IntelliGin: Deep Q-Learning for Gin Rummy

Peter Curtin **Evan Lu** **Michelle Liu**

Brown University

{peter_curtin, evan_lu, michelle_h_liu} @ brown.edu

1 Introduction

Applying artificial intelligence (AI) techniques to games has long been a challenge, and milestones such as defeating human players in games like Chess and Go have demonstrated the immense potential of these algorithms. Recently, there has been growing interest in applying reinforcement learning (RL) techniques to a variety of games, including Atari games, Dota 2, and even Starcraft [4] [6] [8].

Building on the successes of reinforcement learning techniques on other games, our research explores applying similar RL techniques to Gin Rummy. Gin Rummy is a popular two-player card game, where the objective is to form sets or straights of cards in your hand and *knock*. At each step, there are only a few possible moves – draw from discard or draw piles, and then discard, possibly knocking and ending the hand. It presents an interesting challenge for AI because, despite its simple rules, there is complex strategy involved both in forming sets and discarding.

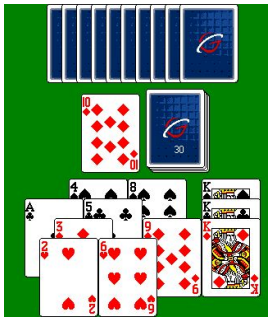

Figure 1: Screen shot from online Gin Rummy games

Despite its simplistic ruleset, Gin Rummy is an interesting candidate for reinforcement learning research due to its lack of complete information, complicated state space, sequential nature, and inherent complexity. Though at any given point the only actions are to draw from either the deck or discard pile, a player must consider past details, such as but not limited to previous cards discarded by both players, cards the opponent has drawn, and cards yet unseen, to form a competitive strategy. Furthermore, the game's scoring system leads to a sparse reward structure. A simple reward function can be crafted where agents with less deadwood get higher rewards and the winning agent gains an additional bonus. Due to the vast and discrete state space, carefully selecting an effective algorithm is crucial for success. Moreover, with such a large state space, any reinforcement learning approach must include significant exploration to succeed.

The objective of this paper is to explore the applicability of deep learning frameworks, namely deep Q-learning, to Gin Rummy. We will specifically measure the impact of different featurizations of the observation space, sequential versus non-sequential models, and the effect of different adversarial

policies used during training. By evaluating these different approaches, we hope to understand how best to solve Gin Rummy using deep Q-learning, laying the groundwork for future work in the environment. Optimally, we hope to produce an agent that can consistently win against random and simple algorithmic agents, and even play competitively against human agents. Our specific research goals are:

1) To develop an environment for simulating Gin Rummy with RLCard or independently, and to modify the environment to be capable of providing engineered features that contain information about previous states. Further, to modify the environment to allow a representation of a simplified Gin Rummy game.
2) To train a simple Q-Learning model on the engineered features against a random policy and a more capable adversary.
3) To train a recurrent model on the vanilla features and test against a random policy and a more capable adversary.
4) (Reach Goal) To train the simple model or the recurrent model against itself using self-play, and compare its performance with the previously trained models.

2 Background and Related Work

2.1 Introduction to Gin Rummy

Gin Rummy is a popular 2-player strategic card game. The game is played with a standard 52-card deck. The cards are ranked from lowest to highest by $\{A, 2, ..., J, Q, K\}$. The objective of Gin Rummy is to form *melds* of three or more cards. *Melds* are defined as either sets of cards of the same rank or straights of cards of the same suit. There is no wraparound, that is, $\{K, A, 2\}$ would not form a valid straight but $\{10, J, Q, K\}$ would. One important concept in Gin Rummy is *deadwood*, the sum of the values of cards not in a meld. Aces are worth 1, and J, Q, and K are all worth 10 – all other cards have deadwood points equal to their face value.

To start the round, players are dealt 10 random cards from the deck. Gameplay starts with the non-dealer – and the dealer for each round is the winner of the previous round. Players alternate turns drawing a card from either the discard (face-up) or draw (face-down) piles. They then must discard a card into the discard pile. When their deadwood falls below a certain threshold, typically 10, they have the option to *knock* and end the round. Scoring is based on the difference in deadwood between players, where the player with the lower deadwood adds the difference to their total score. The game starts anew until some player reaches a total score count above some threshold, typically 100. Obtaining a *Gin*, or knocking with 0 deadwood, equates to a 25 point bonus [1].

Although the rules are not complex, Gin Rummy involves a lot of strategic elements and decision-making that are critical to success. On each turn, it is important to consider which cards to keep for making potential melds and which to discard – keeping in mind deadwood. Although one cannot directly observe an opponent's hands, the cards that an opponent takes and discards provide information about an opponent's hand or strategy. A player may withhold cards from their opponent, for instance. Because of the significant Gin bonus, there may also be times when it makes sense to not knock even when below the threshold.

