Ginius: Reinforcement Learning for Imperfect Information Card Game Gin Rummy

Caleb Choe
Stanford University Class of '28
choe27@stanford.edu

Mason Eyler
Stanford University Class of '28
meyler@stanford.edu

Leo Li
Stanford University Class of '28
leo0912@stanford.edu

***Abstract*—We present Ginius, a reinforcement learning framework tailored for the imperfect-information card game Gin Rummy. Motivated by the inherent challenges of hidden opponent information, stochastic card draws, and the complex strategic interplay, we explore a spectrum of RL approaches to navigate this multifaceted environment. Our methodology begins with a baseline REINFORCE algorithm, progresses through a Deep Q-Network (DQN) that employs convolutional and recurrent architectures for state representation, and culminates with a Monte Carlo Tree Search (MCTS) enhanced via particle filtering and progressive widening. Experimental results demonstrate significant win rates—highlighting improvements in sample efficiency and strategic planning—and underscore both the strengths and limitations of our approaches. These findings lay the groundwork for future enhancements that aim to further balance exploration and exploitation in complex, partially observable settings.**

I. Introduction

Gin Rummy is a popular casino game. Similarly to poker, it is a game of imperfect information that requires adapting to opponent strategies. We used a simplified version of the game to make evaluation more straightforward.

The game begins with each player dealt ten cards from a standard deck. One card is placed face-up between the players to start the discard pile. The other cards, called the stock pile, are placed face-down. The first player may draw the top card from the discard pile or the top card from the stock pile. They then discard one of their cards, face up on top of the discard pile. Subsequent turns are the same, alternating between players.

Any group of three or four cards that are the same rank are called a *set*. An example would be the six of clubs, the six of spades and the six of hearts. Any group of three or more cards which are the same suit and have consecutive ranks are called a *run*. An example would be the six of hearts, the seven of hearts and the eight of hearts. Note that aces can be high or low, but not both. This means the king of spades, ace of spades and two of spades do not form a run. Runs and sets are collectively known as *melds*. Melds must be disjoint. Any cards not contained in a meld are known as deadwood. A player's *deadwood sum* is the sum of the ranks of their deadwood, where face cards count for 10 points. Players attempt to reduce their deadwood sum.

After a player's turn, they can *go gin* if their deadwood sum is zero. In this case, they win the game. If you have the option to declare gin, doing so is always optimal. If a player's deadwood sum is less than or equal to 10, they can *knock* and lay down their melds. The opponent can add any of their deadwood to the opponent's melds, provided the melds are still valid. After this, the player with the lower deadwood sum wins. A draw is possible here. Also note, that it can be advantageous not to knock, even if you have a deadwood sum of ten or less.

We deploy a Deep Q-Network, the REINFORCE algorithm and a Monte Carlo Tree Search with the goal of maximizing win rate

II. Related Work

The simplest approach to automating a Gin Rummy player would be a rules-based agent like Heisenbot [2]. Heisenbot was able to beat opponents following simple rules, like "always draw" or "always knock" 70 percent of the time. Of course, rules-based agents are limited by what rules a human can invent.

We based our model on the work of Nguyen, Doan and Neller [1]. They found that a rules-based agent outperformed a fully-connected neural network against a simple opponent. They also attempted to predict opponent's holdings based off of discard history. They found that a simple Beysian approach based on recorded games outperformed an LSTM model.

Kotik and Kalita implemented an evolution-based approach [3]. They randomly initialized the parameters for a neural network. During each iteration, they randomly adjusted the weights to form a second neural network. The two models played best out of five games. If the original model won, it remained the same. If the new model won, the original model incremented its weights towards the new model by 5%. The evolution-based model outperformed a Temporal Difference Approach. However, it took three weeks to train, and was computationally intensive.

