Extended Abstract

Motivation We wanted to examine the use of reinforcement learning methods in complex game play environments. While No-Limit Texas Hold'em poker has been heavily explored, the variant of Open-Face Chinese Poker (OFCP) has minimal prior research. Through this paper, we aim to explore various methods of RL methods that can applied to optimal gameplay for OFCP. We seek to understand how the sparse reward system and complex hand dynamic make OFCP more challenging than other poker variants, determining whether Q-Learning, PPO, and MCTS can be applied to create successful agents.

Methods We implemented several RL algorithms on OFCP in a self-play environment, requiring no external datasets and compared its effectiveness against rule-based and learning-based opponents. Many of these implementations haven't been explored for OFCP previously. Q-Learning is used as a baseline method as it was previously used by Tan and Xiao (2018) for OFCP. We also implement Deep Q-Learning and the Double DQN and Dueling DQN variants. Inspired by Tan and Xiao (2018), the DQN uses a neural network to approximate Q-values, allowing scalability for large state representations like OFCP. Double DQN mitigates Q-value overestimation in DQN through decoupling action selection and evaluation. Dueling DQN creates a state-value stream and a action-advantage stream, allowing the agent to create more ideal Q-values compared to DQN. Our second category of methods is PPO. PPO has a stable policy gradient learning and stochastic decision-making, which can better handle imperfect information and uncertainty in game play. We implement clipped surrogate optimization and entropy regularization, and to improve tie-breaking, Generalized Advantage Estimation (GAE) is implemented as an extension. Finally, the third class of methods is MCTS. MCTS explores potential card placements and simulates future rollouts through early termination and pruning. We implement the cross-entropy method, RAVE, and Counterfactual Regret Minimisation to address less-informed early moves and long-term planning.

Implementation We created a custom two-player OFCP environment aimed to compete against our method bots. The reward function follows after OFCP rules, incorporating winning a hand, fouling, scooping, and royalties. The environment includes deck management (shuffling and dealing cards) and game simulation (placing cards, checking hand validity, and running rounds). All of the agents are trained through self-play and evaluated through method win rate, bot win rate, average points per game, and training efficiency.

Results The results from our experiments demonstrate that **MCTS with optimizations of CEM, RAVE, and CFR** achieved the highest model win rate of 89%, limiting the bot win rate to 3%, and averaging 11.2 points per game over 100 evaluated games. This far outperformed the second model of PPO with the GAE optimization which has a model win rate of 41% and 5.02 points per game, limiting the bot to 0% win rate. Among the Q-learning methods, Double DQN optimized the best, achieving a model win rate of 35% and 3.50 points per game, though significantly less effective than the PPO and MCTS. Notably, while MCTS has strong performance for the method implementation, it comes at a computational cost, taking 447 minutes for evaluating 100 games which is almost 200x longer than PPO's 20-second evaluation, highlighting the importance of performance and computational efficiency.

Discussion and Conclusion Our methods focus on two-player OFCP and omits variants like Fantasyland/Shoot the Moon and multi-player dynamics that would demand richer reward modeling. While MCTS achieves top performance, its high computational cost limits real-time play. Future work will optimize MCTS efficiency (e.g. via parallel roll-outs or learned policies), explore hybrid neuroevolution–RL to tackle sparse rewards, and systematically tune lightweight methods to improve their decisiveness without prohibitive compute. We showed both of our hypotheses for this project: demonstrating that all three classes of methods outperform a Random bot and that MCTS outperforms the other two classes with a 89% Method Win Rate.

Advancing Multi-Agent Reasoning in Open-Face Chinese Poker

Alice Guo
Department of Computer Science
Stanford University
azguo@stanford.edu

Ramya Iyer
Department of Computer Science
Stanford University
ramya1@stanford.edu

Isabella Lee
Department of Computer Science
Stanford University
leeij@stanford.edu

Abstract

While not explored as often as traditional poker, this paper analyzes various reinforcement learning (RL) methods to solve Open-Face Chinese Poker, a complex, imperfect information with sparse rewards. We implement and compare major approaches like Deep Q-Learning (including Double DQN and Dueling DQN), Proximal Policy Optimization (PPO) with Generalized Advantage Estimation, and Monte Carlo Tree Search (MCTS) with Cross-Entropy Method (CEM), Rapid Action Value Estimation (RAVE), and Counterfactual Regret Minimization (CFR). Across 100 evaluation games per method, the MCTS achieved the highest win rate of 89% and highest average points per game of 11.2, significantly outperforming the other RL methods. However, this method performance trades off with computational efficiency. Our findings highlight MCTS's superiority in OFCP environments, suggesting future works on computational efficiency, faster inference, and optimizing methods for more complex game variants.

1 Introduction

Open-Face Chinese Poker (OFCP) is a multiplayer, imperfect information card game where players place 13 cards into three rows (top, middle, and bottom). Players create poker hands of 3, 5, and 5 cards, respectively, aiming to gather the largest amount of points possible without knowing the order of future cards or the card their opponent is currently placing on their board. OFCP requires multi-agent reasoning under uncertainty, strategy adaption, and opponent modeling, making it a benchmark to test Multi-Agent Reinforcement Learning (MARL) techniques.