Because deadwood is directly associated with a player's score, a simple Gin Rummy strategy involves *deadwood minimization*. An example of such a simple agent is implemented by RLCard, the `NoviceGinRummyAgent`. On its turn, the agent seeks to Gin or knock if possible. Otherwise, it places the draw card in the best possible meld and discards the highest deadwood card [2].

2.2 Deep Q-Learning

Q-learning, a model-free classic reinforcement learning algorithm, addresses sequential decision-making processes by learning an optimal action-value function representative of the existing environment. Q-values are algorithmically assigned to each action and iteratively updated to estimate the rewards for taking the respective action. During evaluation, the action with the highest associated Q-value is chosen as the optimal action.

Q-learning's capacity to learn optimal policies within MDPs has kept the algorithm relevant. Recent developments in deep learning have given rise to deep Q-learning, also known as deep Q-networks (DQN), which utilize neural networks to approximate Q-values effectively.

The recent successes of DQN implementations span many domains, highlighting their effectiveness in various complex challenges. The landmark Mnih et al. (2013) paper showcases the capacity of DQNs in Atari 2600 games, demonstrating the algorithm's potential and setting a benchmark for learning from raw image inputs [6]. Since then, DQNs have arisen in fields such as, but not limited to robotics, natural language processing, medical imaging, and more. In the realm of strategic card games, DQNs have been applied to Texas hold 'em poker [9].

2.2.1 Double DQN

In a standard Q-learning, the target Q-value is given by

$$Y_t = R_{t+1} + \gamma \max_a Q_\pi(S_{t+1}, a).$$

However, using the max operator introduces a bias where overestimated values are more likely to be selected. As Q-learning learns new estimates from prior estimates, this can lead to suboptimal policy decisions. Double Q-learning and subsequently double deep Q-networks (Double DQN) are an extension of Q-learning designed to address this overestimation bias.

Double Q-learning involves using two independent Q-value functions that update each other. At each step, the *target* Q-function is used to determine the best action while the *online* Q-function is used to estimate the value of that action. By decoupling the selection of actions from their evaluation, double Q-learning reduces overestimation bias as opposed to using the same function to both select actions and evaluate.

Deep double Q-learning (DDQN) extends double Q-learning to deep neural networks like that in DQN. Rather than two Q-functions, however, it utilizes two separate neural networks – a *target* and *online* network which function similarly to their Q-function counterparts in double Q-learning. In DDQN, the target Q-value is give by

$$Y_t^{\text{DoubleDQN}} \equiv R_{t+1} + \gamma Q(S_{t+1}, \text{argmax}_a Q(S_{t+1}, a; \boldsymbol{\theta_t}); \boldsymbol{\theta_t^-}).$$

In this equation, $\boldsymbol{\theta_t}$ and $\boldsymbol{\theta_t^-}$ are the two sets of weights corresponding to the online and target networks, respectively. R_{t+1} corresponds with the immediate reward and the resulting state S_{t+1} after taking an action, and γ is the discount factor. Periodically, $\boldsymbol{\theta_t}$ is copied to $\boldsymbol{\theta_t^-}$. This equation doesn't modify the original double Q-learning algorithm significantly in order to retain the benefits of double Q-learning [3].

2.3 Related Work

There is little prior work on the topic of solving Gin Rummy using reinforcement learning. In Kotnik et al. (2003), the authors explore an online, temporal difference learning approach to learning a value function for the observations in Gin Rummy, as well as applying an evolutionary approach to train a similar model [5]. Interestingly, the evolutionary model, EVO-rummy, was shown to achieve better performance. The discussion of the results outlined that the evolutionary approach generally learned more sensible policies that had no strong suit preference and valued lower cards more than higher cards. Furthermore, the authors assumed that due to the stochastic nature of the environment, no explicit exploration was required. They suggest evaluating the effect of further exploration on the online policy.

Next, in Nguyen et al. (2021), several architectures/methods were used to train hand estimation models and card value estimators from an offline dataset of example games played between two simple algorithmic agents [7]. Notably, they trained a long short-term memory (LSTM) model to estimate the opponent's hand. Rather than using any RL models directly for action selection, they apply deterministic algorithms to the model outputs to determine statistically best actions.

Although there have been papers on Gin Rummy and DQN separately, applying modern deep Q-learning techniques to Gin Rummy has never been properly studied. Epsilon-greedy exploration commonly deployed with Q-learning may offer exploration advantages and help learned agents perform better than the TD-Rummy agents. Additionally, the online learning approach may result in more sophisticated policies than those presented in Nguyen et al. (2021) [5][7].

