

BACHELOR'S THESIS COMPUTING SCIENCE



RADBOUD UNIVERSITY NIJMEGEN

Evaluating Reinforcement Learning Strategies in the Card Game Gin Rummy: A Comparative Study of DQN, CDQN, and Reward Shaping Approaches

Author:
Michel van Wijk
s1059491

First supervisor/assessor:
Prof. Dr. Nils Jansen

Second assessor:
Dr. Tal Kachman

March 18, 2024

Abstract

This thesis explores the application of Reinforcement Learning (RL) techniques, specifically Deep Q-Networks (DQN), Constrained DQN (CDQN), and Reward Shaping in DQN (DQN-rs), in the game of Gin Rummy. It investigates the development of efficient strategies using these methods, with a particular focus on constrained reinforcement learning. This offers a framework that integrates additional constraints into the learning process to guide agents towards not only maximizing rewards but also adhering to specific game-related constraints. In a strategy-based game like Gin Rummy, this can significantly influence success. The findings reveal that while the standard DQN agent gradually improved, especially when the epsilon-greedy strategy was omitted, the CDQN agent underperformed in effectively concluding games. In contrast, DQN-rs agents displayed superior performance, achieving high proficiency within a short training span of 500 episodes. A comparative tournament among different agents, including a rule-based agent, further validated these insights, with DQN-rs emerging as the most proficient.

Contents

1	Introduction	2
2	Preliminaries	4
2.1	Reinforcement learning	4
2.2	Constrained reinforcement learning	5
2.3	Reinforcement Learning Methods	6
2.3.1	Q-learning	6
2.3.2	Deep Q-Network	7
3	Environment and Training Methods	8
3.1	Game rules	8
3.2	CMDP	9
3.3	Constrained Deep Q-network (CDQN)	13
3.4	Reward shaping	14
4	Results	15
4.1	Preliminaries	15
4.2	DQN	15
4.3	Constrained DQN	17
4.4	Reward shaping	19
4.5	Decision making example	20
4.6	Tournament	21
5	Related Work	23
6	Conclusions	25
A	Appendix	30
A.1	CMDP Realistic Version	30
A.2	Computer Specifications	33

Chapter 1

Introduction

Reinforcement Learning (RL) represents a significant area within Artificial Intelligence (AI), focused on enabling agents to learn decision-making through interaction with their environment. This field has applications across a diverse range of areas, including finance [7, 14] and strategy games such as chess [23] and Go [22]. It also extends to complex video games, including Dota 2 [20], Starcraft II [24, 13], and classic Atari titles [9, 16], as well as imperfect information card games like DouDizhu [30] and poker [6, 18]. These games are often used as they provide challenging environments that require a mix of strategy, decision-making, and adaptability.

Within RL, we have a subset called Constrained RL (CRL) where the learning process is bound by predefined constraints. These constraints introduce additional goals or limitations that the agent must consider, often reflecting real-world complexities. This approach can be useful in fields like autonomous driving [8, 12] and robotics [27, 4] where safety is a critical consideration. CRL can then help to, for example, avoid collisions during the training process.

Within this domain, the focus will be on a card game called Gin Rummy. The main question guiding this research is: How can effective and efficient strategies for (constrained) reinforcement learning in Gin Rummy be developed? In this game, each player aims to meld their hand of ten cards into sets and runs. A set here is three (or four) of a kind, and a run is a straight flush with at least three cards. The challenge lies in achieving a low 'deadwood' score, which is the value of cards not in any meld. An ace counts for one point, a face card for ten points, and all others follow their numeric value. Players must strategically draw and discard cards with the ultimate goal of knocking or going gin. Knocking can be done once your hand contains ten or less deadwood, and going gin requires all your cards to be in a meld.

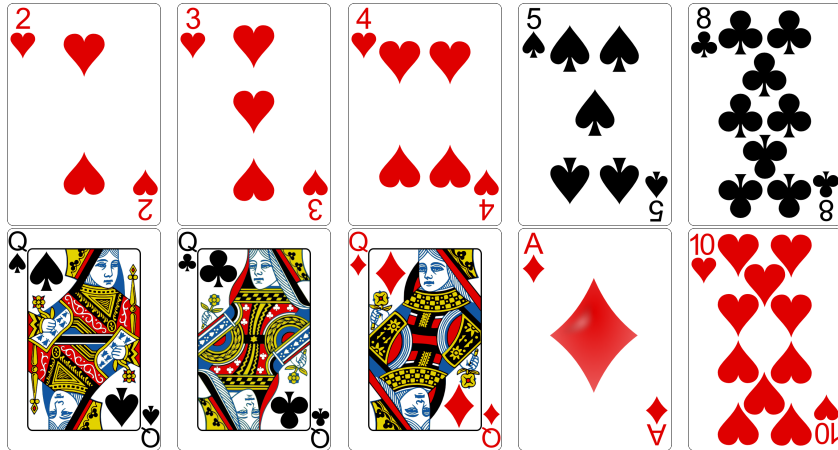


Figure 1.1: Hand in Gin Rummy containing a run and set

This thesis aims to train agents on a simplified version of Gin Rummy using a Deep Q-Network (DQN) [17], a RL algorithm, based on RLCard's implementation [29]. This approach can then be enhanced by incorporating elements from a constrained DQN approach [10], exploring the effectiveness of different constraint levels and reward-shaping strategies in the game.

The following chapters are structured as follows: first, some preliminary information will be introduced. This is followed by a more formal definition of the game and delving into the methods used. After that, the results of the experiments will be presented, comparing the performance of the trained agents. Finally, a chapter on related work and a chapter where the conclusions are discussed as well as some potential topics for future work.

Chapter 2

Preliminaries

2.1 Reinforcement learning

RL is a field of machine learning where agents learn to make decisions sequentially within an environment to learn a policy π , a set of rules that guide the agent's actions [25]. This interaction can be seen in Figure 2.1. The agent observes a state s from the set of possible states S and performs an action a from the set of possible actions A . The environment then responds to the action by presenting a new state to the agent and providing a reward r , which is a scalar value in the set of real numbers \mathbb{R} . The agent's main objective is to maximize the expected cumulative reward over time, steadily progressing toward an optimal policy.

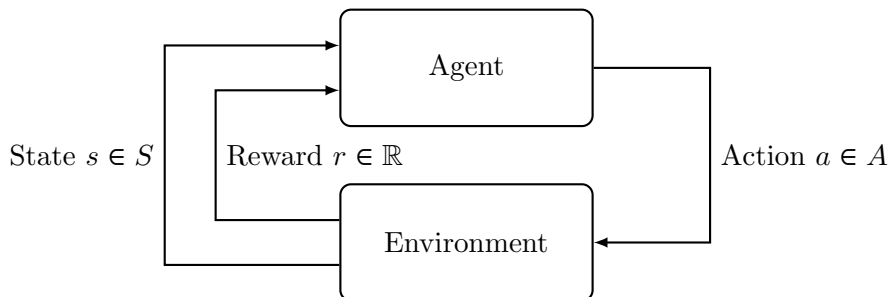


Figure 2.1: The agent-environment interaction in RL.