Reinforcement learning has been previously used to play and solve poker variants like No-Limit Texas Hold'em (NLTH) through the creation of agents such as *Libratus* and *AlphaHoldem*. However, the game of OFCP remains largely underexplored due to its complicated three-tiered structure, making the state space more complicated than other poker variants. Methods proposed by Tan and Xiao (2018) and Kirklin (2014) have identified additional challenges of sparse rewards and computational expenses, particularly for methods such as MCTS. Seeing this opportunity, our paper and final project explores reinforcement learning and search-based approaches for creating an OFCP agent.

We constructed two hypotheses to test. First, all tested methods in this project (MCTS, Q-Learning, Deep Q-Learning, and PPO) can show significant improvement over a bot placing random cards, displaying an ability to play strategically or learn patterns through self-play without relying on pre-existing data. Secondly, we hypothesize that MCTS will perform better than other RL methods

such as Q-Learning, Deep Q-Learning, and PPO due to MCTS's ability to systematically explore large game possibility spaces and to handle imperfect information by averaging outcomes over many sampled rollouts. As a result, we anticipate that MCTS will emerge as the preeminent method of solving Open-Face Chinese Poker.

2 Related Work

Reinforcement learning has driven major advances in poker AI, particularly in No-Limit Texas Hold'em (NLTH). Landmark systems like Libratus employed abstraction, endgame solving, and counterfactual regret minimization to defeat professional human opponents in heads-up NLTH Brown and Sandholm (2018), and DeepStack further demonstrated continual re-solving with a learned value function and sparse lookahead Moravčík et al. (2017). These approaches excel in two-player zero-sum settings but depend on expensive offline computation and heavily engineered abstractions, making them ill-suited to the sequential, multi-row structure of Open-Face Chinese Poker (OFCP).

Multi-agent reinforcement learning (MARL) extends these ideas to environments with hidden information and strategic opponents. For example, attention-based actor–critic methods learn opponent embeddings to guide policy updates in multiplayer NLTH Shi et al. (2022), and value decomposition networks coordinate decentralized agents through a centralized critic in partially observable games Foerster et al. (2018). While these frameworks offer powerful opponent-modeling capabilities, they typically require explicit communication or centralized training data, whereas our PPO agent learns purely via self-play and shapes its own rewards to elicit strategic behavior.

Within OFCP itself, prior work remains sparse. Early attempts applied Deep Q-Networks to raw OFCP states, only to find that sparse rewards and large action spaces severely hindered convergence Tan and Xiao (2018). Monte Carlo Tree Search has been adapted to Pineapple Poker, a close variant of OFCP, by pruning low-value roll-outs to manage computational cost Luo et al. (2018), and pure Monte Carlo bots like Kachushi use hand-ranking lookups to simulate millions of random games Kirklin (2014). These methods rely on extensive roll-out simulations or handcrafted heuristics to break ties and avoid fouls.

In contrast to prior methods that rely heavily on handcrafted heuristics, extensive game abstractions, or pure tree-search simulations, we explore several end-to-end reinforcement-learning methods, including Q-Learning, Deep Q-Learning, Proximal Policy Optimization, and Monte Carlo Tree Search. Each method learns directly from raw card-placement states with a custom reward function to highlight how modern deep-RL techniques can be applied to OFCP in a complex, multi-row scoring environment.

3 Experimental Setup

We implemented a custom environment to simulate 2-player Open-Face Chinese Poker that aligns with the game mechanics while being computationally feasible to use for reinforcement learning experiments. We implement several interlocking pieces of functionality to imitate a OFCP game. We create custom data structures to imitate the three-tier hand structure, allowing players to place cards in a specific row in the player's hand and check the hand validity. We also use a custom data structure to imitate the deck, beginning with a normal 52-card deck. Our deck allows shuffling and drawing both the starting hands and subsequent cards. Furthermore, we allow simulations of an entire game, from the initial draw to the final scoring, as well as simulating a single round.

We also built a custom reward function based on usual OFCP rules. For example, winning a hand tier yields +1 point per winning tier. Scoops, winning all three hand tiers, yields +3 points. However, if a player were to foul with a invalid hand, they would be penalized with -6 points. We also implement royalties, which yield up to 25 points based on high-value combinations of cards such as flushes and straights. We chose to use the usual rules of Open-Face Chinese Poker as the basis of our reward function as we wanted the agents to be able to understand the rules of OFCP and work towards high-value combinations of cards. By finding concrete rewards for completing the objectives of the real game, our bots would be able to learn optimal card placements and strategies to achieve high-scoring hands. However, a modification we made to made to the base model was increasing the positive benefit of completing a non-fouled hand. We also highly penalized a fouled hand regardless of whether the opponent fouling or not, discouraging fouls altogether. This ensured that our methods

would not settle for a fouling hand if their opponent fouled - the agents would instead seek valid hands over fouling ones.