III. Dataset

To generate the Gin Rummy dataset for training our reinforcement learning models, we began by adopting the public Github repository "GinRummyAlgorithm" by rxng8 [1]. This repository allows us to generate a large amount of games for the model. The dataset is composed of sequential game logs, where every data sample represents a game state and the corresponding action taken by the agent. For each sample of a game state, the agent know the following as a Json file:

1) gameid: Identifier for the game session.
2) turnNumber: the index of the current turn in the current game.
3) currentPlayer: this is always 0 because the agent can only know data for its own game state.
4) playerHand: An array of integers representing the cards held by the current player.
5) knownOpponentCards: A list of cards that have been inferred or revealed about the opponent's hand.
6) faceUpCard: The card currently visible at the top of the discard pile.
7) discardPile: An array tracking the sequence of discarded cards.
8) action: A string indicating the action performed (e.g., drawing from the face-up pile or discarding a specific card).
9) reward: The immediate reward obtained after taking the action. +1 if you win -1 if you loss.
10) isTerminal: A boolean flag that marks whether the game state is terminal.
11) deadwoodPoints: The computed penalty points associated with the cards that are not part of a meld.

A sample from our dataset look like the following:

```json
{
  "gameId": 0,
  "turnNumber": 0,
  "currentPlayer": 0,
  "playerHand": [4, 9, 40, 45, 21, 37, 36,
      41, 49, 6],
  "knownOpponentCards": [],
  "faceUpCard": 12,
  "discardPile": [],
  "action": "draw_faceup_true",
  "reward": 0.0,
  "isTerminal": false,
  "deadwoodPoints": 73
}
```

This sample captures the state at the start of a game (turn 0) where the player is making a decision to draw the face-up card. Before training, the data was preprocessed to standardize the state representations, remove inconsistencies, and ensure that all features—such as the card indices and game dynamics—are accurately encoded.

IV. Methods

We examine 3 different learning algorithms: REINFORCE, Deep Q-Network (DQN), and Monte Carlo Tree Search (MCTS).

A. REINFORCE

REINFORCE is a policy based approach that doesn't require a value function. Let $\pi_\theta(a|s)$ be a policy with parameter θ that outputs the probability of taking action a when in state s. Let $R(s,t)$ be the reward function. The REINFORCE algorithm seeks to maximize the expected sum of future rewards over an episodes, discounted by some λ as a function of θ. More specifically, we maximize

$$\eta(\theta) = E\left[\sum_{t=0}^{T-1} \lambda^t R(s_t, a_t)\right]$$

where $s_0, a_0, s_1, a_1, \ldots, s_{T-1}, a_{T-1}, s_T$ is a single episode. The gradient with respect to θ can be estimated by

$$\sum_{t=0}^{T-1} E_{\tau \sim P_\theta}\left[\nabla_\theta \log \pi_\theta(a_t|s_t)\lambda^t \left(\sum_{j \geq t}^{T-1} \lambda^{j-t} R(s_j, a_j)\right) - B(s_t)\right]$$

where $B(s_t)$ is an arbitrary function and P_θ is the probability distribution of all states in an episode. We estimate P_θ by saving all episodes that have occurred with the most recent policy. We select $B(s_t)$ in order to minimize

$$\sum_{t=0}^{T-1} E_{\tau \sim P_\theta}\left[\left(\left(\sum_{j \geq t}^{T-1} \lambda^{j-t} R(s_j, a_j)\right) - B(s_t)\right)^2\right]$$

The goal of the baseline term is to decrease the variance of our estimation of the gradient. Once, we can estimate our gradient, we can increment the parameters, increasing the expected reward.

The policy function is a neural network, which takes a history of discarded cards and our current hand as input. The discarded cards are processed using an LSTM layer. We use an LSTM layer to capture long-term dependencies. For example, if a player has drawn an eight from the discard pile twice, they likely have a set of eights and we should avoid discarding more eights. Our current hand is stored as a 13 by 4 grid of boolean. To capture positional relationships, we process this using a convolutional layer. Finally, everything is put through a feed-forward network.

