, player 1 will earn +6 points and player 2 will earn -6 points (3 points for winning each of the 3 hands and 3 bonus points for collecting the scoop). If both players foul, neither will earn points. Theses points are added to any royalties earned to determine final score.

### 1.2.1 Royalties

Royalties are extra points that may be awarded to players with particularly strong hands. Hands that qualify for additional royalties in Open-face Chinese poker and their corresponding scores are shown below:

| Front | Units | Middle | Units | Back | Units |
|---|---|---|---|---|---|
| 66 | 1 | Three of a kind | 2 | Straight | 2 |
| 77 | 2 | Straight | 4 | Flush | 4 |
| 88 | 3 | Flush | 8 | Full house | 6 |
| 99 | 4 | Full house | 12 | Four of a kind | 10 |
| TT | 5 | Four of a kind | 20 | Straight flush | 15 |
| JJ | 6 | Straight flush | 30 | Royal flush | 25 |
| QQ | 7 | Royal flush | 50 | | |
| KK | 8 | | | | |
| AA | 9 | | | | |
| 222 | 10 | | | | |
| 333 | 11 | | | | |
| 444 | 12 | | | | |
| 555 | 13 | | | | |
| 666 | 14 | | | | |
| 777 | 15 | | | | |
| 888 | 16 | | | | |
| 999 | 17 | | | | |
| TTT | 18 | | | | |
| JJJ | 19 | | | | |
| QQQ | 20 | | | | |
| KKK | 21 | | | | |
| AAA | 22 | | | | |

Figure 1: These are the royalty points scored for exceptionally strong poker hands on each street (Source: Wikipedia)

As mentioned in the rules section, the final score is determined by a pairwise comparison between the streets of both players. Let $f_i$, $m_i$, and $b_i$, where $i \in \{1, 2\}$ define the royalty points scored by each player in the front, middle, and back streets respectively. Note that players only earn royalties if they produce a valid board. Even if a player has a royal flush on the back hand, if their front hand is stronger than their middle hand (resulting in a foul), the player will not earn any royalties. Additionally, let $p_i$ be the score of each player and $n_i$ be the number of rows where player $i$ has a better hand. Let $scoop_i$ be the number of points awarded by the scoop bonus: 3 if $p_i$ wins all 3 hands, -3 if $p_i$ loses all 3 hands, and 0 otherwise. The points of the game are determined as follows.

$$p_1 = (n_1 - n_2) + scoop_1 + (f_1 - f_2) + (m_1 - m_2) + (b_1 - b_2)$$
$$p_2 = -p_1$$

The opposite holds true when Player 2 wins. In the case where both players bust, meaning that neither player generates a valid board, they are both awarded zero points for that particular round. This game can theoretically continue infinitely, and if two agents of equal strength played against each other, the score should theoretically be around 0.

# 2 Background

## 2.1 Theoretical Background

### 2.1.1 Markov Decision Process

A Markov Decision Process is a model that contains a set of possible world states ($S$), a set of possible actions ($a$), a probability distribution that a certain action at time t will lead to a certain state at time t + 1 ($P_a(s, s')$), a discount factor ($\gamma$) and a real valued reward function ($R_a(s, s')$). The problem that the Markov Decision Process attempts to solve is to find a policy that specifies what actions an agent chooses in a given state. As stated, most games, including OFCP, can be modeled as a Markov Decision Process.

The algorithm to solve this problem is defined recursively as follows:

$$\pi(s) := arg\max_a \sum_{s'} P_a(s,s')(R_a(s,s') + \gamma V(s'))$$
$$V(s) := \sum_{s'} P_{\pi(s)}(s,s')(R_{\pi(s)}(s,s') + \gamma V(s'))$$

Markov decision processes have the Markov property i.e. the probabilities of the next state are independent of the states before the current state given the current state. Because the value and policy calculations involve extremely deep recursive algorithms (each policy value is calculated by iterating over the entire action space and state space until the reward is reached) heuristics are commonly used. An example of this would be Monte-Carlo tree search which uses random sampling of the search space and weighted backpropagation to calculate a heuristic for a given path.This is also one of the reasons deep neural networks with sampling is used to estimate the value function.