We did not use an external dataset. Instead, all models were trained using self-play in this environment. Our baseline was a reimplementation of the existing literature on Open-Face Chinese Poker. Specifically, previous literature attempted to use Q-Learning and Deep Q-Learning to solve OFCS. These two methods served as a baseline for this project against which we compare our other methods in the Discussion Section. Another baseline was playing against the Random Bot, which chooses random positions in which to place the cards rather than strategically selecting a hand. This baseline serves as one of our evaluation metrics.

To quantitatively evaluate the performance of our agents, we used three metrics. First, we compared method and Random Bot win rates. In a set of n games, the method win rate was calculated as the number of games the method won divided by n. Similarly, the bot win rate was calculated as the number of games the bot won (over the same set of n games) divided by n. Second, we compared points per game. In some cases, This metric represents the average points per game scored by the agent. Third, we compare training and computation efficiency by examining the time and resources used to train the agents.

4 Methods

The agents train on self-play, collecting experiences through learning rather than data. The Q-Learning, Deep Q-Learning, Doubling DQN, Dueling DQN, Proximal Policy Optimization (PPO), Monte Carlo Tree Search (MCTS), and their relevent extensions are described in detail in the following sections.

4.1 Q-Learning

We implemented Q-Learning as an off-policy, tabular RL algorithm, learning the state-action value function $Q(s, a)$ through interacting with the environment and updating the Q-table through the Bellman equation. The Q-values learned are selected during evaluation to determine the accuracy. The agent encodes states as a binary vector with the player's current hand and the incoming card.

Action Space: During each action step, the agent selects among three placement actions as seen in OFCP (top row, middle row, and bottom row).

Bellman Equation: During training, we use the ε-greedy policy, with $\varepsilon = 0.1$ and during the evaluation the agent acts greedily, choosing the best action. We can express the updates on Q-table as the Bellman Equation below

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right]$$

with learning rate of $\alpha = 0.1$ and discount factor of $\gamma = 0.9$.

Training and Evaluation: We train the model agent on 2000 episodes against a random bot and evaluate the model on 100 episodes.

4.2 Deep Q-Learning (DQN)

The DQN model is an extension of the Q-learning that uses a neural network to estimate $Q(s, a)$. The state representation and action space are given from the Q-learning method. For the state, we encode

- Top Row: Maximum of 3 cards $\times$ 52 = 156
- Middle Row: Maximum of 5 cards $\times$ 52 = 260
- Bottom Row: Maximum of 5 cards $\times$ 52 = 260
- Current card: Next 1 card $\times$ 52 = 52

which represents $156 + 260 + 260 + 52 = 728$-dimensional binary vector.

Network Architecture: The DQN architecture consists of 3 multilayer perceptron with layer dimension $728 \to 128 \to 64 \to 3$ with ReLU activation functions. We update the target network after every 100 episodes to help stabilize training.

Loss Function: The agent utilized an experience replay buffer that has a maximum size of $10,000$ with batch sizes of 64. The Q-Network is updated through the MSE loss function explicitly denoted as

$$L = (r + \gamma \max_{a'} Q_{\text{target}}(s', a') - Q(s, a))^2.$$

The neural network also uses the Adam optimizer with learning rate of $1e-3$.

The training and evaluation hyperparameters of training on 2000 episodes and evaluation on 100 episodes.

4.3 Doubling DQN

We extend upon the DQN by implementing a Doubling DQN that migrates Q-value overestimation. The Doubling DQN involves two networks, one online Q-network and a target network. The target network loss function is computed through

$$L = (r + \gamma * Q_{\text{target}}(s', \arg\max_{a'} Q_{\text{online}}(s', a')) - Q(s, a))^2$$

The target network updates every 20 episodes. We utilize the Adam optimizer with a learning rate of $1e-4$, differing from DQN. We create a replay buffer with size 50000 tuples for batch updates. The agent also follows a linear decay ε-greedy policy going from 1.0 to 0.1 over 1000 steps.

4.4 Dueling DQN

The Dueling DQN improves the standard DQN by creating two streams: the value stream and the advantage stream. We find the Q-values through

$$Q(s, a) = V(s) + \left(A(s, a) - \frac{1}{|A|} \sum_{a'} A(s, a') \right)$$

where the value stream outputs the scalar $V(s)$ and the advantage stream outputs action-specific advantages $A(s, a)$. This allows the agent to focus on the states that have higher attention, indicating which Q-values are most valuable. The architecture uses an Adam optimizer and MSE loss on target Q-values.

This model is trained on 2000 episodes and evaluated on 100 episodes, similar to the other Q-Learning methods above.

4.5 Proximal Policy Optimization (PPO)

We implement Proximal Policy Optimization (PPO) as an on-policy actor–critic algorithm that directly maximizes a clipped surrogate of the policy objective while learning a separate value function. Our overall training pipeline consists of the following components:

Agent architecture Our policy/value network is a small multilayer perceptron (MLP) with one hidden layer of 128 ReLU-activated units and two output heads:

- **Policy head**: outputs a softmax over the three placement actions (*top*, *middle*, *bottom*).
- **Value head**: predicts the scalar state-value $V_\theta(s)$.