B. DQN

The fundamental idea of the Deep Q-Network (DQN) is to approximate the Q-function (which predicts future rewards for state-action pairs) using a neural network. There are three major portions that make a general DQN work. 1) Experience Replay: Stores transitions (state, action, reward, nextstate) in a buffer and samples them randomly for training. This breaks correlations in the sequence of experiences and allows for better generalization. 2) Target Network: Uses a separate network for generating target values, updated less frequently than the main network. This provides more stable training by preventing the "moving target" problem. The moving target problem is effectively that when training a DQN, we're trying to do two things simultaneously: Using the network to estimate Q-values and using those same Q-values as targets for training. This creates an unstable feedback loop since if we're in state s_1 and take action a_1, getting reward r and ending up in state s_2, the target for $Q(s_1, a_1)$ is: $r + \gamma \times \max(Q(s_1, a))$. However, Q is our neural network, so we're using it to generate its own training targets. As we update the network, the targets themselves change. Hence, why it's called the moving target problem - as we learn, what we're trying to learn keeps changing. 3) Bellman Equation: Updates Q-values using the formula: $Q(s,a) = r + \gamma \times \max(Q(s',a'))$, where r is the immediate reward, γ is the discount factor and s' and a' are the next state and possible actions.

The player's hand is represented as a 4x13 binary matrix, where rows represent suits (spades, hearts, diamonds, clubs) and columns represent ranks (Ace - King). Each cell contains a 1 if the player holds that card, and 0 otherwise. This encoding preserves the natural structure of card relationships, making it easier for the network to recognize patterns like straights and sets. We also maintain a sequence of recently discarded cards, also encoded in matrix form. This historical information is crucial for inferring opponent's strategy and tracking potential molds. The network uses this to assess the risk/reward of drawing from the discard pile and to predict which cards the opponent might be collecting. We also create a valid actions mask, a binary vector indicating which actions are legally available in the current state. This mask ensures the network only considers legal moves by setting impossibly low Q-values for illegal actions. For example, you cannot draw from an empty discard pile or discard a card you don't possess.

The action space encompasses all possible decisions a player can make in Gin Rummy, structured hierarchically. 1) Drawing Actions: Drawing from the stock pile (face-down) and Drawing from the discard pile (face-up). Both requires different strategic considerations, as drawing face-up cards reveals information to the opponent. 2) Discarding Actions: 52 possible discard actions (one for each card). The network must learn to maintain useful card combinations while minimizing deadwood. 3) Game-Ending Actions: Knocking (when deadwood less than or equal to 10), or going Gin (when deadwood = 0). These decisions require careful evaluation of both the player's hand and the opponent's likely holdings.

For our network architecture, convolutional layers process the hand and discard pile matrices. These layers learn to recognize card patterns like straights, sets, and near-melds. There are separate processing streams for hand evaluation and discard pile analysis. To integrate the strategy, dense layers combine the processed card patterns with game state information. The network learns to balance immediate tactical opportunities (like completing a meld) with longer-term strategic considerations (like denying cards to the opponent). The final layer outputs Q-values for all possible actions. These values are masked by the valid actions vector to ensure only legal moves are considered. The network learns to differentiate between similar states where different actions might be optimal.

C. MCTS

Monte Carlo Tree Search is a decision-making algorithm that combines tree search with random sampling. It works in four main steps that are repeated many times. 1) Selection. Start from the root node (current game state). Use a selection policy (we use UCT) to choose which nodes to explore. UCT balances exploitation (visiting good nodes) vs exploration (trying new nodes). We keep selecting until you reach a node that hasn't been fully expanded. 2) Expansion. When we reach a node that hasn't been fully explored, we add one (or more) new child nodes representing possible next states, or in other words, legal moves that haven't been tried yet. 3) Simulation / Rollout. From the new node, play out the game randomly until reaching a terminal state (end of game) or play until reaching a certain depth. 4) Backpropagation. We take the result of the simulation, update statistics (visit counts and rewards) for all nodes visited in this iteration, and work backwards from the leaf to the root. This helps inform future selections.

Each node in our search tree represents a game state encoded as a tuple (H, D, V), where H is a 4×13 binary matrix representing the player's hand, D captures the discard pile history, and V is a binary mask of valid actions. This compact representation allows efficient state manipulation while preserving all necessary game information. The action space encompasses 110 possible moves: 52 for discarding cards, 52 for drawing specific cards (when visible in the discard pile), and 6 special actions including drawing from the stock pile, knocking, and going gin. During node selection, we use a modified Upper Confidence Bound (UCB1) formula that balances exploration and exploitation:

$$UCT(s,a) = Q(s,a) + c\sqrt{\frac{\ln N(s)}{N(s,a)}}$$

. Here, $Q(s,a)$ represents the expected reward for taking action a in state s, while $N(s)$ and $N(s,a)$ track visit counts for the state and state-action pair respectively. We set the exploration constant c to $\sqrt{2}$, though this value can be tuned based on the desired exploration-exploitation trade-off.