### 2.1.2 Value Iteration

If the entire state space fits in memory, we can iteratively calculate the maximum expected total reward directly by taking the maximum reward action:

$$V(s_i) = \max_{a_i} R(s_i|a_i) + \gamma \sum_{s_{i+1}} V(s_{i+1}) p(s_{i+1}|s_i, a_i)$$

This is known as the Bellman update. Essentially all we are saying is that the value for a given state is the action that maximizes reward ($R(s_i|a_i)$) plus expected reward for the next state ($\gamma \sum_{s_{i+1}} V(s_{i+1}) p(s_{i+1}|s_i, a_i)$) where $\gamma$ is the discount factor. Note that these Bellman updates can be computed with dynamic programming to take $O(|S||A|)$ steps where $S$ is the set of states and $A$ is the set of actions.

### 2.1.3 Q-Learning

Q-learning is a reinforcement learning technique that can be used to tackle a Markov Decision Process when the probabilities or rewards are unknown. In the case of OHCP, the probability distribution is unknown. The main idea is to explore state-action pairs to estimate the reward value of applying a given action to a given state (Q(s,a)). To perform Q-learning, an additional function is defined:

$$Q(s,a) = \sum_{s'} P_a(s,s')(R_a(s,s') + \gamma V(s'))$$

Before learning begins, Q is initialized at a fixed value for all state action pairs. The way it is updated is a simple value iteration:

$$Q(s_t, a_t) \leftarrow (1-\alpha) Q(s_t, a_t) + \alpha * (r_t + \gamma \max_a Q(s_{t+1}, a))$$

Our final policy is defined:

$$\pi(a_t|s_t) = \begin{cases} 1, & \text{if } a_t = \arg\max_{a_t} Q_\phi(s_t, a_t). \\ 0, & \text{otherwise.} \end{cases}$$

Intuitively, what is happening here, is the agent is selecting an action (sampled from some probability distribution) and observing a reward at which point Q is updated using a weighted average of the old Q value and the reward plus a discounted maximum over the next action that the agent can take. The discount factor on the $\max_a Q(s_{t+1}, a))$ can be interpreted as the probability that the action will actually give you the reward you expect ($Q(s_{t+1}, a)$).

Q-learning can be shown to eventually find an optimal policy, however Q-learning is known to be unstable due to correlations present in the observations, small updates to Q significantly changing policy values and correlations between the action-values ($Q$) and the target values ($r + \gamma \max_a Q(s', a')$) since sequential states are strongly correlated.

#### 2.1.4 Deep Q-Networks

One way of approximating the Q is by representing it with a neural network. This essentially defines a parametric function to attempt to learn about the environment. The classic deep Q-learning algorithm are the following steps:

**Algorithm 1** Deep Q-learning Algorithm

1. Take action $a_i$ and observe $(s_i, a_i, s'_i, r_i)$, add it to the set $B$
2. Sample from $B$ uniformly at random
3. Compute $y_i = r_j + \gamma \max_{a'_j} Q_{\phi'}(s'_j, a'_j)$ using *target* network $Q_{\phi'}$
4. $\phi \leftarrow \phi - \alpha \sum_j \frac{dQ_\phi}{d\phi}(s_j, a_j)(Q_\phi(s_j, a_j) - y_j)$
5. Update $\phi'$ and copy $\phi$ every N steps

## 2.2 Importance

*AlphaGo Zero* has been able to achieve superhuman performance in the game of Go with no expert knowledge and *Alpha Zero* has been able to generalize these results for other deterministic games such as Chess and Shogi [Sil17b][Sil17a]. The superiority of these models are clear: they require no human data, they use only a single neural network rather than separate policy and value networks and empirically, they are able to train much faster and better than traditional alpha-beta search trees with clever heuristics and domain-specific adaptations such as *Stockfish* or *DeepBlue* [Sil17a].