State encoding At each decision step t, the environment state s_t is encoded as an 8-dimensional vector

$$s_t = \big(c_{\text{top}}, v_{\text{top}},\ c_{\text{mid}}, v_{\text{mid}},\ c_{\text{bot}}, v_{\text{bot}},\ \text{rank}, \text{suit}\big),$$

where (c_i, v_i) denote the category and numeric tiebreaker value of pile i, and rank, suit index the incoming card.

Reward design We convert the environment's native scoring (fouls, royalties, scoop multipliers) into a dense, stepwise signal:

$$r_t = \begin{cases} -6, & \text{if the placement invalidates the hand (foul),} \\ (P_t - B_t) - (P_{t-1} - B_{t-1}), & \text{if this difference is nonzero,} \\ -0.1, & \text{otherwise.} \end{cases}$$

At game end, we further layer on a small tie penalty and scoop bonus (see Results and Analysis for the exact values and their effect).

Numerical stability and clipping Masking illegal moves can produce zero-sum logits and NaNs. We therefore fall back to a uniform distribution whenever the masked logits sum to zero. To bound policy updates and prevent exploding gradients, we adopt the standard PPO clipped surrogate with $\epsilon = 0.2$:

$$L^{\text{CLIP}}(\theta) = -\mathbb{E}_t\Big[\min\{r_t(\theta)A_t,\ \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\, A_t\}\Big],$$

where $r_t(\theta) = \pi_\theta(a_t \mid s_t) / \pi_{\theta_{\text{old}}}(a_t \mid s_t)$.

Entropy regularization To maintain exploration, we add an entropy bonus:

$$-\beta\, \mathbb{E}_t\big[\mathcal{H}(\pi_\theta(\cdot \mid s_t))\big], \quad \beta = 0.01.$$

Optimization The full loss is optimized by Adam (learning rate 5×10^{-4}):

$$L(\theta) = L^{\text{CLIP}}(\theta) + \tfrac{1}{2}\, \mathbb{E}_t\big[(V_\theta(s_t) - R_t)^2\big] - 0.01\, \mathbb{E}_t[\mathcal{H}(\pi_\theta(\cdot \mid s_t))].$$

Training is run for 2,000 episodes with stochastic self-play evaluation every 50 episodes.

4.6 Monte Carlo Tree Search (MCTS)

MCTS is a best-first, assymmetric search technique for sequential decision-making problems. Given a state s_0 and a finite action set A, MCTS seeks to find and choose the action that maximizes the expected return

$$Q(s_0, a) = \mathbb{E}[reward(s_0, a, future)].$$

To do so, MCTS relies on random roll-outs to estimate the value of child nodes. Each child node stores the visit count N, the total reward W, the mean value N/W, and the set of child nodes C. In our implementation, we run 100 simulations. Each simulation consists of 4 stages:

1. **Selection:** From the root node, the algorithm chooses the best child using the UCT equation,

$$v = \arg\max_{u \in \mathcal{C}(v)} \Big[\, Q(u) + c \sqrt{\frac{\ln N(v)}{N(u)}} \Big].$$

   $c > 0$ represents the exploration parameter. We continue choosing the best child until we reach an unvisited action.
2. **Expansion:** The algorithm creates all child nodes based on previously untried actions a, initializing $N = 0$ and $W = 0$.
3. **Roll-Out:** The algorithm randomly chooses a new node and conducts a random roll-out until we reach the game end state. We calculate the reward from this game end state.
4. **Back-Propagation:** We then traverse the path back to the root node, incrementing the counts of each visited node by 1 and accumulating the reward based on the game end state in the Roll-Out stage.

After running all 100 iterations, we select the best child based on the UCT equation as our next action. This allows the algorithm to strike a balance between exploitation and exploitation without handcrafted evaluation functions.

MCTS is particularly effective for Open-Face Chinese Poker as it naturally handles stochastic and partially observable domains such as the unknown cards still in the deck. However, we anticipated

one notable challenge. Though the exploration constant c is recommended to be $\sqrt{2}$ for rewards within $[0, 1]$ based on previous literature, our reward function outputs rewards beyond this range. As a result, we implemented the Cross-Entropy Method (CEM). CEM learns a probability distribution over placements and lets MCTS sample actions from a narrower distribution, shrinking the effective branching factor and allowing the model to do more with the same playout budget (of 100). Additionally, random rollouts can often lead to high variance of leaf rewards if the range of rewards is high, requiring hundreds or thousands of simulations to find a stable Q. CEM creates higher-quality playouts with lower variance by adapting online, leading to quicker convergence. Importantly, since OFCP's reward function is discontinuous, CEM's requirement of simulation returns makes it applicable as a policy-search method. Thus, we are able to auto-tune MCTS hyperparameters and learn a global placement policy that decreases roll-out variance, leading to quicker convergence.

Additional architecture decisions for MCTS extensions are explained in the Results section.