Let us define some RL concepts more formally:

- The **policy** π followed by the agent is a mapping from states to a probability distribution over the possible actions in that state: $\pi : S \rightarrow \text{Dist}(A)$. The notation $\mathbf{Dist}(A)$ represents the set of all possible probability distributions over the action space A .
- The **value function** $V^\pi(s)$ quantifies the expected return of being in a state s and following policy π thereafter. It is defined as $V^\pi(s) =$

$\mathbb{E}[\sum_{t=0}^{\infty} \gamma^t R_{t+1} | S_t = s]$ where:

- R_{t+1} denotes the immediate reward received after transitioning to a new state from state s at time t by following policy π .
 - γ is the discount factor, a scalar within the interval $[0, 1]$. A higher γ values future rewards more compared to immediate rewards. A value of 0 would mean that we only care about immediate rewards, while a value of 1 means that future rewards are as important as immediate ones.
- The **action-value function** $Q^\pi(s, a)$ extends the concept of the value function by evaluating the expected return of taking an action a in state s and following policy π thereafter. It is defined as $Q^\pi(s, a) = \mathbb{E}[\sum_{t=0}^{\infty} \gamma^t R_{t+1} | S_t = s, A_t = a]$. The action-value function thus assesses the value of performing a specific action in a given state, taking into account the policy's future actions.

An RL **environment** is modelled as a Markov Decision Process (MDP) [21], a mathematical framework encapsulating the dynamics of decision-making scenarios. MDPs are formally defined by a 5-tuple (S, A, T, R, γ) where:

- **State Space** (S): A set, either finite or infinite, representing the various states in which the decision-maker might find themselves.
- **Action Space** (A): A set, also finite or infinite, of actions available to the decision-maker. For each state $s \in S$, there exists a subset $A_s \subseteq A$ indicating the actions available in the state s .
- **Transition Function** (T): A function $T : S \times A \rightarrow \text{Dist}(S)$ that defines the probability distribution over states resulting from taking an action $a \in A_s$ in state s .
- **Reward Function** (R): A function $R : S \times A \rightarrow \mathbb{R}$ that assigns a real number as the immediate reward received for taking an action $a \in A_s$ in state s . This function quantifies the benefit associated with each action in each state.
- **Discount Factor** (γ): As previously defined, this scalar within the interval $[0, 1]$ indicates the agent's preference for immediate versus future rewards.

2.2 Constrained reinforcement learning

A challenge in RL is the exploration-exploitation dilemma. In the pursuit of optimal decision-making, an RL agent must find a balance between exploring new actions, including those that may be suboptimal and exploiting

known high-reward actions. This balance is essential for efficient learning.

However, certain practical scenarios demand even more control over the agent’s behaviour. In some cases, there are actions in specific states that should never be taken because they could lead to unsafe or undesirable outcomes. This is where Constrained RL (CRL) becomes useful. For instance, consider a robot that has to navigate a grid world. You could then constrain the robot to never fall off this grid. CRL enables us to enforce these kinds of restrictions on the agent’s actions, ensuring safety and desired behaviour.

To implement constraints within the RL framework, the standard MDP can be augmented with a cost function, transforming it into a Constrained MDP (CMDP) [2]. This cost function, denoted as C , is defined as follows: $C : S \times A \rightarrow \mathbb{R}$. It associates a cost with each state-action pair, allowing us to encode and enforce constraints by penalizing undesirable actions in specific states. This penalization encourages the agent to make decisions that align with the defined safety criteria, thus ensuring the desired level of control and constraint in the RL process.

The objective then becomes to find a policy π that maximizes the expected cumulative reward while ensuring that the cumulative cost does not exceed predefined limits. Specifically, the agent seeks to solve the following optimization problem:

$$\max \mathbb{E}_{\pi} \left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t) \right]$$

subject to

$$\mathbb{E} \left[\sum_{t=0}^{\infty} \gamma^t C(s_t, a_t) \right] \leq d$$

where:

- $C(s_t, a_t)$ is the cost received for taking action a in state s at time t .
- \mathbb{E}_{π} represents the expected value under policy π and in this case, the expected cumulative reward or cost.
- d is the predefined threshold that the cumulative cost must not exceed.

2.3 Reinforcement Learning Methods

2.3.1 Q-learning

At the core of Q-learning [28] is the concept of the Q-value function (also known as the action-value function defined earlier), denoted as $Q(s, a)$.

The ultimate goal of Q-learning is to learn the optimal Q-value function, $Q^*(s, a)$, which guides the agent to the best action in each state. Q-learning updates its estimates of Q-values towards the optimal Q-values using the Bellman optimality equation. The update rule applied after each action taken in the environment is as follows:

$$Q_{\text{new}}(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right]$$

Here, α is the learning rate ($0 < \alpha \leq 1$), which determines the extent to which new information overrides old information. The term $R_{t+1} + \gamma \max_a Q(s_{t+1}, a)$ represents the reward for the current action plus the discounted value of the future reward, obtained by taking the best possible next action. The difference between this term and the current estimate, $Q(s_t, a_t)$, is the temporal difference error, which is used to update the Q-value.

The objective of Q-learning is to iteratively update the Q-values for each state-action pair in such a way that they converge to the optimal Q-values, $Q^*(s, a)$. This would then give the optimal policy by always taking the action that maximizes the Q-value in each state.

Epsilon-greedy policy

When picking actions in Q-learning, an epsilon greedy approach can be used to balance exploration and exploitation. With this, a value $\epsilon \in [0, 1]$ can be chosen and used to pick the action with the highest value (exploitation) with probability $1 - \epsilon$ and a random action with probability ϵ (exploration).

2.3.2 Deep Q-Network

In large or continuous state spaces, traditional Q-learning faces scalability challenges due to the impracticality of maintaining a comprehensive state-action value table. To address this, the Deep Q-Network (DQN) [17] algorithm employs a neural network to approximate the Q-function. The approximated Q-value for a state-action pair when using neural network parameters is denoted as $Q(s, a; \theta)$, where θ represents the weights of the neural network. To improve stability, DQN utilizes a separate target network, yielding Q-values $Q'(s, a; \theta^-)$ with weights θ^- that are updated less frequently than the weights of the primary network. This separation of parameters between the primary network (θ) and the target network (θ^-) helps to mitigate the issues of correlations between the Q-values and the target values.

Chapter 3

Environment and Training Methods

3.1 Game rules

Before defining the CMDP, the game's rules are discussed more in-depth, starting with a reiteration of some key game concepts. A card is in a meld if it is in a set, three (or four) of a kind, or in a run, a straight flush with at least three cards. Each card not in a meld is called deadwood and contributes to your deadwood score, which is to be minimized. An ace contributes one point, a face card ten points, and all other cards contribute their numerical value.

At the start of the game, two players are each dealt ten cards. One card is put face up on the discard pile and all other remaining cards are put face down on the stockpile. After all cards have been dealt, the game proceeds through a series of turns. On each turn, a player must:

- **Draw:** the player begins their turn by taking a card. They can choose to draw the top card from either the stockpile or the discard pile. Choosing from the discard pile gives the player a known card that might fit into their hand for a potential meld, but also reveals some information about their strategy to the opponent. Taking from the stockpile gives a random card but keeps the player's strategy more concealed.
- **Discard:** the player ends their turn by discarding a card from their hand face up onto the discard pile. The discard is an important tactical decision because it can either be a card that is least useful to the player's hand or a card they believe will be least beneficial to the opponent. You can also choose to play for potential future gain by keeping a card with higher value, but with good connectivity in your hand.

The game can end in one of three ways:

- **Knocking:** a player may end the game on their turn if, after discarding, the total value of their unmatched cards, those not in a meld, is 10 points or less.
- **Gin:** a player can end the game by achieving 'Gin', which occurs when all ten cards are used in melds without any unmatched cards.
- **Dead hand:** if a player does not wish to draw the top card of the discard pile and the stockpile is also empty, the round is declared dead.

As a simplified version of the game with just a single round is being discussed here, there is no need for keeping score. The focus is on the final action taken, after which the reward function defined in the subsequent chapter will give a value that indicates how well the agent did that round.