3 Method

Representing Gin Rummy in a computational medium requires implementations for a representation of the game environment, a model for an agent that interacts with the game, and a reward scheme. A standardized method of training is additionally required when comparing different implementations and approaches.

Research questions that guided our investigative process were the importance of the degree of the state space's abstraction, sequential representations – namely if simple observations are sufficient for sequential decision-making or if a hidden state with encoded past experiences is necessary – and the impact of adversarial policy during training.

Our code is available on our GitHub repository (`https://github.com/pmcurtin/intelligin`). Our `DoubleDQNAgent` implementation is inspired by RLCard's existing implementation, such that it interfaces most easily with existing RLCard environment functions.

3.1 Environment Setup

For all experiments, we use RLCard's provided Gin Rummy environment, in some cases with some modifications. In the RLCard environment, each game state is organized into one of two phases, either a draw phase or a discard/knock phase. A given player's turn starts in the draw phase, where the only two valid actions are to either draw a card from the draw pile or the discard pile. Immediately afterward, the player's turn continues into the discard/knock phase, in which they can Gin, knock using a specific card, or simply discard a card. Of course, they can only Gin if they have exactly one non-melded card in their hand, and can only knock if they have less than 10 deadwood, under the standard rules. In this way, each player always has two action turns in a row.

Before each action turn, the player receives an observation of the current game state. By default this consists of five binary vectors of length 52, which we'll reference as the *rich* observation representation, indicating:

1. The cards in the player's hand (always exactly 10 or 11).
2. The top card of the discard pile (one-hot).
3. The cards (other than the top one) known to be in the discard pile.
4. The cards the opponent drew from the discard pile and has not discarded yet.
5. The cards with unknown locations (either in the draw pile or in the opponent's hand).

This representation contains information from previous states, namely the three last vectors. Thus, we modified the environment (`ModifiedGinRummyEnv`) to also offer less generous observation features, creating the *simple* observation representation. The modified environment provides three binary vectors of length 52 as an observation:

1. The cards in the player's hand (exactly 10 or 11).
2. The top card of the discard pile (one-hot).
3. Card the opponent just drew from the discard pile, if applicable.

This representation reflects the immediate information available to an agent playing the game, with no information from previous observations.

The comparison of our modified state representation with the original addresses our first research question of the importance of state abstraction. In section 4, we compare the performance of models trained on *simple* and *rich* observations.

The action space has a size of 110. Of these, most importantly, there are 52 actions each corresponding to discarding and knocking with particular cards, a Gin action, and two drawing actions indicating whether to draw from the discard pile or the draw pile. The environment includes several utility actions for ending the game in the case of a dead hand (draw pile exhausted) or scoring the players.

The reward model is simple and sparse. At the end of the game, players are assigned the following rewards:

1. If the player Gins, they are awarded a score of 1.

2. Otherwise, if a player knocks, they are awarded a score of 0.2.

3. If a player loses (does not knock, or dead hand), they are awarded $\frac{-\text{deadwood}}{100}$.

Finally, several parameters can be modified to change the rules of the game. For example, the `going_out_deadwood_count` parameter, with a default value of 10, determines the maximum amount of deadwood a player can have to knock [2].

3.2 Model

Agent models were implemented based on 1) a DoubleDQN implementation for stability and to normalize overestimation error, as motivated in section 2.2.1 and 2) a recurrent implementation to evaluate the efficacy of a sequential implementation. The underlying architecture we used is a multi-layered perceptron (MLP) with an input dimension of the state space and an output dimension of action space, where the output index with the highest value is selected epsilon-greedily as the action taken. The underlying architecture for the recurrent implementation is a gated recurrent unit (GRU) with input dimensions of the state space and hidden dimension and with output dimensions of the action space and hidden dimension, where the output index with the highest value is selected epsilon-greedily as the action taken and the hidden state is passed back into the model for the next step.

The comparison of these two model architectures addresses our second research question of the importance of sequential representations. The double-DQN architecture represents performance on pure observation whereas the GRU architecture represents performance with sequential context represented in the hidden state. In section 4, we compare the performance of these two model types.