5 Results and Analysis

5.1 Q-Learning Methods: Q-Learning, DQN, Double DQN, Dueling DQN

We hypothesized that the Q-Learning methods would present as a baseline method for OFCP. We implemented each of the variants of Q-Learning, specifically DQN, Double DQN, and Dueling DQN. We enhance the baseline Q-Learning by using deep neural networks to estimate Q-values more effectively. As expected, we find that the win rate of Q-Learning is lower than the DQN implementation. Additionally, the model win rate for Double DQN and Dueling DQN are higher than the DQN. This is due to the improvements that each of the models had on the overestimation of $Q(s, a)$, decreasing the biases through different optimization. Each of the methods were trained on 2000 episodes and evaluated on 100 episodes. These comparisons are seen in Table 1.

Table 1: Comparison of Q-Learning Methods

Method	**Method WR**	**Bot WR**	**Points/Game**	**Total Time**
Baseline Q-Learning	22%	35%	0.98 pts/game	33s
DQN	31%	33%	2.08 pts/game	5m 7s
Double DQN	35%	23%	3.50 pts/game	4m 16s
Dueling DQN	33%	22%	3.01 pts/game	2m 26s

5.2 Monte Carlo Tree Search (MCTS)

We implemented Monte Carlo Tree Search to test our hypothesis that MCTS would perform better than other RL methods such as Q-Learning, Deep Q-Learning, and PPO due to MCTS's ability to systematically explore large game possibility spaces and to handle imperfect information by averaging outcomes over many sampled rollouts. To do so, we first implemented the base algorithm for MCTS, later expanding upon our initial implementation to address some of the challenges our model faced. Our experiments showed the best performance from MCTS with cross-entropy loss for optimizing action proposals at the root node, Rapid Action Value Estimation (RAVE) to improve initial estimates of the best next move given the large search space and slow warm up, and Counterfactual Regret Minimization to better handle hidden information within OFCP to take temporarily suboptimal moves in order to gain better long-term rewards.

We implemented each extension sequentially. Our best results for each step of the implementation are included in Table 2. All experiments were run over a set of 100 games. In these 100 games, Method Win Rate describes the percentage of games the implemented method won. The Bot Win Rate describes the percentage of games the Random Bot won.

Tuning exploration parameter c with the Cross-Entropy Method yielded a best result for $c = 10.721$. However, even after implementing the baseline MCTS and the Cross-Entropy Method, we saw in testing that each child node had few visits and there was still high variance of random playouts. Additionally, our agent played rather conservatively to avoid fouling, missing out on larger bonuses for

Table 2: Comparison of All MCTS Agents

Method	Hyperparams	Method WR	Bot WR	Points/Game	Total Time
Baseline MCTS	c=5	68%	19%	8.2 pts/game	372m 12s
+ CEM	c=10.721	71%	15%	9.6 pts/game	334m 40s
+ CEM, RAVE	c=10.721, k=50	84%	6%	9.8 pts/game	356m 37s
+ CEM, RAVE, CFR	c=10.721, k=75	89%	3%	11.2 pts/game	447m 03s

taking risks. As a result, we implemented two additional extensions: Rapid Action Value Estimation (RAVE) and Counterfactual Regret Minimisation (CFR).

RAVE stores an action-independent estimate of how good each action is whenever it is played anywhere later in the same simulation, not only when it is chosen to expand that particular child, for each state. Then, we can blend this All-Moves-As-First statistic with ordinary UCT. The All-Moves-As-First (AMAF) estimate has lower variance than the normal MCTS Q, allowing us to blend the two equations into a new equation:

$$\beta(N) = \frac{k}{3N + k},$$

where k is a tunable parameter. A higher k keeps β large longer. Since our branching factor is relatively small, with 3 possible actions, RAVE particularly benefits the earliest parts of the search tree while UCT dominates automatically beyond a low depth. After tuning k down to 50, AMAF influence faded once each child had about 200 visits, allowing long-term convergence to be true to UCT values. Running experiments with different values of k with the RAVE architecture yielded the results in Table 3. The exploration hyperparameter c remained at the best value $c = 10.721$.

Table 3: MCTS + CEM, RAVE Experiment Results

Method	Hyperparams	Method WR	Bot WR	Points/Game
+ CEM, RAVE	k=50	84%	6%	9.8 pts/game
+ CEM, RAVE	k=100	72%	18%	8.0 pts/game
+ CEM, RAVE	k=200	45%	21%	6.2 pts/game

We see that the performance of MCTS with CEM and RAVE degrades when k is too high. This means that we are holding β too high for too long, resulting in AMAF dominating long after it has enough data. This can result in the agent becoming biased, assuming that the move's value is independent of when it was played and allowing the agent to continue trusting a systematically incorrect assumption. The value Q associated with each node may also become correlated across actions, and a high k can result in over-propagation of lucky (or unlucky) rollouts. Thus, we settle on $k = 50$ and $c = 10.721$ as the best hyperparameters for MCTS with CEM and RAVE. Each of these experiments took about 356 minutes to run for 100 games.