3.2 CMDP

The CMDP for Gin Rummy according to the RLCard [29] implementation will be defined here.

In the context of our model, we utilize the notation \mathbb{B}^{52} to represent a specific vector space. The symbol \mathbb{B} denotes a binary domain, where each element can take on one of two possible values: 0 or 1. Therefore, \mathbb{B}^{52} refers to a 52-dimensional vector space, where each vector is composed of 52 binary elements. This is used here to represent a standard deck of playing cards.

- S: A state is represented as a tuple comprising:
 - $h \in \mathbb{B}^{52}$: This is the hand of the current player denoted as a 52-element vector where each element represents a card in the deck. If a card is present in the hand, the corresponding element is 1; otherwise, it is 0. The order of the vector follows the sequence: $(A\spadesuit, 2\spadesuit, \dots, K\spadesuit, A\heartsuit, \dots, K\heartsuit, A\diamondsuit, \dots, K\diamondsuit, A\clubsuit, \dots, K\clubsuit)$.
 - $tc \in \mathbb{B}^{52}$: The top card of the discard pile.
 - $p \in \mathbb{B}^{52}$: The cards in the discard pile excluding the top card.
 - $o \in \mathbb{B}^{52}$: The known cards in the opponent's hand (obtained from the discard pile).
 - $u \in \mathbb{B}^{52}$: The unknown cards. These are either in the stockpile or in the unknown part of the opponent's hand.

Thus a state s is a tuple (h, tc, p, o, u) and our state space $S = \{(h, tc, p, o, u) \mid h, tc, p, o, u \in \mathbb{B}^{52}\}$.

It is hard to compute an exact number for the size of this state space due to the constraints in the game. The size cannot be simply calculated as $(2^{52})^5$ because each card's presence in one part of the state (e.g., the player's hand h) inherently means its absence in all others (tc, p, o, u), enforcing a unique distribution of 1s across the five components for any valid game state. There are also constraints on how many cards can be in each component. For example, tc will always only contain one card. These constraints significantly reduce the theoretical maximum size of the state space.

At the start of the game, we know that h contains 10 cards and there are $\binom{52}{10}$ ways to choose 10 cards out of 52 for a player's starting hand. We then have 42 cards left to choose from for our discard pile starting card. All other components are then also defined since p and o start off empty and u contains all other unused cards. This would give a number of $\binom{52}{10} \cdot 42 \approx 6.64 \cdot 10^{11}$ starting configurations. As cards are moved from the stockpile to the player's hands or from the player's hand to the discard pile, the number of possible configurations increases. From this, it is clear that the state space size is vast, making exhaustive methods impractical to solve this directly.

- A: The set of possible actions consists of the following:

Action ID	Action
0	Score player 0: used after knock or gin of player 1, or dead hand to compute the player's score. This action terminates the game.
1	Score player 1: used after knock or gin of player 0, or dead hand to compute the player's score. This action terminates the game.
2	Draw a card: The player draws a card from the deck.
3	Pick top card: The player picks the top card from the discard pile.
4	Declare dead hand : The player declares the hand dead.
5	Gin : The player goes gin.
6-57	Discard a card. 6: $A\spadesuit$, 7: $2\spadesuit$, ..., 18: $K\spadesuit$, 19: $A\heartsuit$, ..., 31: $K\heartsuit$, 32: $A\diamondsuit$, ..., 44: $A\diamondsuit$, 45: $A\clubsuit$, ..., 57: $K\clubsuit$
58-109	Knock . 58: $A\spadesuit$, 59: $2\spadesuit$, ..., 70 : $K\spadesuit$, 71: $A\heartsuit$, ..., 83: $K\heartsuit$, 84: $A\diamondsuit$, ..., 96: $A\diamondsuit$, 97: $A\clubsuit$, ..., 109: $K\clubsuit$

So our action space $A = \{0, 1, \dots, 109\}$

- P: The transition function. Indicator functions are used to see if a card index is in a certain part of the state. For instance, it can be used to see if a card is in the deck of some state s : $\mathbb{1}_u(x) = \begin{cases} 1, & \text{if } x \in u \\ 0, & \text{if } x \notin u \end{cases}$

This gives the following transitions:

$$P(s, 2, s') = \begin{cases} \frac{1}{\|u\|}, & \text{if } \exists! x \in \{0, 1, \dots, 51\} \mathbb{1}_u(x) = 1 \wedge \mathbb{1}_{u'}(x) = 0 \\ \wedge \mathbb{1}_{h'}(x) = 1 \\ 0, & \text{otherwise.} \end{cases}$$

$$P(s, a, s') = \begin{cases} 1, & \text{if } a \in A_s \setminus \{2\}, \text{ and performing } a \text{ in } s \text{ leads to } s'. \\ 0, & \text{otherwise.} \end{cases}$$

Note that the probabilities for drawing a card do not match reality, as there is no access to the opponent's cards here. A similar CMDP version that mimics real life can be found in the appendix.

- R: The reward function. The ultimate goal in Gin Rummy is to go gin with knocking as a good alternative. Otherwise, the reward is determined by the amount of deadwood left in a player's hand. This gives us the following reward function:

$$R(s, a) = \begin{cases} \frac{-\#deadwood}{100} & \text{if } a \in \{0, 4\} \\ 0.5, & \text{if } a \in \{58, 59, \dots, 109\} \\ 1.0, & \text{if } a = 5 \\ 0, & \text{otherwise} \end{cases}$$

- C: the cost function. A function $d : \mathbb{B}^{52} \rightarrow \mathbb{N}$ is defined which computes the deadwood in a player's hand according to the rules of Gin Rummy. With this, the cost function can be defined:

$$C(s, a) = \begin{cases} 1, & \text{if } a \in \{6, 7, \dots, 57\} \wedge |\{a' \in A_s | a' \neq a \wedge d(T(s, a).h) > \\ d(T(s, a').h)\}| \geq 3 \\ 1, & \text{if } a = 2 \wedge d(s.h \text{ OR } s.tc) < d(s.h) \\ 0, & \text{otherwise} \end{cases}$$

Now, to make these constraints more clear, there is a hand in Figure 3.1 as well as a stockpile with a visible discard pile card in Figure 3.2. There are ten cards in our hand, so there is a decision between picking up the visible card and a random card from the stockpile. In this case, the visible card

is a queen and would give us a set of queens. This means that the second constraint comes into play as the deadwood of our hand with this card (28) is lower than our current deadwood (48). Ignoring this and picking up the card from the stockpile anyway would thus give us a penalty.

Assuming that the queen was picked up, it is now time to discard a card, and the first constraint matters. It says to avoid discarding a card if at least three other discard options lead to a lower deadwood value. This aims to prevent for example discarding a queen that is now in a set or the five, six, or seven of spades that are in a run. In this case, that would mean that the best discards are either the ten of clubs, nine of hearts, or the six of hearts.