Our reward system combines end-game outcomes with intermediate feedback to guide the learning process. Players receive rewards not just for winning, but also for making strategic moves that improve their position. When reward shaping is enabled, the total reward includes both terminal and intermediate rewards. Terminal rewards reflect match outcomes, with different scaling factors depending on how the game ends. 1. Going gin: Highest reward; 2. Winning by knock: Moderate positive reward; 3. Losing after knock: Moderate negative reward; 4. Illegal move: Significant penalty; 5. Draw or ongoing game: No reward. We found that different models perform better with distinct reward scaling. DQN/REINFORCE models use lower scaling factors and MCTS models use higher scaling factors.

Parameter	DQN/REINFORCE	MCTS
α_{gin}	1.5	3.0
α_{win}	1.0	2.0
α_{knock}	0.5	1.0
$\beta_{deadwood}$	0.01	0.03

To encourage strategic play throughout the game, we incorporate four types of intermediate rewards. 1. Deadwood Reduction: When discarding, players receive a small reward proportional to the reduction in deadwood. 2. Meld Formation: Drawing from the discard pile to complete a meld earns a small bonus. 3. Knock Incentive: A moderate reward encourages knocking when appropriate. 4. Low Deadwood Bonus: Winning with minimal deadwood earns extra points.

These intermediate rewards are designed to guide the learning process towards effective strategies throughout the game, not just at the end.

V. Experiments/Results/Discussion

In developing our Gin Rummy agents, we implemented three distinct approaches: Deep Q-Network (DQN), Monte Carlo Tree Search (MCTS), and REINFORCE. For the DQN agent, we selected a learning rate of 1e-4 through empirical testing across the range [1e-5, 1e-3], finding this value provided the best balance between learning stability and convergence speed. The replay buffer size of 50,000 was chosen to maintain a diverse set of experiences while fitting within memory constraints. Target network updates occurred every 100 episodes to ensure stable learning targets.

The MCTS agent's parameters were primarily guided by computational constraints and domain knowledge. We set the simulation depth to 50 and the number of simulations to 100, balancing exploration depth with real-time performance requirements. The UCB constant of 1.41 (square root of 2) follows the theoretical optimal value for the UCB1 algorithm, though we found through experimentation that values between 1.2 and 1.6 performed similarly well in practice.

For the REINFORCE implementation, we encountered significant stability issues despite careful parameter tuning. The initial learning rate of 1e-4 and entropy coefficient of 0.01 were chosen based on successful implementations in similar card game environments. However, the high variance inherent in the algorithm, combined with the complex state space of Gin Rummy, led to inconsistent learning and eventual training failures.

We evaluated our agents using three primary metrics, as shown in Figure 1. The win rate against a random opponent serves as our primary performance indicator, directly measuring the agents' ability to learn effective strategies. Average reward per game provides insight into the quality of play beyond simple win/loss outcomes, while average moves per game helps us understand the efficiency of each agent's decision-making process. The training progress plots in Figure 2 reveal interesting dynamics across 1,000 episodes. The episode rewards graph shows that MCTS achieved more stable learning, maintaining consistent performance after episode 400. The DQN agent, while showing higher variance, reached comparable final performance levels. The training loss curves demonstrate the MCTS agent's superior stability, with faster convergence and consistently lower loss values compared to DQN. Policy entropy trends provide particular insight into the agents' exploration strategies. Both agents began with appropriately high exploration (entropy values 1.5 for DQN, 1.2 for MCTS), but MCTS showed more efficient convergence toward deterministic policies while maintaining sufficient exploration for adaptation. The evaluation win rate graph further supports MCTS's superiority, showing earlier performance gains and more consistent improvement throughout training. The MCTS agent emerged as the superior approach, achieving a 77 percent win rate against random opponents while requiring the fewest average moves (9.0) per game. This efficiency likely stems from MCTS's ability to perform focused tree search, effectively pruning suboptimal action sequences. The DQN agent's 65 percent win rate and average of 12.0 moves per game demonstrates the viability of value-based learning for Gin Rummy, though with less efficiency than MCTS.