The network behind *Alpha Zero* is a deep convolutional neural networks trained with reinforcement learning from games of self-play. In *Alpha Zero*, Monte Carlo tree search, a heuristic search algorithm for Markov Decision Processes, is used in conjuncture with the neural network values to calculate a probability distribution over the action space. An action is than sampled proportional to the visit count of each move. Once a given model performs better than another for some number of iterations, those parameters are saved and used in the next iteration of self play.

Stochastic games, such as Open-face Chinese poker, have some differences and similarities with the deterministic games explored by *Alpha Zero*. One difference is that the reward is not just win or lose as the agent can go for stronger or weaker winning states. This does not change the algorithm much as it only changes the reward value given at the end of the game. Another difference is that even if the agent plays completely optimally, there is still a chance the agent could lose due to luck.

A similarity, specific to Open-face Chinese poker, is that the game involves information symmetry i.e. all players have identical information about the current state of the game. This makes the game easier to tackle than a game such as poker which has imperfect information. The reason being is that a player only knows their own hand and the correct action depends on a probability distribution over what the opponent has given their actions so far in the game. The opponent then observes this first players actions creating a recursive loop in each players beliefs about the other player. This leads to the necessity of techniques such as Counterfactual Regret Minimization (CRM) which is a recursive reasoning technique which uses a concept of "regret" (how much you lose for not taking a different action) to minimizes a nuanced formulation of its value.

Successes in applying the *Alpha Zero* technique to Open-face Chinese Poker is a first step in applying the technique to other stochastic games as Open-face Chinese Poker contains some but not all of the difficulties associated with stochastic games. As current artificial intelligence techniques for poker still require domain knowledge, a successful *Alpha Zero* type implementation would be a large step forward in mastering these games.

# 3 Tools

We utilize several open source tools throughout our project including Keras, TensorFlow, and deuces.

We used Keras with a TensorFlow backend to build and train our RL agent and deep neural network. For our network, we used a sequential model with dense layers, batch normalization, dropout, and ReLU activations. For training, we used mean squared error loss and Adam optimizer.

Deuces [Dre16] is a Python library for creating, visualizing, and evaluating poker hands. Deuces handles 5, 6, and 7 card poker hand lookups with blazing speed and was used to rank hands when determining scores and royalties. We used the poker infrastructure and game-play provided by Deuces to develop our own game logic for Open-face Chinese poker. The library also supports methods that display cards in a user-friendly manner and allows for simple game-play. The following is an example of what an Open-face Chinese poker game board might look like using visualizations from Deuces:

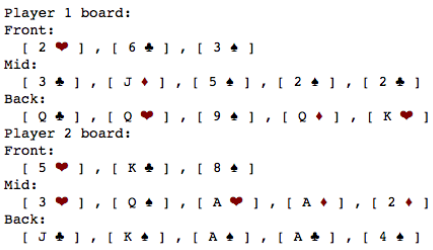



Figure 2: User-friendly display for easy game-play and board visualization

The repository for Deuces can be found here: `https://github.com/worldveil/deuces`

Since our RL agent utilizes self-play to train, there was no need for external data sources. Data was collected in the form of rollouts and games played against itself.

# 4 Naive Model

The problem is modeled as as a Markov Decision Process with the states being the cards on both sides of the board along with the current card in the players hand. The action space is small being just 1 of the three positions i.e. front, middle or back street.

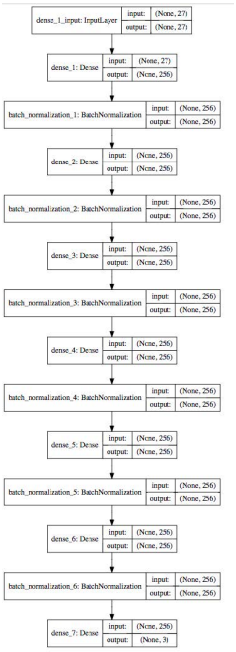



Figure 3: This is our Deep Q-network

We use Deep Q-Learning to determine the optimal policy at a given state. Our network is a simple, 5 layer network with batch-normalization between hidden layers. Our network is trained using rollouts from games between our model and a random agent. We define the rollouts as follows:

$$[(s_1, s_2, a_1, r_1), ..., (s_9, s_{10}, a_9, r_9)]$$

These values allow us to train our network against a target network using the mean squared error loss. When the loss is minimized, this indicated that our model has converged on an optimal policy (in this case, against a random agent). Then we planned to iterate this process by replacing the opponent with the newly trained policy and refining our model. First, we need to define an algorithm to generate these rollouts. Define $T(s, a)$ to the be the state that results from taking action $a$ at state $s$. Our naive approach is as follows:

**Algorithm 2** Generate Rollouts

**Result:** Computes a single rollout for training.

$cards \leftarrow$ First 5 cards

$S \leftarrow$ All states with $cards[:4]$

$A \leftarrow$ {front, mid, back}

$s_1 \leftarrow arg\max_s\{Q(s, a)|s \in S, a \in A\}$

$a_1 \leftarrow arg\max_a\{Q(s, a)|s \in S, a \in A\}$

**for** $i = 2, ..., 8$ **do**

  $s_i \leftarrow T(T(s_{i-1}, a_{i-1}), a_{i-1}^{opponent})$

  $a_i \leftarrow arg\max_a Q(s_i, a)$

**end**

$s_9 \leftarrow T(T(s_8, a_8), a_8^{opponent})$ $a_9 \leftarrow$ last remaining action

$s_{10} \leftarrow$ final game board

Additionally, we hold off the reward until the final state, so the reward is defined as follows

$$r_i = \begin{cases} 0 & i < 9 \\ p_1 & i = 9 \end{cases}$$

where $p_1$ represents the score of Player 1 (for this model, we assumed that our agent was always Player 1 for ease of implementation. Once we have all this information, we can process the outputs to generate the rollouts. When training the model, we used an $\epsilon$-greedy strategy to achieve both exploration and exploitation. We first give our model an *observation* stage, by enforcing completely random decisions for a period of time. Then, we allow the value of $\epsilon$ to decrease over a series of time steps until it reaches a final value of 0.05 (which was determined relatively arbitrarily). The idea was to expose the model to more rollouts so it could have data on possible end states for the game. As in traditional DQN, we used a mean squared error loss function over $N$ rollouts between the model and a target network, which was a previous iteration of the current model's weights. This target network is updated every 10 iterations (equivalent to 10,000 full games) in order to move closer to convergence. Let $Q'$ represent the target network and $\phi_t$ be the parameters of our DQN at iteration $t$. Our loss function and update step are defined as follows.

$$y_t^{(i)} = R(s_t^{(i)}, a_t^{(i)}) + \max_a Q'(s_{t+1}^{(i)}, a)$$

$$\mathcal{L}(\phi_t) = \frac{1}{9N} \sum_{i=1}^{N} \sum_{t=1}^{9} \|Q_{\phi_t}(s_t^{(i)}, a_t^{(i)}) - y_t^{(i)}\|^2$$

$$\phi_{t+1} = \phi_t - \alpha \nabla_\phi (\mathcal{L}(\phi_t))$$

This process repeats until the loss converges, which in theory should produce an optimal policy to play OFCP.

# 5 Modifications

## 5.1 Greedy Starting Hand Policy

In an effort to accelerate learning during the early and most ambiguous stages of the game, we propose a semi-greedy method for discovering favorable starting hand positions. We identified ten strong five-card starting hands and, when observed, enforced a policy for choosing their positions (streets). These advantageous starting hands are:

1. Four of a kind
2. Full house (three of a kind and pair)
3. Flush (5 cards all of the same suit)
4. Straight (5 cards of sequential rank)
5. 4 cards of the same suit
6. Two pair (2 cards of the same rank, 2 cards of another rank)
7. Three of a kind
8. 4 cards of sequential rank
9. 3 cards of the same suit

10. 3 cards of sequential rank

We want the bottom row to be the strongest and increase our chance of earning royalties, so cards that make up one of the ten advantageous hands will be placed in the bottom row. The remaining cards will be distributed between the top and middle rows, depending on their rank. Cards with rank 8 and above will be placed in the middle row, and cards with rank below 8 will be placed in the top row. This policy enforces a starting hand strategy for strong cards and will ideally speed up training.