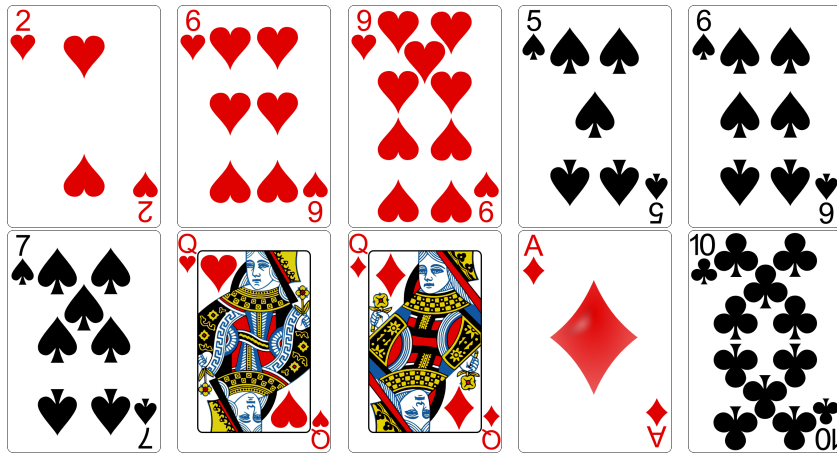


Figure 3.1: Player's hand

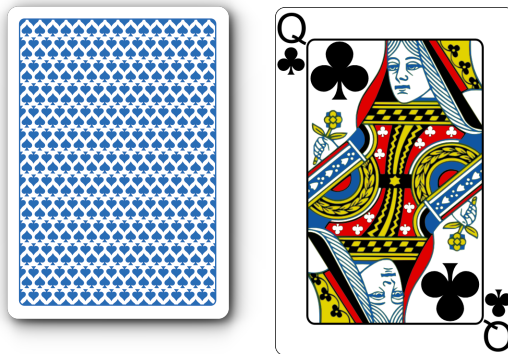


Figure 3.2: Stockpile and discard pile

3.3 Constrained Deep Q-network (CDQN)

In this section, the implementation of the Constrained Deep Q-network (CDQN) is discussed, adapted from the Deep Constrained Q-learning paper by Kalweit et al. [10]. The standard 'DQNAgent' from RLCard [29] can be extended to develop our 'CDQNAgent', incorporating specific constraints in the learning process.

The CDQN algorithm modifies the standard Deep Q-Network (DQN) framework by introducing safe action computation and safe set iteration. Here we have the pseudocode of the algorithm:

Algorithm 1 CDQN Algorithm Pseudocode [10]

```

1: Initialize Q-network  $Q$ , target network  $Q'$ 
2: Initialize experience replay buffer  $R$  to store transition tuples  $(s, a, r, s')$ 
3: for each optimization step do
4:   Sample a mini-batch of  $n$  transitions  $(s_i, a_i, s_{i+1}, r_i)_{1 \leq i \leq n}$  from  $R$ 
5:   Calculate the set of safe actions  $S_C(s_{i+1})_{1 \leq i \leq n}$ 
6:   Set  $y_i = r_i + \gamma \max_{a \in S_C(s_{i+1})} Q'(s_{i+1}, a | \theta^{Q'})$ 
7:   Minimize MSE of  $y_i$  and  $Q(s_i, a_i | \theta^Q)$ 
8:   Periodically update the target network  $Q'$ 
9: end for
10: for each execution step do
11:   Observe the current state  $s_t$  from the environment
12:   Calculate the set of safe actions  $S_C(s_t)$  for the current state
13:   Apply  $a_t = \arg \max_{a \in S_C(s_t)} Q(s_t, a)$  to obtain the best safe action
14: end for

```

Let us go through the steps in a bit more detail:

- We start by initializing the Q-network Q and the target Q-network Q' . These networks are used to estimate the expected rewards for taking certain actions in particular states. The target network's weights are updated less frequently to stabilize training.
- The experience replay buffer R is also initialized, which will store tuples of the form (s, a, r, s') , where s is the current state, a is the action taken, r is the reward received, and s' is the next state.
- The optimization loop is then entered:
 1. A minibatch of n transitions is sampled from the replay buffer. This allows the agent to learn from previous experiences and reduces the correlation between experiences.

2. For each next state s_{i+1} in the minibatch, the set of safe actions $S_C(s_{i+1})$ is calculated. These are the actions that are allowed under the constraints specified by the problem; this is what distinguishes CDQN from standard DQN. We can define this function more formally as $S_C(s_{i+1}) = \{a | a \in A_s \wedge C(s, a) = 0\}$.
 3. The target value y_i for each sample in the minibatch is then set. It is the immediate reward r_i plus the discounted reward of the best safe action in the next state as predicted by the target network Q' .
 4. The Mean Squared Error (MSE) between the target values y_i and the values predicted by the Q-network for the actual actions taken a_i is minimized. This is the learning step where the weights of the Q-network are adjusted. The MSE can be defined here as $\frac{1}{n} \sum_{i=1}^n (y_i - Q(s_i, a_i | \theta^Q))^2$.
 5. Periodically, the weights of the target network Q' are updated to slowly track the learned Q-network, which helps in stabilizing the learning process.
- In the execution step, the current state is observed, and the best safe action is selected.

3.4 Reward shaping

Building upon the constraints defined earlier reward shaping can be explored to refine the agent’s learning process. Reward shaping involves modifying the reward function, and integrating domain knowledge to provide more immediate and informative feedback on the agent’s actions. In our case, penalties (negative rewards) are introduced for actions that violate our constraints. This approach addresses the issue of the original reward function’s sparsity, which primarily provides feedback at the game’s conclusion. By integrating penalties for specific undesirable actions, the agent can be guided through training more effectively, encouraging it to learn strategies that adhere to both the game’s objectives and our defined constraints.

Chapter 4

Results

4.1 Preliminaries

In this section, the performance of the agents will be evaluated. The specifications of the computer on which all code was run can be found in the appendix. All code used can be found on '<https://github.com/Michelwv/ThesisGinRummyRL>'. The following hyperparameters were used during training:

- Replay memory size: 20000.
- Discount factor: 0.99.
- MLP layers: [64, 64]; neural network with two hidden layers, each with 64 neurons.
- Learning rate: 0.00005.

An epsilon-greedy approach with $\epsilon = 0.1$ was also used. All results during training were obtained from playing a tournament of 10000 games every 100 games against a random agent, a benchmark to gauge the agent's learning progress.

4.2 DQN

The first training was done using a standard DQN implementation, without using any constraints. The outcomes of this initial phase are depicted in Figure 4.1. To provide context, a random agent typically achieves an average reward in the range of -0.5 to -0.6 . Surprisingly, our agent's performance was similar to that of a random agent even after 10,000 episodes. Although some success was observed, most games concluded with no cards left to draw and a high deadwood count. Initially, the cause seemed to be suboptimal hyperparameters, but experimenting with alternative configurations failed

to show any noticeable improvements.

Interestingly, eliminating the epsilon-greedy strategy resulted in improved performance, as illustrated in Figure 4.2. The agent showed a similar starting trajectory but demonstrated significant improvement after 5,000 episodes. Reinforcement learning is often unstable and that can be seen here with a substantial regression around the 12,500-episode mark, almost going back to the initial performance. Fortunately, it did go back to its peak close to 0.4. Extended training beyond 20,000 episodes revealed similar spikes without achieving further improvement.

The reasons behind the considerable performance difference between Figure 4.1 and Figure 4.2 remain unclear. Though there is randomness in the game with drawing cards thus possibly eliminating the need for an epsilon-greedy strategy, this big of a gap was unexpected.