3.3 Training Process

During training, we initially tried to use a vanilla DQN algorithm to solve the environment, with a basic MLP model. We ran a variety of experiments to vary hyperparameters such as learning rate, MLP layer and number of hidden nodes per layer, replay buffer size, exploration rate (epsilon), and epsilon decay rate. Despite this, the simple DQN algorithm never converged and acted randomly. Thus, we moved on to using the DoubleDQN algorithm, which along with more hyperparameter tuning and many more training steps, proved much more stable and was able to exploit a random adversary. For most of the ensuing experiments, we trained with the following hyperparameters:

α	ϵ	ϵ decay steps	Replay size	Hidden layers	Hidden size
8.5e-5	0.15	1e6	1e5 to 3e5	1	128

For each model, we trained against a fully random adversary, a rule-based adversary (`GinRummyNoviceRuleAgent`), and a mixture of the two, by alternating between the random and rule-based policy during training on a per-episode basis. Furthermore, we found that convergence times benefited when we modified the environment configuration to allow knocking at any deadwood during training (still evaluating an environment with normal rules), creating a stronger reward signal during training.

To train the GRU model, we used a replay buffer and sampled batches of full episodes from the buffer, computing target Q-values for each step of each episode, also applying the regular DoubleDQN training algorithm.

Training against different opponents addresses our third research question of the importance of the adversarial policy. We investigate the efficacy of a strategy and test if the agent is even able to learn a winning strategy. In section 4, we inter-compare the performance of these models when trained against different adversaries.

4 Experiments, Results, and Discussion

4.1 RNN models

Despite trying many hyperparameter configurations, none of the GRU models achieved better-than-random performance against a random training adversary. We're unsure whether there was an implementation mistake, we didn't find the right hyperparameters, or RNNs are too unstable to be applied to this environment successfully. Figure 2 displays the training curves. Hence, in the following results, the GRU models are not included.

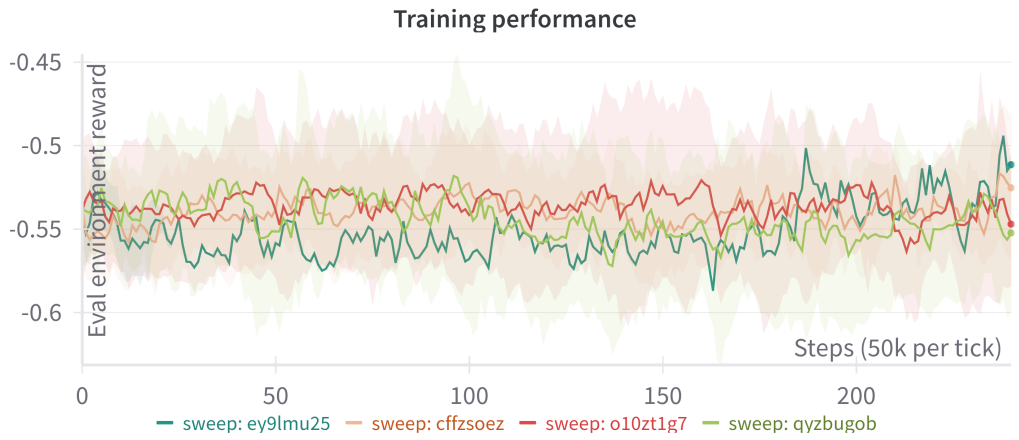

Figure 2: Training performance of several hyperparameter sweeps of GRU models. Shading: standard deviation.

4.1.1 Simplified Observation Representation

Firstly, we examine the performance of models trained on the simplified `ModifiedGinRummyEnv` observation space. The trained agents are $\text{Simple}_{\text{RAND}}$, $\text{Simple}_{\text{MIX}}$, and $\text{Simple}_{\text{RULE}}$, which were trained against random, mixed (random and rule-based), and rule-based adversaries, respectively. See Table 1 for a reward matrix generated by evaluating different pairs of agents against each other. For each trained agent, checkpoints from several runs were used, such that each checkpoint for a given agent was played against each checkpoint of the adversary agents 200 times, after which the results were aggregated and averaged.

Model	Adversary				
	$\text{Simple}_{\text{RAND}}$	$\text{Simple}_{\text{MIX}}$	$\text{Simple}_{\text{RULE}}$	RAND	RULE
$\text{Simple}_{\text{RAND}}$	-0.22 ± 0.11	-0.18 ± 0.11	-0.09 ± 0.13	0.17 ± 0.05	-0.25 ± 0.35
$\text{Simple}_{\text{MIX}}$	-0.19 ± 0.14	-0.16 ± 0.13	-0.12 ± 0.21	0.13 ± 0.07	-0.24 ± 0.39
$\text{Simple}_{\text{RULE}}$	-0.40 ± 0.07	-0.39 ± 0.09	-0.45 ± 0.03	-0.17 ± 0.84	-0.40 ± 0.14
RAND	-0.55 ± 0.03	-0.57 ± 0.03	-0.58 ± 0.28	-0.52	-0.59
RULE	**0.09** ± 0.17	**0.10** ± 0.18	**0.14** ± 0.10	**0.19**	**-0.02**

Table 1: Reward matrix between trained and algorithmic agents

This indicates the trained models never overcome the `GinRummyNoviceRuleAgent` rule-based model, even when directly trained on it. The best-trained model, overall, is the $\text{Simple}_{\text{RAND}}$. This is likely because it was the fastest to converge during training since the random adversary is the weakest adversary of the three, and it learned to exploit it almost as well as the rule-based model.