However, the continued high-variance of random rollouts inspired the last extension: Counterfactual Regret Minimsation-trained rollouts. In the best case winning scenario, the player can receive over 60 points due to royalties. However, they can also end a round with -6 points due to fouling. CFR calculates the regret of playing one move over another, using counterfactual regret rather than raw utility for backpropagation. Using CFR also updates regrets at information-set granularity, allowing us to train regret across different full-game trajectories. This allows us to quickly learn generic rules such as avoiding putting two low pairs on the top hands, sacrificing immediately-optimal hands for longer-term benefit. CFR is complementary to UCT - while UCT handles exploration at the current root, CFR supplies a global prior that avoids obviously-bad moves across the whole game tree as learned over many games. The results from the experiments with different values of k with the full MCTS + CEM, RAVE, CFR implementation are as follows. Once again, the exploration constant is set to the optimal $c = 10.721$.

As shown in Table 4, the choice of hyperparameter k greatly influenced the performance of the model. This is due to k's control over balancing the AMAF statistic with UCT, as increased AMAF control results in increased AMAF noise. Since CFR controls variance, a higher k value is no longer

Table 4: MCTS + CEM, RAVE, CFR Experiment Results

Method	Hyperparams	Method WR	Bot WR	Points/Game
+ CEM, RAVE, CFR	k=50	75%	13%	7.9 pts/game
+ CEM, RAVE, CFR	k=75	89%	3%	11.2 pts/game
+ CEM, RAVE, CFR	k=100	68%	32%	6.7 pts/game
+ CEM, RAVE, CFR	k=125	42%	27%	3.2 pts/game

necessary to control noise. Rollouts are already fairly informative, making RAVE's integration of AMAF statistics less important (though still beneficial to MCTS). Compared to the results with just CEM and RAVE however, we note that the lower-variance and order-consistent rollouts mean that the CFR policy tends to play a move at roughly the same stage it appears in the real game. This means that the penalty for a big k is smaller, though it still is not 0.

All experiments took the same time to run. Training CFR for 200 iterations took approximately 133 minutes and 20 seconds. Running MCTS with CEM, RAVE, and (pre-trained) CFR took approximately 315 minutes to run for 100 games.

5.3 Proximal Policy Optimization (PPO)

Our goal for Proximal Policy Optimization was to learn a robust card-placement policy in the complex environment of Open-Face Chinese Poker. Our initial results supported our hypotheses that MCTS performs the best, with PPO coming in second. However, the primary constraint on PPO's win rate was the large number of games that resulted in ties.

To tackle this, we first introduced additional reward shaping in Table 5. Beyond the defined reward function, we layer a small negative penalty for ties and a modest bonus for scoops on top of the environment's native fouls and royalties. This gave the agent an extra gradient signal to break deadlocks. We then incorporated Generalized Advantage Estimation (GAE) as an extension to reduce variance in our advantage estimates, yielding smoother and more informative policy updates. Having strengthened the reward signal to encourage decisive play, we next turned to variance reduction in our policy updates.

Table 5: Reward-shaping hyperparameters.

Parameter	Value
Tie penalty, r_{tie}	-0.1
Scoop bonus, r_{scoop}	$+1.0$
Max. scoop diff., $D_{\max}$	3

Generalized Advantage Estimation (GAE) We compute temporal-difference residuals and GAE to reduce variance:

$$\delta_t = r_t + \gamma\, V_\theta(s_{t+1}) - V_\theta(s_t), \quad \text{GAE}_t = \delta_t + \gamma\lambda\, \text{GAE}_{t+1},$$

where $\gamma = 0.99$ and $\lambda = 0.95$. We set the advantage $A_t = \text{GAE}_t$ and the target return $R_t = A_t + V_\theta(s_t)$.

After training for 2,000 episodes, we assessed the agent's performance by sampling 100 games from its learned policy. As shown in Table 6, adding GAE increases the player win rate by about 10% and raises the average point differential from 3.72 to 5.02.

To ensure these findings weren't due to sampling noise, we also ran a larger-scale evaluation over 1,000 games (Table 7).

We see that incorporating Generalized Advantage Estimation (GAE) into PPO yielded a consistent 10% boost in win rates across both 100- and 1,000-game evaluations, demonstrating that lower-variance advantage estimates lead to more stable updates and more effective scoop-seeking strategies. Average point differentials also rose, indicating not only more wins but more decisive victories.

Table 6: Comparison of PPO agent performance over 100 games, with and without GAE.

Metric	No GAE	With GAE
Player wins	31%	41%
Ties	69%	59%
Bot wins	0%	0%
Avg. player score	3.72	5.02
Avg. bot score	-1.86	-2.46

Table 7: Comparison of PPO agent performance over 1,000 games, with and without GAE.

Metric	No GAE	With GAE
Player wins	26.9%	35.8%
Ties	73.1%	64.2%
Bot wins	0%	0%
Avg. player score	3.25	4.31
Avg. bot score	-1.61	-2.15

However, ties still dominate, reflecting the game's split-pot scoring, which naturally produces many deadlocks even when the agent aggressively pursues full-hand wins. Further reductions in draw frequency will likely require complementary techniques: stronger reward shaping (e.g. larger tie penalties or adaptive bonuses), entropy annealing to reduce randomness post-training, or auxiliary objectives that directly maximize scoop rates.