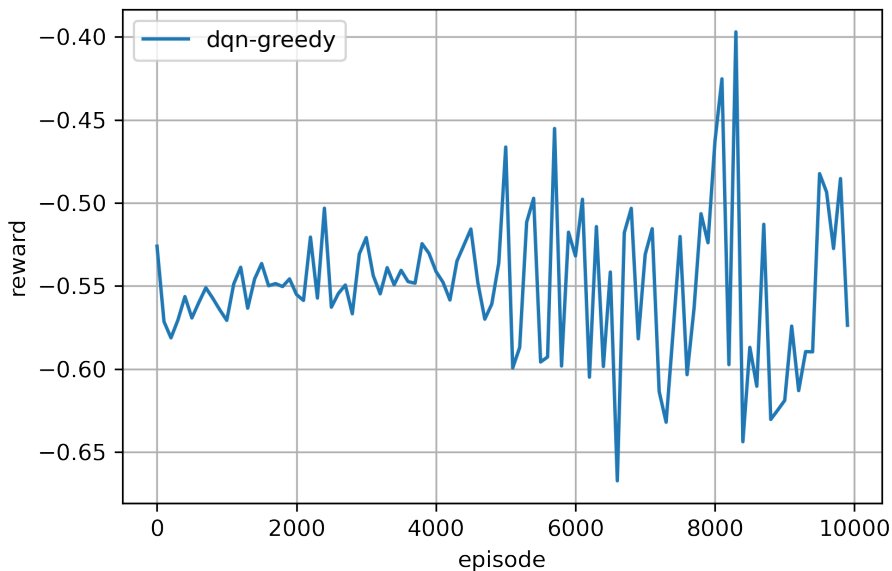


Figure 4.1: Results DQN training with an epsilon-greedy policy

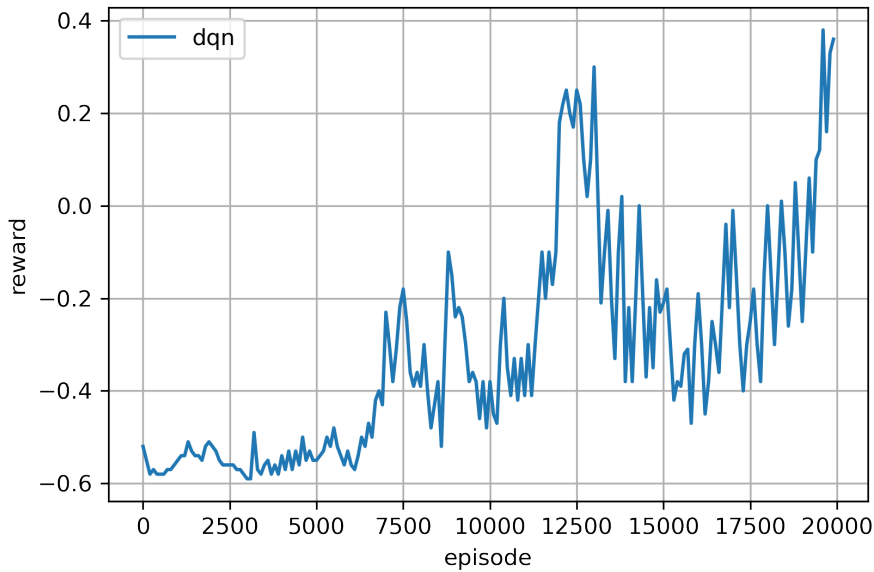


Figure 4.2: Results DQN training without an epsilon-greedy policy

4.3 Constrained DQN

This section will go into the outcomes of training with CDQN. Unlike the previous approach with DQN, the decision against implementing an epsilon-greedy strategy from the beginning was made here. Different constraint levels were used during training, which is indicated in Figure 4.3 as 'cdqn-n'. Here, 'n' represents the number of discard options the constraints allow. In our most restrictive setting, labelled 'cdqn-1', the agent could only choose the optimal discard option, creating a highly constrained environment. Interestingly, as depicted in Figure 4.3, this approach was counterproductive, with the agent's performance dropping to -0.6—worse than a random agent. A slight improvement was observed after 15,000 episodes, but it was not substantial.

In a surprising turn, the 'cdqn-5' setup, which permitted a wider range of discard choices, recorded the highest peak in performance. However, it instantly went back down again, and stabilized at a level similar to 'cdqn-3'. Notably, none of the CDQN variants could surpass the performance of the DQN agent shown in Figure 4.2. All CDQN models remained below the zero-reward threshold, a stark contrast to the 0.4 peak achieved by the standard DQN.

Determining the exact reasons for this underwhelming performance is challenging. The original paper by Kalweit et al. [10] reported significant success using CDQN in a different application. One possible explanation for our results is the larger state space in our case, suggesting that the earlier findings might not be directly applicable to more complex environments. Further research, applying CDQN in various settings, would be necessary to confirm this hypothesis. Another potential factor could be the nature of the constraints themselves; however, this seems less likely since a strategy based on these constraints should theoretically return better results than what was observed. Experimenting with varying learning rates and neural network architectures was attempted as well, as the optimal parameters for CDQN might differ from those for DQN. Nonetheless, the outcomes presented in Figure 4.3 were the best from the experiments.

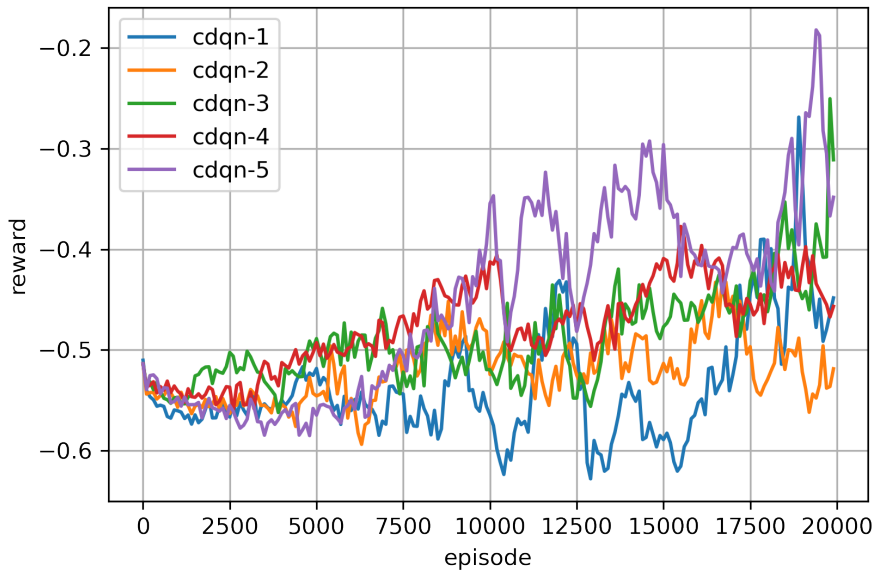


Figure 4.3: Results during training by CDQN agent

4.4 Reward shaping

In our exploration of reward shaping, several agents with penalties for non-optimal actions were trained, denoted as 'rs-n', where 'n' mirrors the constraint setting used in the CDQN approach. Unlike CDQN, which avoids penalties by adhering to constraints, these DQN-based agents were subjected to a significant penalty of -2 for each non-constrained move, determined to be optimal after preliminary trials with values ranging from -0.1 to -5. The penalty magnitude's main effect was the speed of reaching a reward above 0.4. As depicted in Figure 4.4, the reward-shaping agents achieved a level of performance comparable to what the standard DQN attained after approximately 20,000 episodes, but remarkably, they did so within a mere 500 episodes.

The more restricted agents showed more inconsistencies during training with 'rs-1' and 'rs-2' going below 0.4 a few times, but always maintaining a solid performance. Despite this, the difference in peak performance between the agents was minimal, suggesting that even with varying degrees of constraint, reward shaping provides a quick and solid convergence to effective strategies.