### 5.2 Final Model

In addition to this greedy start strategy, there appeared to be some flaws in the way we initially designed our model. The Q-Network is supposed to evaluate the expected reward given an optimal policy, but the rollout we trained it on did not fully reflect this idea. As previously mentioned, we used a mean squared error loss between the target prediction and the current network prediction, with the idea that eventually the weights would converge to a parametric approximation of the optimal policy. Instead, we observed the loss plateauing, and the network's decision did not seem to improve.

For our rollouts, we selected our actions based on only the next state observed in the game that was played against the random agent. However, because OFCP has a lot of variance, the sample size (or search space) of a single outcome branching from a specific game state provides little information about an agent's decision making. Unlike in deterministic games, in stochastic games, sometime decisions with high expected return might still lose in particular instances due to variance. The law of large numbers tells us that the sample mean from a population will approach the population mean as the sample size grows large. In this case, the population in question is the Q-value of taking an action. If we increase the sample size of the possible future states at time $t+1$ given our action at time $t$ in the loss function, we should observe a more accurate representation of the quality of our agent's decision, which should lead to faster convergence. Let $M$ be the number of simulations we run from any state from the rollout. Our updated loss function is the following:

$$y_t^{(i)} = R(s_t^{(i)}, a_t^{(i)}) + \frac{1}{M} \max_a \sum_{j=1}^{M} Q'(s_{j_{t+1}}^{(i)}, a)$$

$$\mathcal{L}(\phi_t) = \frac{1}{9N} \sum_{i=1}^{N} \sum_{t=1}^{9} \|Q_{\phi_t}(s_t^{(i)}, a_t^{(i)}) - y_t^{(i)}\|^2$$

Because the action space of OFCP is so small, we can directly run this simulation without doing a more informed sampling approach like Monte Carlo Tree Search. In the case of a game with a larger action space (e.g. Go or Chess) this greedy simulation strategy would probably be a lot more effective in that fewer samples would be needed. The larger we set $M$ to be, the quicker our DQN should converge to an optimal policy. This makes more sense in the DQN formulation because a single state in OFCP can lead to many different state of the game (which are out of control of either player). This randomness needs to be accounted for when approximating the Q function.

## 6 Challenges

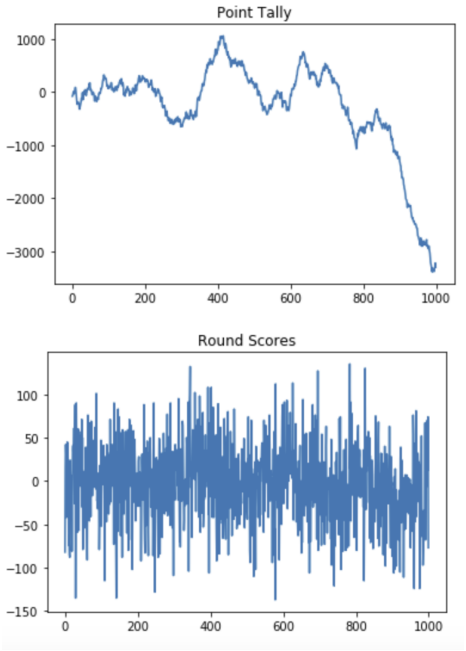
During the early stages of training, we noticed that our agent was learning but the loss and gradients were exploding. We realized this was due to the way we computed our loss. Instead of calculating loss by the mean squared error of the entire output of our DQN (all three action values), we should only be measuring loss based on the actual action taken. Additionally, our DQN took very long to train and converge, largely due to our computational constraints. We maneuvered around this by

introducing a greedy action selection during the early stages of the game. This sped up training and early game strategy for our RL agent. Another recommendation was to implement Monte Carlo tree search (MCTS). Our model was relatively simple, and introducing more nuance in exploration and a deeper network may have been helpful.