The 100 game experiments took approximately 20 seconds to run; 30 seconds for the 1,000 game experiments.

5.4 Comparing Methods

The hyperparameters for each of these methods are the same as described in previous subsections, though not repeated again for the sake of space.

Table 8: Comparison of Selected OFCP Agents

Method	Method WR	Bot WR	Points/Game	Total Time
Baseline Q-Learning	22%	35%	0.98 pts/game	33s
Double DQN	35%	23%	3.50 pts/game	4m 16s
PPO	41%	0%	5.02 pts/game	20s
MCTS + CEM, RAVE, CFR	89%	3%	11.2 pts/game	447m 03s

All results in Table 8 were run over 100 games. As shown in Table 8, comparing the best-performing versions of all three methods and the baseline Q-Learning, we find that our results somewhat support our first hypothesis. We initially hypothesized that the tested methods in this project (MCTS, Q-Learning, Deep Q-Learning, and PPO) show significant improvement over a bot placing random cards, displaying an ability to play strategically or learn patterns through self-play without relying on pre-existing data. All of our non-baseline methods achieve a higher win rate than the Random Bot over 100 games, though with varying degrees of success. Baseline Q-Learning fails to beat the Random Bot. Out of Double DQN, PPO, and MCTS, we see that Double DQN performs the worst. The Baseline Q-Learning model also does not outperform the Random Bot. This is because large state dimensions (728) makes tabular updates infeasible and the agents learn insignificant policies. Furthermore, Deep Q-Learning suffers from sparse/delayed rewards. Most card placements only affect final scoring much later, so the Q-network struggles to propagate useful value estimates back through 13 sequential decisions. High-dimensional inputs and extremely sparse feedback prevent stable convergence, even with 2,000 training episodes and experience replay.

Our results do, however, fully support our second hypothesis. We see that MCTS outperforms all other methods, beating the next highest method in Method Win Rate (PPO) by about 48%. MCTS also

holds the bot at a 3% win rate, though slightly above the PPO Bot Win Rate of 0%. However, MCTS has the lowest tie rate, showing a lack of double fouling. Specifically, the Monte Carlo Tree structure naturally mirrors the game's branching actions and allows efficient reuse of simulations, yielding low outcome variance by averaging hundreds of roll-outs. PPO and Q-Learning, by comparison, rely on a single value or policy estimate per move, making them more prone to conservative, split-pot outcomes. Furthermore, MCTS outperforms all methods with respect to points per game with over 2 times PPO's points per game. This outperformance is due to MCTS's ability to handle imperfect information and simulating possible outcomes with lower outcome variance compared to PPO and Q-Learning. Specifically, the Monte Carlo Tree structure provides a natural representation of the game state, as the three possible actions each player may take lends itself naturally to a branching tree. This tree then allows for efficient backtracking and reuse of game simulations within the MCTS algorithm. Averaging rollouts throughout this tree gives MCTS a lower outcome variance compared to PPO and Q-Learning due to averaging over hundreds of rollouts, whereas PPO and Q-Learning rely on a single prediction of the future. However, we see that MCTS requires significantly greater training time than the other three methods, showing a computational inefficiency. It requires over 100 times as long as Double DQN when accounting for training time and playing time, showing that the superior performance comes at a high time cost.

6 Discussion and Future Work

Our project does have a few notable limitations. We do not implement the Fantasyland and Shoot the Moon rules for Open-Face Chinese Poker. Instead, we solve the basic version of Open-Face Chinese Poker. In the future, additional complexity and strategy can be integrated by using these variants of Open-Face Chinese Poker. Furthermore, we do not include higher numbers of players, which can introduce more complexity as there are more hidden cards and potential bonuses to win off of opponents. As a single player's point score is tied to whether they beat another player's equivalent hand, adding more players would change the reward function and lead to interesting strategies such as targeting individual players to gain scoops and royalties while sacrificing to other players. Finally, from a model perspective, our best performing model is also computationally intensive. Testing MCTS further would be time-intensive, and the more complicated MCTS becomes with state-of-the-art heuristics or strategies, the more time it takes to solve a single game. For example, we see that implementing CFR raises the method win rate and the points per game but at the cost of over an hour of testing time.

The time requirements for running MCTS made extensive training difficult as mentioned. Implementing CEM was a workaround rather than running a full sweep of hyperparameters. Another challenge we faced during this project was dealing with tied games in particular. This difficulty has been discussed above. Sparse rewards also made PPO and Q-Learning difficult to implement, and it also made it nearly impossible to truncate the MCTS tree, resulting in longer runtimes.

This work has implications for strategy games in uncertain environments as a whole. In particular, the outperformance of MCTS demonstrates how effective the tree-based search method is for solving large state space games with uncertainty. This affirms previous literature which names MCTS as a recommended method for solving sequential decision making problems and games such as Go or Magic: The Gathering. However, our work also highlights the computational requirements for a high-performing agent.