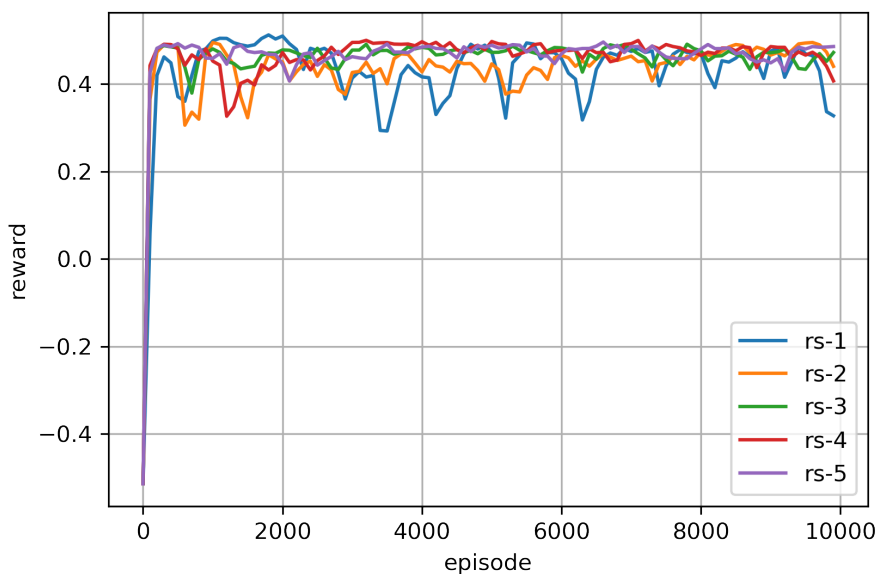


Figure 4.4: Results during training by DQN-rs agent

4.5 Decision making example

Before continuing with the final tournament results, let us analyze two situations covered by the constraints and see how the agents handle them. For the first situation, the decision between taking a random card and taking the visible card is looked at. As shown in Figures 4.5 and 4.6, the player must decide whether to pick up the visible king of diamonds or draw a random card. The optimal decision here is to pick up the king of diamonds, as it gives us a set of kings and directly decreases our deadwood by 20. Fortunately, all agents (DQN, CDQN, and DQN-rs) also come to this conclusion. Having picked up the king, a card now has to be discarded and put on the discard pile. For this, the agents all come to different conclusions:

- DQN: it decides to discard the king of clubs undoing our progress immediately. This is alongside discarding the other kings the worst decision you can make here. The only slight positive here is that the action was not a clear winner. The q-value for discarding the 7 of hearts, which is still not optimal, was within one-tenth.
- CDQN: this agent somehow manages to take an even worse action in discarding the king of diamonds, which was just picked up. This action was also clearly favoured by the agent as none of the others came close to being picked.
- DQN-rs: displaying better judgment, this agent discards the ten of diamonds. It is still not optimal as getting rid of the queen of clubs would result in slightly lower deadwood, but a significant improvement compared to the previous two agents.

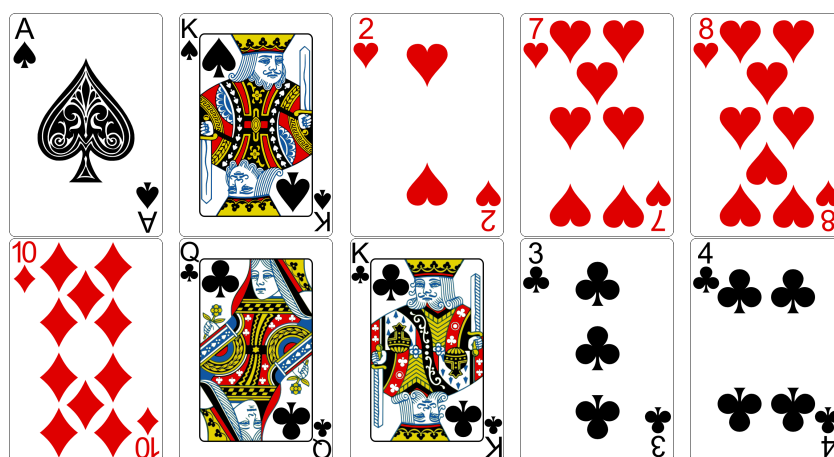


Figure 4.5: Player's hand

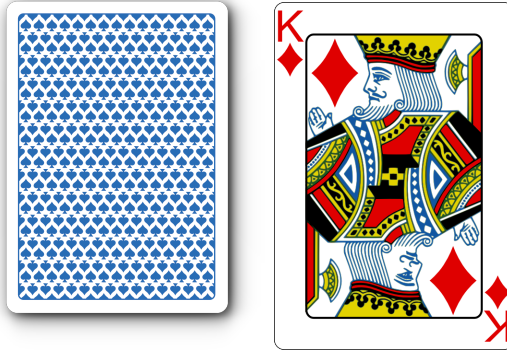


Figure 4.6: Stockpile and discard pile

4.6 Tournament

Having trained and tested the different agent implementations, a tournament was organized featuring DQN, CDQN ('cdqn-3'), DQN-rs ('rs-3'), a random agent, and a rule-based agent. The rule-based agent employs a strategy of knocking or going gin whenever possible and discards the card leading to the lowest deadwood, resorting to random actions otherwise.

All results were obtained by playing 10000 games against each other. Win-rates can be found in Table 4.1, unsafe moves made per game in Table 4.2, the rate of ending a game by knocking or going gin in Table 4.3, and the average number of moves a game in Table 4.4. Note that a move here is just a decision taken i.e. discarding a card is a move just like picking up a card is a move.

The results are mostly as expected with only some slight surprises coming from the CDQN agent. Table 4.3 shows that it rarely ends a game by knocking or going gin and all of its wins instead come from having the lowest amount of deadwood when no more cards are left in the deck. This aligns with its high number of unsafe moves (Table 4.2), indicating a less effective discarding strategy, as also observed in the previous section's hand example. However, CDQN achieved a respectable win rate against DQN, suggesting that its strategy, while not optimal for ending games, still holds some merit in a competitive setup.

Apart from this, these results confirm what was previously seen during training. From our trained agents, DQN-rs emerged as the strongest contender even outperforming the rule-based agent in their direct matchup. Though DQN-rs did make more unsafe moves in their match, coming from discards, it gains an advantage in drawing the correct cards.

Table 4.1: Tournament Results: winrates

	DQN	DQN-rs	CDQN	Random	Rule-based
DQN	-	14.2	65.1	98.3	10.8
DQN-rs	85.8	-	89.1	99.3	53.1
CDQN	34.9	10.9	-	63.2	0.9
Random	1.7	0.7	36.8	-	0.8
Rule-based	89.2	46.9	99.1	99.2	-

Table 4.2: Tournament Results: unsafe moves % per game

	DQN	DQN-rs	CDQN	Random	Rule-based
DQN	-	8.6	14	5.7	7.9
DQN-rs	2.7	-	5.7	2.5	3.4
CDQN	16.4	18.6	-	14.9	16.3
Random	19.4	19.3	19.6	-	19.3
Rule-based	0.62	1.2	0.8	1.7	-

Table 4.3: Tournament Results: rate of ending a game by knocking/going gin

	DQN	DQN-rs	CDQN	Random	Rule-based
DQN	-	2.9	20.5	89	8.5
DQN-rs	33.5	-	45	98.4	52.6
CDQN	0.2	0.3	-	22.7	0.3
Random	0.9	0.8	1.45	-	0.75
Rule-based	90	46.8	98.8	98.5	-

Table 4.4: Tournament Results: average number of moves a game

	DQN	DQN-rs	CDQN	Random	Rule-based
DQN	-	78	172	81	74
DQN-rs	78	-	127	47	48
CDQN	172	127	-	135	70
Random	81	47	135	-	52
Rule-based	74	48	70	52	-

Chapter 5

Related Work

Though Gin Rummy has been around since the early 20th century, there is limited literature surrounding it. More has been done in other games, both involving perfect and imperfect information. Examples of perfect information games include work done in chess and go with AlphaGo [22] and AlphaZero [23] where they use Monte Carlo tree search, deep neural networks, and reinforcement learning to obtain superhuman performance. As for imperfect information games, poker is often used in research as the benchmark, more specifically the Heads-up no-limit Texas Hold'em (NLHU) variant. DeepStack [18] and Libratus [3] are two HUNL AIs with DeepStack being the first to beat human professionals.