Generally, however, the policies learned against the random adversary did not hold up well against non-random adversaries. The $\text{Simple}_{\text{MIX}}$ may have improved performance slightly against the rule model, however not significantly. Additionally, the $\text{Simple}_{\text{RULE}}$ struggled to converge, harming its end performance. The 95% confidence interval for the scores is very high when trained models are played against adversaries that they were never trained on, indicating that the performance is high-variance in these cases. The $\text{Simple}_{\text{RAND}}$ and $\text{Simple}_{\text{MIX}}$ evidently played fairly consistently against the random agent, and less so against the rule-based agent. Meanwhile the $\text{Simple}_{\text{RULE}}$ model played more consistently, although poorly, against the rule agent, and much less consistently against the random agent. This again outlines the sensitivity of the learned policies to their adversary.

Of course, this is all based on the average rewards, rather than actual win rates. At times, due to dead hands, the rewards for both agents in a game can be negative. Thus, by treating the agent with the highest reward in a game as the winner, we can generate a similar table of win rates. In Table 2, the table was generated from games between each pair of models, where each model has several checkpoints from different runs. Each configuration was evaluated over 500 games.

Model	Adversary				
	$\text{Simple}_{\text{RAND}}$	$\text{Simple}_{\text{MIX}}$	$\text{Simple}_{\text{RULE}}$	RAND	RULE
$\text{Simple}_{\text{RAND}}$	50%	53%	74%	**98%**	25%
$\text{Simple}_{\text{MIX}}$	47%	50%	72%	**98%**	24%
$\text{Simple}_{\text{RULE}}$	23%	28%	49%	83%	15%
RAND	1%	3%	15%	46%	0%
RULE	**73%**	**75%**	**85%**	**98%**	**48%**

Table 2: Win matrix between trained and algorithmic agents. Rounded to nearest 1%. Ties ignored.

Despite the low average rewards of some match-ups in Table 1, the trained models won against the rule-based model in a significant portion of games. This result supports the previous by also indicating that $\text{Simple}_{\text{RAND}}$ is the dominant trained model.

Additionally, we can study the average lengths of the episodes which generate the results of Table 1.

Model	Adversary				
	$\text{Simple}_{\text{RAND}}$	$\text{Simple}_{\text{MIX}}$	$\text{Simple}_{\text{RULE}}$	RAND	RULE
$\text{Simple}_{\text{RAND}}$	54	48	53	36	26
$\text{Simple}_{\text{MIX}}$	48	46	57	42	29
$\text{Simple}_{\text{RULE}}$	50	56	85	67	28
RAND	36	41	67	59	23
RULE	27	28	29	25	23

Table 3: Average episode lengths. The 95% confidence interval for all values is ± 2 turns.

Here, all games involving the rule-based agent terminate comparatively fast, at around 25 turns in the environment which corresponds to around 12-13 turns in the game. Considering that all the trained agents appear to play to a much higher number of turns, sometimes almost twice as many, this indicates that learned agents have some strategy. However, they are still too slow to compete with the rule-based agent.

By observing the performance of each trained model as they were trained, we estimate that they would've benefited from longer training times. The $\text{Simple}_{\text{MIX}}$ and $\text{Simple}_{\text{RULE}}$ models clearly were steadily improving throughout the training runs, and likely would've continued to do so were they longer. The $\text{Simple}_{\text{RAND}}$ model behaved similarly, although it appeared to be stabilizing more by the end of training and generally learned faster. This can be seen in Figure 3.