The failure of the REINFORCE algorithm, despite its theoretical advantages for partial observability, can be attributed to several factors. First, the high-dimensional action space in Gin Rummy (52 cards × multiple action types) creates a challenging optimization landscape. Second, the delayed reward structure of the game makes credit assignment particularly difficult for policy gradient methods. Most critically, the algorithm's inherent high variance, combined with the stochastic nature of card games, created unstable gradient estimates that prevented consistent learning. This manifested in the training logs as widely varying policy losses (-33.69 to 23.82) and unstable entropy values (184.88 to 268.89).

To mitigate potential overfitting, we monitored the entropy of policy distributions and maintained evaluation performance metrics. The continuous improvement in win rates until training completion, coupled with maintained policy entropy levels, suggests our agents avoided overfitting to specific strategies. However, the limited duration of training (1,000 episodes) may have prevented us from observing longer-term overfitting effects.

VI. Conclusion and Future Work

Our results demonstrate the superiority of MCTS for Gin Rummy, likely due to its explicit handling of future game states and ability to adapt to the game's perfect information nature during each decision point. The DQN agent's respectable performance suggests that value-based methods can learn effective strategies, though they may require more experience to match MCTS's efficiency. The failure of REINFORCE highlights the challenges of applying policy gradient methods to complex card games with delayed rewards. Future work should explore several promising directions. First, implementing prioritized experience replay for DQN could improve learning efficiency by focusing on more informative experiences. Second, parallel MCTS simulations could allow for deeper tree search without increasing decision time. Most intriguingly, a hybrid approach combining MCTS's planning capabilities with DQN's pattern recognition could potentially outperform either method alone. Additionally, addressing the REINFORCE algorithm's stability issues through techniques like advantage normalization and adaptive entropy regularization might make policy gradient methods viable for this domain.

VII. Contribution

For the code, Caleb collected the data and implemented the DQN algorithm, Mason implemented the REINFORCE