Inspired by success in Backgammon [26], one work that focuses on Gin Rummy by Kotnik and Kalita in 2003 [11] uses temporal-difference (TD) learning and co-evolution to train agents and explore their performance. Their results show that the evolved players outperformed the TD players as well as all players beating a random agent 100% of the time, though it is all over a very small sample of 10 games. They also struggled with long training times and finding optimal hyperparameters, a common problem in reinforcement learning.

Almost two decades later in 2021, an EAAI research challenge by Gettysburg College to develop the best Gin Rummy agent was created [19]. A total of 14 papers were submitted of which 13 were accepted after peer review. They also held a tournament at the end of the challenge between all agents to determine the best-performing one. The winner [1] came from an agent using a hand strength estimation model for discarding alongside counterfactual regret minimization (CFR) [31] to find a knocking strategy. As for picking up a card from the discard pile or the remaining deck, they used a simple strategy matching the one defined in our constraints. A few other entries focus specifically on estimating the opponent's hand with tech-

niques ranging from using random forests [5] and using deep neural networks alongside heuristic strategies [15]. They then use this data to improve their existing agent.

Chapter 6

Conclusions

This thesis presented an exploration of different RL strategies applied to the game of Gin Rummy, focusing on the standard DQN, CDQN, and DQN-rs. The standard DQN agent showed improvement over time, particularly when the epsilon-greedy strategy was removed. This finding suggests that in certain stochastic environments like card games, traditional exploration techniques may not always be the most effective.

Our CDQN agent did not achieve the expected and desired results. Despite adjustments to the constraints and hyperparameters, it failed to outperform standard DQN, struggling to end games by knocking or going gin. Conversely, the DQN-rs agents achieved a high level of performance and did so in only 500 episodes. This goes against the findings of the CDQN authors where CDQN outperformed reward shaping. It is unclear whether this is due to suboptimal hyperparameters or constraints, or something in the algorithm itself. The tournament between different agents further confirmed these findings, with DQN-rs emerging as the most proficient, even managing to beat the rule-based agent, followed by the standard DQN and CDQN.

Directions for future work include exploring different RL techniques such as PPO and A3C, applying these strategies to other games or domains, refining reward shaping and constraint methods, and seeing how it all compares. Seeing how CDQN performs in different environments could be especially interesting considering the difference between our results and the CDQN author's results.

Bibliography

- [1] Aqib Ahmed, Joshua Leppo, Michal Lesniewski, Riken Patel, Jonathan Perez, and Jeremy Blum. A heuristic evaluation function for hand strength estimation in gin rummy. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(17):15465–15471, May 2021.
- [2] Eitan Altman. *Constrained Markov Decision Processes*, volume 7. CRC Press, 1999.
- [3] Noam Brown and Tuomas Sandholm. Safe and nested subgame solving for imperfect-information games, 2017.
- [4] Lukas Brunke, Melissa Greeff, Adam W. Hall, Zhaocong Yuan, Siqi Zhou, Jacopo Panerati, and Angela P. Schoellig. Safe learning in robotics: From learning-based control to safe reinforcement learning, 2021.
- [5] Anthony Hein, May Jiang, Vydhourie Thiyageswaran, and Michael Guerzhoy. Random forests for opponent hand estimation in gin rummy. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(17):15545–15550, May 2021.
- [6] Johannes Heinrich and David Silver. Deep reinforcement learning from self-play in imperfect-information games, 2016.
- [7] Yuh-Jong Hu and Shang-Jen Lin. Deep reinforcement learning for optimizing finance portfolio management. In *2019 Amity International Conference on Artificial Intelligence (AICAI)*, pages 14–20, 2019.
- [8] David Isele, Alireza Nakhaei, and Kikuo Fujimura. Safe reinforcement learning on autonomous vehicles, 2019.
- [9] Lukasz Kaiser, Mohammad Babaeizadeh, Piotr Milos, Blazej Osinski, Roy H Campbell, Konrad Czechowski, Dumitru Erhan, Chelsea Finn, Piotr Kozakowski, Sergey Levine, Afroz Mohiuddin, Ryan Sepassi, George Tucker, and Henryk Michalewski. Model-based reinforcement learning for atari, 2020.

- [10] Gabriel Kalweit, Maria Huegle, Moritz Werling, and Joschka Boedecker. Deep constrained q-learning, 2020.
- [11] Clifford Kotnik and Jugal Kalita. The significance of temporal-difference learning in self-play training td-rummy versus evo-rummy. pages 369–375, 01 2003.
- [12] Hanna Krasowski, Xiao Wang, and Matthias Althoff. Safe reinforcement learning for autonomous lane changing using set-based prediction. 07 2020.
- [13] Dennis Lee, Haoran Tang, Jeffrey O Zhang, Huazhe Xu, Trevor Darrell, and Pieter Abbeel. Modular architecture for starcraft ii with deep reinforcement learning, 2018.
- [14] Xiao-Yang Liu, Hongyang Yang, Jiechao Gao, and Christina Dan Wang. Finrl: deep reinforcement learning framework to automate trading in quantitative finance. In *Proceedings of the Second ACM International Conference on AI in Finance, ICAIF '21*, New York, NY, USA, 2022. Association for Computing Machinery.
- [15] Bhaskar Mishra and Ashish Aggarwal. Opponent hand estimation in gin rummy using deep neural networks and heuristic strategies. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(17):15607–15613, May 2021.
- [16] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.
- [17] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dhharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, February 2015.
- [18] Matej Moravčík, Martin Schmid, Neil Burch, Viliam Lisý, Dustin Morrill, Nolan Bard, Trevor Davis, Kevin Waugh, Michael Johanson, and Michael Bowling. Deepstack: Expert-level artificial intelligence in heads-up no-limit poker. *Science*, 356(6337):508–513, May 2017.
- [19] T. Neller. Gin rummy eaaai undergraduate research challenge, 2021. Accessed on: 29-11-2023.

- [20] OpenAI, :, Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemyslaw Debiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique P. d. O. Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang. Dota 2 with large scale deep reinforcement learning, 2019.
- [21] Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.
- [22] David Silver, Aja Huang, Christopher Maddison, Arthur Guez, Laurent Sifre, George Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–489, 01 2016.
- [23] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dhharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm, 2017.
- [24] Peng Sun, Xinghai Sun, Lei Han, Jiechao Xiong, Qing Wang, Bo Li, Yang Zheng, Ji Liu, Yongsheng Liu, Han Liu, and Tong Zhang. Tstarbots: Defeating the cheating level builtin ai in starcraft ii in the full game, 2018.
- [25] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.
- [26] G. Tesauro. Temporal difference learning and tdgammon. In *Communications of the ACM* 38(3), pages 58–68, 1995.
- [27] Brijen Thananjeyan, Ashwin Balakrishna, Suraj Nair, Michael Luo, Krishnan Srinivasan, Minh Hwang, Joseph E. Gonzalez, Julian Ibarz, Chelsea Finn, and Ken Goldberg. Recovery rl: Safe reinforcement learning with learned recovery zones. *IEEE Robotics and Automation Letters*, 6(3):4915–4922, 2021.
- [28] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3):279–292, 1992.
- [29] Daochen Zha, Kwei-Herng Lai, Songyi Huang, Yuanpu Cao, Keerthana Reddy, Juan Vargas, Alex Nguyen, Ruzhe Wei, Junyu Guo, and Xia

- Hu. Rlcard: A platform for reinforcement learning in card games. In *IJCAI*, 2020.
- [30] Daochen Zha, Jingru Xie, Wenye Ma, Sheng Zhang, Xiangru Lian, Xia Hu, and Ji Liu. Douzero: Mastering doudizhu with self-play deep reinforcement learning. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 12333–12344. PMLR, 18–24 Jul 2021.
- [31] Martin Zinkevich, Michael Johanson, Michael Bowling, and Carmelo Piccione. Regret minimization in games with incomplete information. In J. Platt, D. Koller, Y. Singer, and S. Roweis, editors, *Advances in Neural Information Processing Systems*, volume 20. Curran Associates, Inc., 2007.