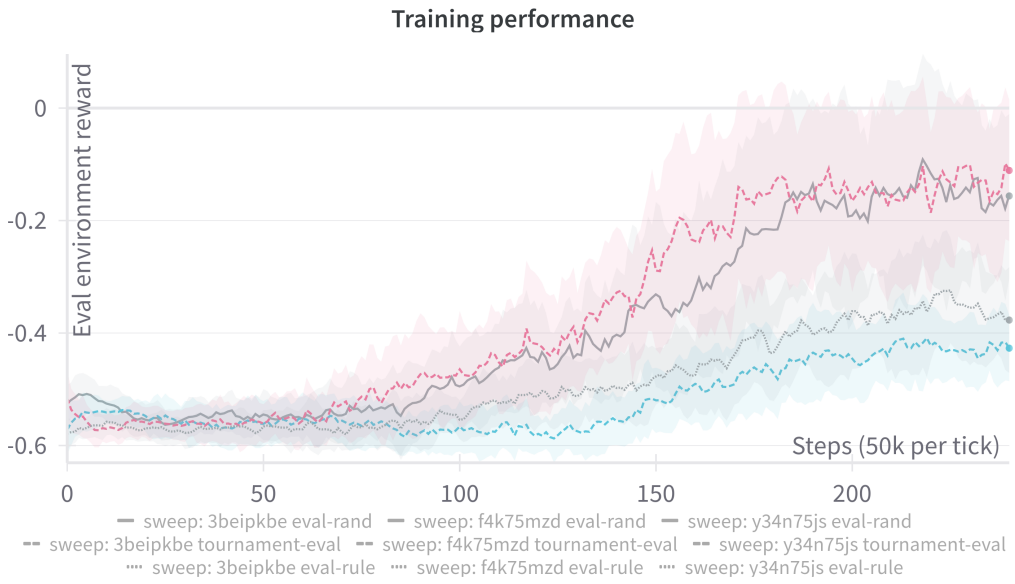

Figure 3: Training performance of $\text{Simple}_{\text{RAND}}$ (pink) against random adversary, $\text{Simple}_{\text{RULE}}$ (blue) against rule-based adversary, and $\text{Simple}_{\text{MIX}}$ (grey) against random (solid) and rule-based (dotted) adversaries. Shading: standard deviation.

4.1.2 Rich Observation Representation

Next, we examine the performance of models trained on the standard RLCard Gin Rummy observation space. $\text{Rich}_{\text{RAND}}$, Rich_{MIX}, and $\text{Rich}_{\text{RULE}}$ were all models on the rich observation features, trained against random, a mixture of random and rule-based, and rule-based adversaries respectively, as in the previous section. Training models with this observation space are significantly less stable than with the simpler featurization. We succeeded in training the $\text{Rich}_{\text{RAND}}$ model to a similar, in fact slightly better performance to the $\text{Simple}_{\text{RAND}}$ model. However neither the Rich_{MIX} or $\text{Rich}_{\text{RULE}}$ models did as well as their *simple* counterparts. Due to an oversight during training, there are no checkpoints for the Rich_{MIX} or $\text{Rich}_{\text{RULE}}$, and hence they cannot be cross-compared with the other models as done before. However, we can still consult their training curves (Figure 4) which indicate rewards against several adversaries.

It's possible due to the increase in dimensionality of the observation space from "only" 156 to 260, the environment became more difficult to solve, despite more information available to the model. We again observed, except for the $\text{Rich}_{\text{RULE}}$ model, that the performance of the trained model continued to increase throughout training, which indicates that a longer training process likely would increase the model performance. Even the $\text{Rich}_{\text{RAND}}$ model consistently showed little signs of improvement until suddenly improving in the last two million steps.

4.2 Self-play

We also experimented with self-play, to compare it to other training techniques. The methodology here was to train as usual, except to copy the network to a static adversary model every 100k steps. We did not experiment with bootstrapping the self-play with an already semi-trained model. In this case, we opted for the simple observation space for its stability, hence the model is named $\text{Simple}_{\text{SELF}}$. See Figure 5 for the training graph.