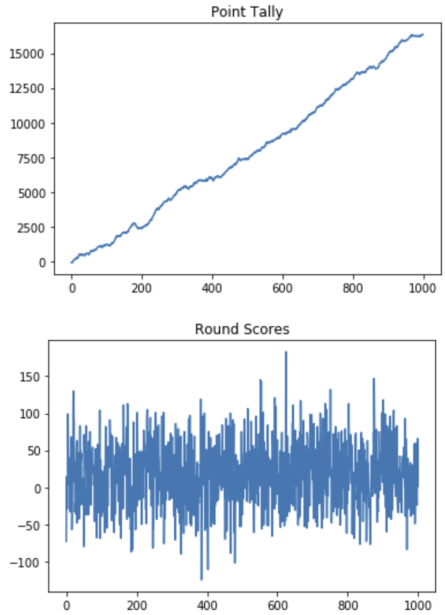
# 7 Results

Our naive model had issues with convergence as mentioned in the section above. As a result, our model exhibited a lot of variance in its performance against a random agent. As mentioned, we initially dealt with issues regarding the model's loss, but this was mitigated when we tweaked the model only backpropagate on the action taken by the RL agent. However, the issue of varying performance remained. In fact, it seemed as if the performance of the action was heavily influenced by the weights selected in the model's initialization. When training, we kept track of a running sequence of scores over the course of each iteration (100 games) as well as a total cumulative score between the RL agent and the random agent. Below are some of the results from a particularly well performing agent.

Later, we decided to utilize a greedy strategy for training as it would lead to a faster sequence of rollouts as well as potentially prove the model's performance against a random agent. These hypotheses were both verified as we observed a speedup in training time, and the agent's performance became much more consistent. Because the starting strategy "guided" our DQN towards a better policy and also increased the chances of the agent building strong boards by chance, this improvement was not unexpected. However, the linear growth of the cumulative point tally indicates that learning has not truly occurred. We unfortunately did not have time to implement the final model listed in section 6, but we believe that would have been a better approach of training that may in fact lead to measurable improvement.




(a) Training data for naive model




(b) Training data for updated model

Figure 4: Differences between models

The training loss for both of these models plateaued around 0.4 for the majority of the training sequence. However, because the results are not linear, we believe that there is still a lot of room for our model to improve.

# 8 Conclusion

This project was a challenging endeavor in applying our knowledge of deep reinforcement learning to a stochastic game. Due to constraints on our model and computational resources, we were unable to achieve the level of game-play as AlphaGoZero. As next steps, we intend on implementing methods for speeding up training and developing a better heuristic for exploration. The repository with our code can be found here: `https://github.com/andrewztan/deep-rl-ofc-poker`

# References


[BD] Andrew Barto and Michael Duff. Monte carlo matrix inversion and reinforcement learning. In *Machine Learning*, Amherst, MA, USA. University of Massachusetts.

[Dre16] Will Drevo. Deuces: A pure python poker hand evaluation library, 09 2016.

[HS16] Johannes Heinrich and David Silver. Deep reinforcement learning from self-play in imperfect-information games. London, UK, 2016. University College.

[Sil17a] Julian; Simonyan Karen Silver, David; Schrittwieser. Mastering the game of go without human knowledge. DeepMind, 2017.

[Sil17b] Thomas; Schrittwieser Julian; Antonoglou Ioannis; Lai Matthew; Guez Arthur; Lanctot Marc; Sifre Laurent; Kumaran Dharshan; Graepel Thore; Lillicrap Timothy; Simonyan Karen; Hassabis Demis Silver, David; Hubert. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. London, UK, 2017. DeepMind.

[WD92] Christopher Watkins and Peter Dayan. Q-learning. In *Machine Learning*, Boston, MA, USA, 1992. Kluwer Academic Publishers.