Appendix A

Appendix

A.1 CMDP Realistic Version

- S: We define a state as a tuple of the following items:
 - $h_1 \in \mathbb{B}^{52}$: This is the hand of player 1 denoted as a 52-element vector where each element represents a card in the deck. If a card is present in the hand, the corresponding element is 1; otherwise, it is 0. The order of the vector follows the sequence: $(A\spadesuit, 2\spadesuit, \dots, K\spadesuit, A\heartsuit, \dots, K\heartsuit, A\diamondsuit, \dots, K\diamondsuit, A\clubsuit, \dots, K\clubsuit)$.
 - $h_2 \in \mathbb{B}^{52}$: The hand of player 2.
 - $tc \in \mathbb{B}^{52}$: The top card of the discard pile.
 - $p \in \mathbb{B}^{52}$: The cards in the discard pile excluding the top card.
 - $d \in \mathbb{B}^{52}$: This is the remaining deck from which cards can be drawn.
 - $t \in \mathbb{B}^3$: a variable indicating the state of the turn. The first element indicates whose turn it is: 0 for player 0 and 1 for player 1. The second element gives information about the part of the turn we are at: 0 for drawing a card, picking up a card from the discard pile, declaring the hand dead or going gin, and 1 for discarding a card or knocking. The final element gives information about whether scoring is available yet: 0 if it is unavailable and 1 if it is.

Thus a state s is a tuple (h_1, h_2, tc, p, d, t) and our state space $S = \{(h_1, h_2, tc, p, d, t) \mid h_1, h_2, tc, p, d \in \mathbb{B}^{52}, t \in \mathbb{B}^3\}$

- A: The set of possible actions consists of the following:

Action ID	Action
0	Score player 0: used after knock, gin, or dead hand to compute the player's hand.
1	Score player 1: used after knock, gin, or dead hand to compute the player's hand.
2	Draw a card: The player draws a card from the deck.
3	Pick top card: The player picks the top card from the discard pile.
4	Declare dead hand: The player declares the hand dead.
5	Gin: The player goes gin.
6-57	Discard a card. 6: $A\spadesuit$, 7: $2\spadesuit$, ..., 18: $K\spadesuit$, 19: $A\heartsuit$, ..., 31: $K\heartsuit$, 32: $A\diamondsuit$, ..., 44: $A\diamondsuit$, 45: $A\clubsuit$, ..., 57: $K\clubsuit$
58-109	Knock. 58: $A\spadesuit$, 59: $2\spadesuit$, ..., 70: $K\spadesuit$, 71: $A\heartsuit$, ..., 83: $K\heartsuit$, 84: $A\diamondsuit$, ..., 96: $A\diamondsuit$, 97: $A\clubsuit$, ..., 109: $K\clubsuit$

So our action space $A = \{0, 1, \dots, 109\}$

- P: the transition function.

We will use indicator functions to see if a card index is in a certain part of the state. For instance, we can use it to see if a card is in the

$$\text{deck of some state } s: \mathbb{1}_d(x) = \begin{cases} 1, & \text{if } x \in d \\ 0, & \text{if } x \notin d \end{cases}$$

We now have the following transitions:

$$P(s, 2, s') = \begin{cases} \frac{1}{\|d\|}, & \text{if } \exists! x \in \{0, 1, \dots, 51\} \mathbb{1}_d(x) = 1 \wedge \mathbb{1}_{d'}(x) = 0 \wedge \mathbb{1}_{h'_1}(x) = 1 \\ 0, & \text{otherwise.} \end{cases}$$

$$P(s, a, s') = \begin{cases} 1, & \text{if } a \in A_s \setminus \{2\}, \text{ and performing } a \text{ in } s \text{ leads to } s'. \\ 0, & \text{otherwise.} \end{cases}$$

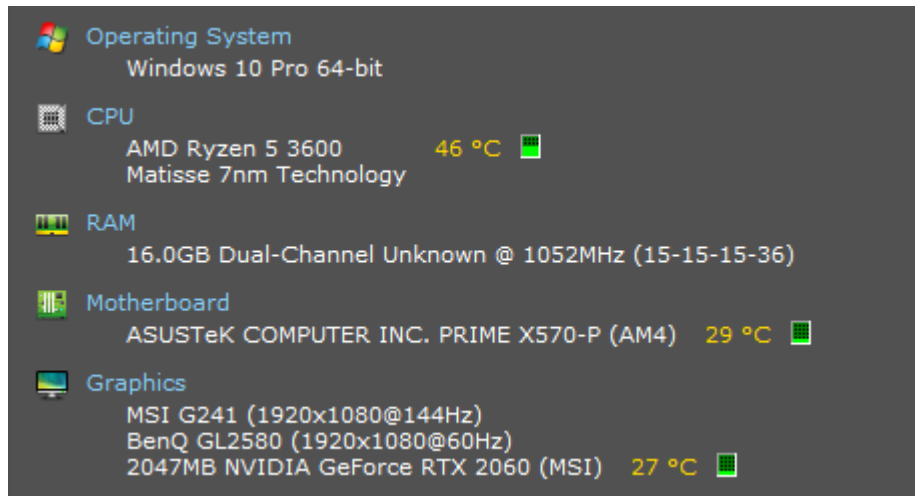
- R: the reward function. The ultimate goal is to go gin with knocking as a good alternative. Otherwise, we determine the reward by the amount of deadwood left. This gives us the following reward function:

$$R(s, a) = \begin{cases} \frac{-\#deadwood}{100} & \text{if } a \in \{0, 4\} \\ 0.5 & \text{if } a \in \{58, 59, \dots, 109\} \\ 1.0 & \text{if } a = 5 \\ 0, & \text{otherwise} \end{cases}$$

- C: the cost function. We define a function $d : \mathbb{B}^{52} \rightarrow \mathbb{N}$ which computes the deadwood in a player's hand according to the rules of Gin Rummy. With this, we define the following cost function:

$$C(s, a) = \begin{cases} 1, & \text{if } a \in \{6, 7, \dots, 57\} \wedge |\{a' \in A_s | a' \neq a \wedge d(T(s, a).h) > \\ & d(T(s, a').h)\}| \geq 3 \\ 1, & \text{if } a = 2 \wedge d(s.h \text{ OR } s.tc) < d(s.h) \\ 0, & \text{otherwise} \end{cases}$$

A.2 Computer Specifications



The image shows a screenshot of the Windows System Information window, displaying the following specifications:

- Operating System:** Windows 10 Pro 64-bit
- CPU:** AMD Ryzen 5 3600, Matisse 7nm Technology, 46 °C
- RAM:** 16.0GB Dual-Channel Unknown @ 1052MHz (15-15-15-36)
- Motherboard:** ASUSTeK COMPUTER INC. PRIME X570-P (AM4), 29 °C
- Graphics:** MSI G241 (1920x1080@144Hz), BenQ GL2580 (1920x1080@60Hz), 2047MB NVIDIA GeForce RTX 2060 (MSI), 27 °C