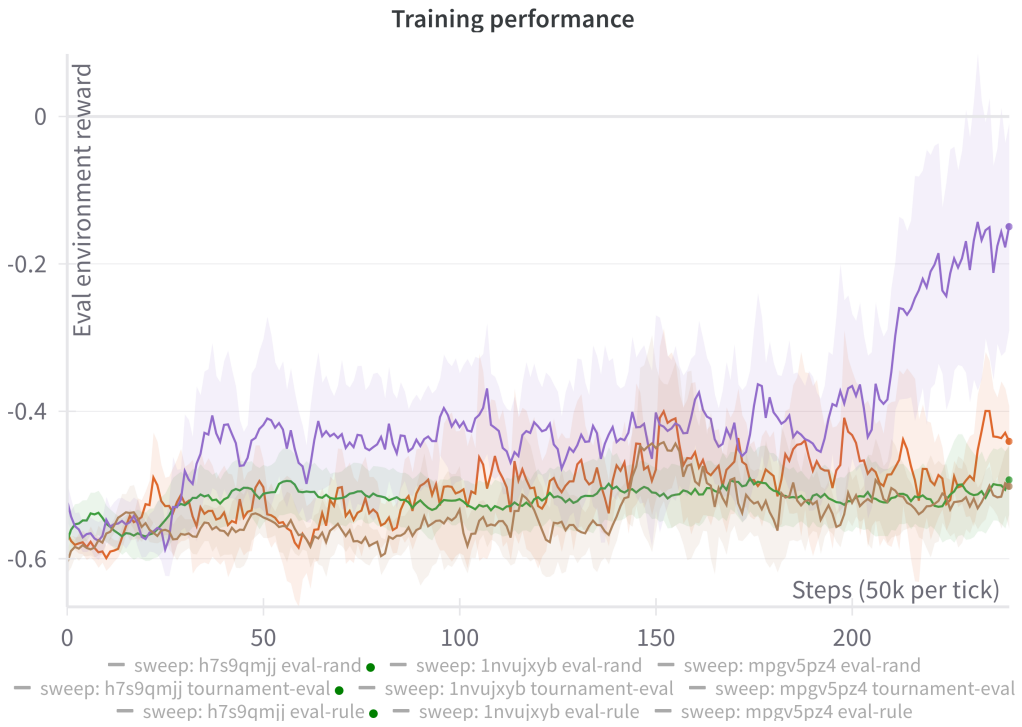

Figure 4: Training performance of $\text{Rich}_{\text{RAND}}$ (purple) against random adversary, $\text{Rich}_{\text{RULE}}$ (green) against rule-based adversary, and Rich_{MIX} against random (orange) and rule-based (brown) adversaries. Shading: standard deviation.

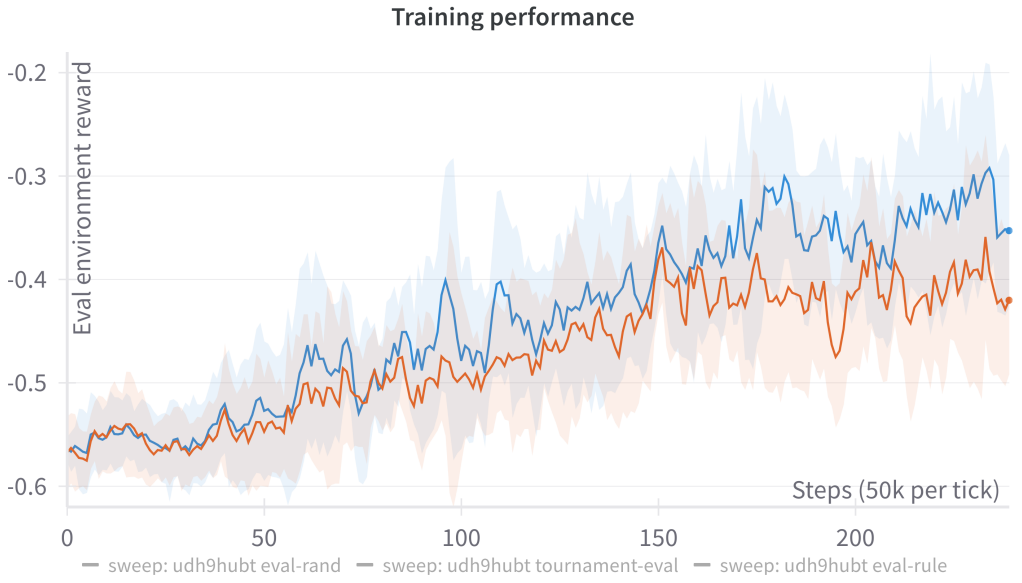

Figure 5: Training performance of $\text{Simple}_{\text{SELF}}$ against itself (orange) and random agent (blue). Shading: standard deviation.

Again, we can only conclude that the model performance was consistently increasing, and likely would benefit from a longer training run. This model displayed unexpected training stability since we were expecting that swapping the adversary model so frequently would be fitting to a moving target, which is difficult. The rate of improvement is somewhat better than $\text{Simple}_{\text{RULE}}$'s, however, indicating that for this domain, an adversary that slowly grows in capability along with the model may benefit training stability and performance. We include the performance against a random agent to highlight that the model isn't simply learning to exploit its adversary after every adversary update but instead learns general strategies that still work against other agents.

5 Conclusion and Future Work

5.1 Summary

Based on our findings, Gin Rummy is a promising candidate for the successful application of deep Q-learning. Furthermore, we showed that the adversary used during training has a large impact on the convergence rate and stability of learned models, and that self-play may also be a viable method of training models. Additionally, the Gin Rummy environment is difficult to learn a model for, particularly an RNN model, and the richer observation space presents unexpected challenges to learning compared to the simplified space.

Moreover, concerning the research goals posed in section 1, we successfully addressed points 1) and 2) by training non-sequence models against several adversaries and modifying the environment to provide multiple observation representations. In addition, we laid the groundwork for future work with recurrent models (point 3), although we were unable to show their efficacy at reasoning over multiple timesteps. Finally, for 4), we were able to experiment and get started with an implementation that utilized self-play, though our results are currently inconclusive due to the unfinalized training.