7 Conclusion

In this project, we implemented three main methods to solve the strategic card game Open-Face Chinese Poker. Using Deep Q-Learning, PPO, and MCTS, we investigated two hypotheses. Namely, we showed that all tested methods in this project (MCTS, Deep Q-Learning, and PPO) improved over a bot placing random cards, displaying an ability to play strategically or learn patterns through self-play without relying on pre-existing data. In particular, we investigated reward shaping in our PPO implementation to better limit the number of tied games. We were also able to improve Double DQN over the baseline Q-Learning by decreasing the biases through different optimization. Secondly, we also successfully showed that MCTS performed better than other RL methods such as Q-Learning, Deep Q-Learning, and PPO. This was due to MCTS's ability to handle the large game space and moderate high-variance rollouts with the assistance of Rapid Action Value Estimation and

Counterfactual Regret Minimisation. We showed that MCTS is the preeminent method of solving Open-Face Chinese Poker based on its 89% Method Win Rate and 11.2 average points per game.

Future work may include optimizing the best-performing method MCTS to take less time to be a truly playable agent for real Open-Face Chinese Poker games. Additional future work may include testing new methods such as neuroevolutionary methods in combination with RL. This novel approach limits sparse rewards in imperfect information environments while still retaining RL's gradient-based method for higher learning efficiency, allowing for more effective learning. This would overcome the challenges faced by PPO and Q-Learning regarding the sparse reward space.

8 Team Contributions

All team members contributed to writing the final report and creating the poster for the poster session.

- **Alice Guo:** Alice implemented the PPO method with clipped surrogate optimization and entropy regularization. She also implemented the GAE extension.
- **Ramya Iyer:** Ramya implemented the Q-Learning baseline, DQN inspiration from related works, Dueling DQN extension, and Double DQN extension.
- **Isabella Lee:** Isabella implemented the custom game environment and reward function. She also implemented the MCTS method, CEM extension, RAVE extension, CFR extension, and ran the MCTS-associated experiments.

Our code can be found here: https://github.com/leejisabella/224rfinalproject.git.

9 Changes from Proposal

We originally planned to examine uses of RL in No-Press Diplomacy but realized after running initial experiments that it was not a feasible final project. In particular, it was not possible to retrieve the training data or Diplomacy game skeleton. The existing data and framework is blocked by Meta Research and requires a $5k fee, licensing, and setting up notarized NDAs and contracts requiring up to a month of processing. Furthermore, the Diplomacy dataset does not provide enough data to train an effective bot for no-communication games, as it was designed for negotiation-enabled Diplomacy games. As a result, our models were unable to reach the performance of even existing agents. As MCTS and Q-Learning both required self-play, the lack of a game skeleton or framework made the project impossible to complete as we could not compare our results against the existing literature, which relied on the Meta framework.

We instead pivoted to building an agent to solve open-face Chinese poker, a strategic card game with similar game characteristics to Diplomacy. This makes solving open-face Chinese poker a valuable benchmark for testing the limits of current Multi-Agent Reinforcement Learning (MARL) techniques. With the pivot to open-face Chinese poker, our project no longer relied on any external datasets. Instead, we were able to build a custom game environment and use a self-play reinforcement learning framework where the agent plays against a baseline agent. This self-generated data enables the model to iteratively improve its policy without human supervision or previously-generated data. While we initially intended to implement SARSA in addition to our other three methods above, we found that it was more beneficial to improve the performance of our three main methods. Please see the modifications we made in the Results section as we implemented our methods.

References

Noam Brown and Tuomas Sandholm. 2018. Superhuman AI for heads-up no-limit poker: Libratus beats top professionals. *Science* 359, 6374 (2018), 418–424.

Jakob N. Foerster, Gregory Farquhar, Triantafyllos Afouras, Nantas Nardelli, and Shimon Whiteson. 2018. Counterfactual Multi-Agent Policy Gradients. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*. IFAAMAS, 297–305.

John Kirklin. 2014. Kachushi: An Artificial Intelligence for the Game of Open-Face Chinese Poker. `https://scrambledeggsontoast.github.io/2014/06/26/artificial-intelligence-ofcp/`. Accessed: 2025-06-01.

V. Luo, A. Patel, and L. Zhou. 2018. Building a Pineapple AI: A Monte Carlo Tree Search-Based Approach. (2018). Unpublished manuscript.

Matej Moravčík, Martin Schmid, Neil Burch, Viliam Lisy, Dustin Morrill, Nolan Bard, Trevor Davis, Jaromír Čižek, Frederic Davis, Kevin Waugh, Michael Bowling, and Michael Johanson. 2017. DeepStack: Expert-Level Artificial Intelligence in Heads-Up No-Limit Poker. *Science* 356, 6337 (2017), 508–513. `https://doi.org/10.1126/science.aam6960`

Daming Shi, Xudong Guo, Yi Liu, and Wenhui Fan. 2022. Optimal policy of multiplayer poker via actor-critic reinforcement learning. *Entropy* 24, 6 (2022), 774.

Andrew Tan and Jarry Xiao. 2018. Mastering Open-face Chinese Poker by Self-play Reinforcement Learning. (2018).