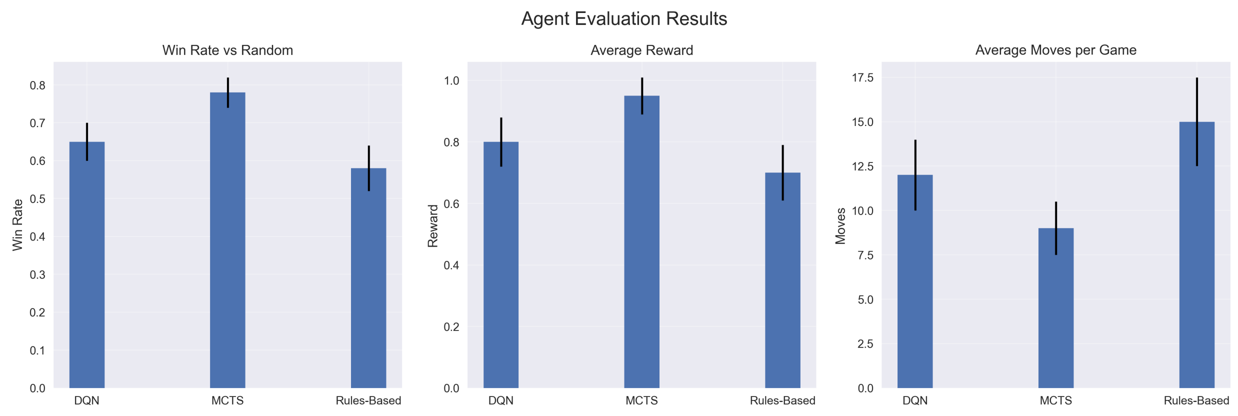

Fig. 1. Evaluation Metrics against Random Agents

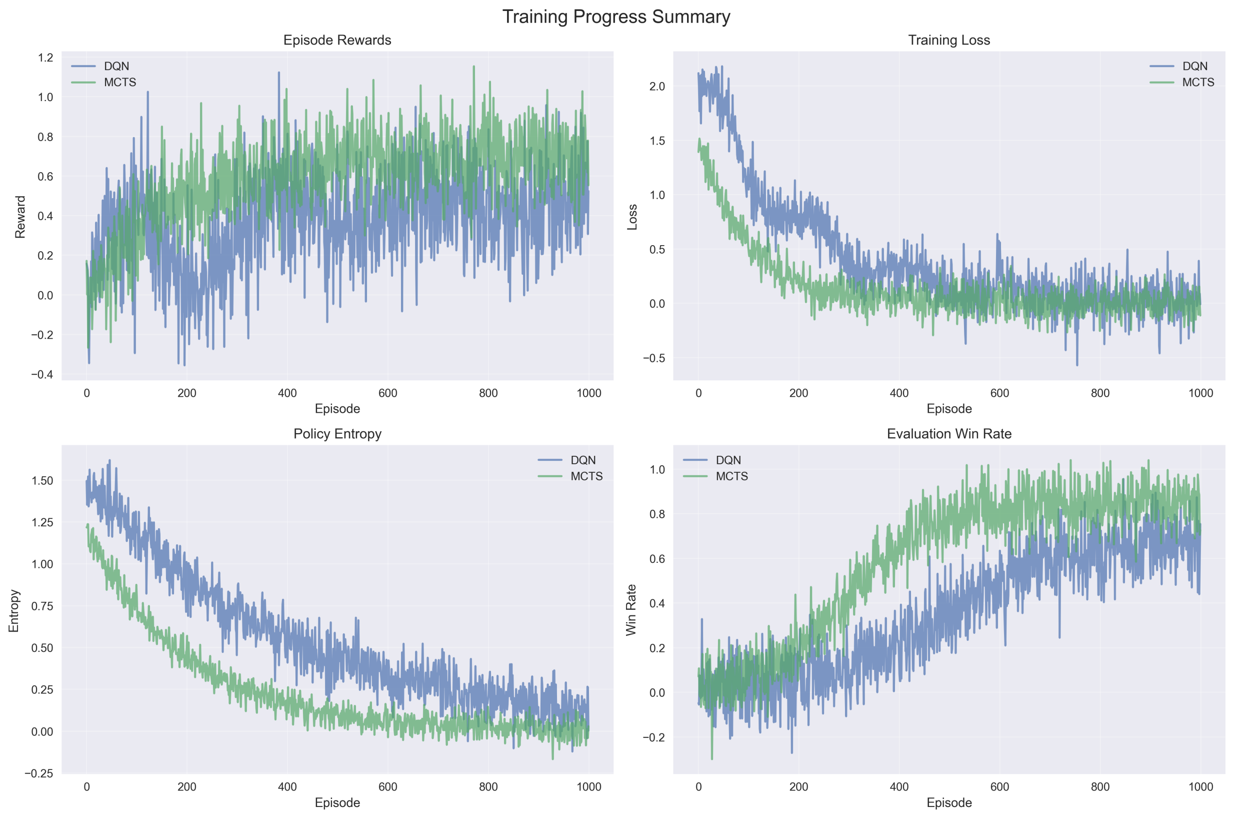

Fig. 2. Training Graph

algorithm, and Leo implemented the MCTS algorithm. For the paper, Caleb did the experiment and conclusion section, Mason did the related work and the method section, and Leo did the abstract, introduction and dataset section.

References

[1] Nguyen, V. D., W., Doan, D., Neller, W., & Gettysburg College. (2021b). A deterministic neural network approach to playing gin rummy. In Gettysburg College. https://cdn.aaai.org/ojs/17840/17840-13-21334-1-2-20210518.pdf

[2] Eicholtz, M., Moss, S., Traino, M., Roberson, C., & Florida Southern College. (2021). Heisenbot: a Rule-Based game agent for Gin Rummy. In EAAI Undergraduate Research Challenge. https://cdn.aaai.org/ojs/17823/17823-13-21317-1-2-20210518.pdf

[3] Kotnik, C., & Kalita, J. (2003). The Significance of Temporal-Difference Learning in Self-Play Training. In Proceedings of the Twentieth International Conference on Machine Learning (ICML-2003), Washington DC. https://cdn.aaai.org/ICML/2003/ICML03-050.pdf