5.2 Limitations

Our main limitation was our computational capacity. Of our group members, we each had access to one consumer-grade CPU and GPU each. When we consider that running 12 million environment steps (one training run) on our machines took an average of 10 hours, generating sufficient data was extremely time-consuming. We observed, as noted in section 4, that longer training runs would have been beneficial to the performance of most agents, especially those with more interesting ramifications such as the models trained using self-play. Unfortunately, this was something we could not pursue with our computational capabilities. For example, it may have been the case that eventually, with the hyperparameters we used, the RNN model would've converged, however, we didn't have the opportunity to test this.

The RLCard environment itself is also limited in its setup. It only awards knocking and not the difference in deadwood between players – so there is nothing that tells the agent whether it is better to knock with less or more deadwood, for example. Additionally, there is no award for knocking with lower deadwood, undercutting (knocking when the opponent has less deadwood), or being punished by the difference in deadwood when an opponent knocks. Instead, rewards are solely based on an individual player's deadwood and whether they knocked. This is generally not faithful to the actual rules of Gin Rummy and may have made reward information less related to the game.

5.3 Future Work

A large bottleneck in training wasn't just our computational resources but additionally the RLCard library's relatively inefficient implementation of Gin Rummy, given that it's implemented entirely in Python. Instead, OpenSpiel is an open-source reinforcement learning environment library, including many games such as Gin Rummy. It is written in C++ and includes a corresponding Python package, PySpiel, so that OpenSpiel can run in Python. Since OpenSpiel is written in C++, we expect significant speed improvements that would allow much faster training – though we have not directly compared this. We did investigate using PySpiel when running experiments, however, the documentation relating to the observation space was not clear enough for us to determine exactly what each feature corresponded to, or how to modify the environment. More research into the library would likely help to overcome these challenges.

Using GPU clusters could drastically reduce our model training times, allowing for more extensive hyperparameter tuning. By combining this with a faster environment simulator such as OpenSpiel, we feel that significantly stronger results could result from a repetition of the same experiments that we executed with longer training runs. With access to more computing, a more comprehensive hyperparameter and architecture search could also be executed for the RNN model, possibly resulting in better convergence.

An analysis was executed by Kotnik et al. (2003), which we wished to conduct to evaluate the quality of the learned agents' policies, examined whether policies had any suit or card preferences. An optimal player should have no suit preference and should prefer lower-value cards. The authors identified that the TD-Rummy model valued specific suits over others, and had a relatively uniform card value preference. It's possible that similar issues explain the shortcomings in our experiments as well, and should be investigated [5].

References

[1] Bicycle Cards. *How to Play Gin Rummy*. Accessed 2024. URL: https://bicyclecards.com/how-to-play/gin-rummy.

[2] *Games in RLCard; RLCard 1.0.7 documentation — rlcard.org*. https://rlcard.org/games.html#gin-rummy. 2024.

[3] Hado van Hasselt, Arthur Guez, and David Silver. *Deep Reinforcement Learning with Double Q-learning*. 2015. arXiv: 1509.06461 [cs.LG].

[4] Wenzhen Huang et al. "Learning Macromanagement in Starcraft by Deep Reinforcement Learning". In: *Sensors* 21 (May 2021), p. 3332. DOI: 10.3390/s21103332.

[5] Clifford Kotnik and Jugal Kalita. "The significance of temporal-difference learning in self-play training TD-rummy versus EVO-rummy". In: *Proceedings of the Twentieth International Conference on International Conference on Machine Learning*. ICML'03. Washington, DC, USA: AAAI Press, 2003, pp. 369–375. ISBN: 1577351894.

[6] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning*. 2013. arXiv: 1312.5602 [cs.LG].

[7] Viet Dung Nguyen, Dung Doan, and Todd W. Neller. "A Deterministic Neural Network Approach to Playing Gin Rummy". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 35.17 (May 2021), pp. 15622–15629. ISSN: 2159-5399. DOI: 10.1609/aaai.v35i17.17840. URL: http://dx.doi.org/10.1609/aaai.v35i17.17840.

[8] Kun Shao et al. "A Survey of Deep Reinforcement Learning in Video Games". In: *CoRR* abs/1912.10944 (2019). arXiv: 1912.10944. URL: http://arxiv.org/abs/1912.10944.

[9] Xiao Chuan ZHANG and Yi Li. "A Texas Hold'em decision model based on Reinforcement Learning". In: *2020 Chinese Control And Decision Conference (CCDC)*. 2020, pp. 3814–3817. DOI: 10.1109/CCDC49329.2020.9164345.