ETH zürich

SDSC

Reinforcement Learning for the Swiss Card Game Schieber Jass

Implementing a Custom Environment and Training RL Agents for Schieber Jass with Deep Q-Learning

Semester project

Author:
Albin Koshy

Advisors:
Dr. Ilnura Usmanova
Steven Stalder
Dr. Simon Dirmeier

Supervisors:
Prof. Dr. Fanny Yang
Prof. Dr. Fernando Perez-Cruz

December 30, 2024

Abstract

This project investigates the application of reinforcement learning (RL) techniques to the Swiss card game Schieber Jass, with a focus on training deep Q-learning (DQN) and Double DQN (DDQN) agents. A custom Schieber Jass environment was developed, incorporating random, greedy, and RL-based agents, allowing for extensive evaluation through both full and partial information settings. The study demonstrates that both DQN and DDQN agents can learn effective strategies, even in the partial information setting. Key findings include the identification of an optimal set of hyperparameters for agent training. The importance of masking invalid actions to enhance training performance was also highlighted. The results also highlight the potential of RL in competitive card games and suggest future research directions. Overall, this work contributes to the development of RL agents in complex, team-based, partial information settings, with potential implications for creating more advanced and collaborative RL systems in similar domains.

Contents

Acknowledgments

I would like to express my heartfelt gratitude to my advisors Dr. Ilnura Usmanova, Steven Stalder, and Dr. Simon Dirmeier for their invaluable guidance and support throughout this project, particularly during the challenging phases of training the agents. Even though my knowledge of the game of Jass was limited, your assistance was instrumental in navigating both the theoretical and technical aspects of the work. I extend special thanks to Ilnura and Steven for their valuable feedback during the writing phase.

I am also thankful for the working spaces in the Swiss Data Science Center, which provided a productive environment and introduced me to many insightful people.

I would like to thank Prof. Dr. Fanny Yang for allowing me to undertake this project at the Swiss Data Science Center. Special thanks to Prof. Dr. Fernando Perez-Cruz for providing me the opportunity to work on this topic.

Lastly, I deeply thank my family for their unwavering support throughout my studies.

1 Introduction

1.1 Motivation

In recent years, reinforcement learning (RL) has gained significant attention due to its success in solving complex decision-making problems, particularly in games. Games often reflect real-world challenges that require strategic decision-making in dynamic and competitive environments. Many of these situations can be looked at through the lens of game theory, where players make decisions to maximize their outcomes while also considering the actions of others. Such dynamics are also prevalent in many real-world domains like business, economics, and policy-making. Deep reinforcement learning techniques have demonstrated superhuman performance in playing perfect information games – where the entire game state is visible to all players at any given time – such as Go (Silver et al. (2017)) and Atari (Mnih et al. (2015)). However, many real-world problems involve incomplete information, making it necessary to apply reinforcement learning to other games, where only part of the game state is observable. Card games like Poker or Bridge highlight this challenge, as players must somehow estimate their opponents' cards based on the actions they take during the game.

This project aims to explore the application of reinforcement learning in the context of Jass, a popular Swiss card game with imperfect information. Jass is one of the most popular card games in Switzerland, often considered the national card game. It is deeply embedded in Swiss culture, with many different variants across different parts of the country. Jass is a complex game that requires both short-term decisions and long-term strategic planning, making it an ideal testbed for reinforcement learning in imperfect information games. Its intricate set of rules can be challenging for beginners, requiring significant practice to become proficient. Mastery of the game typically takes decades of experience. Moreover, the "Schieber" variant, the most popular Jass variant and the focus of this project, involves team-based play, adding further complexity through the need for cooperation and competition between players.

1.2 Related Work

Numerous projects have focused on developing AI agents for various card games, utilizing RL and related techniques. Notable examples include the application of RL in poker, such as the DeepStack AI in Moravčík et al. (2017) and the Pluribus Poker AI in Brown and Sandholm (2019), where agents have achieved superhuman performance. Other card games, like Big 2, have been explored using RL techniques, as demonstrated by Charlesworth (2018), which implements Proximal Policy Optimization (PPO) agents. Similarly, deep Q-network agents have been used to play simpler card games, as seen in De Jong (2021). For Bridge, a trick-based game similar to Jass, recent work such as Bridge AI by Kita et al. (2024) has been conducted.

In the domain of Jass, prior research has also applied RL methods. For instance, Jaquet and Kutirov (2024) implements RL agents for the Differenzler Jass variant using

the RLCard (Zha et al., 2020) framework. A notable contribution is the master thesis "JassTheRipper" by Niklaus (2019), which trained agents for the Schieber Jass variant using Determinized Monte Carlo Tree Search (DMCTS) based on pre-existing Jass gameplay datasets.

In contrast to these previous works, particularly those involving Schieber Jass, this project trains agents directly through gameplay without relying on pre-existing datasets.

1.3 Project Goals

The goal of this project is first to implement a fully functional Schieber Jass environment, which can be easily modified to support self-play, followed by the development of reinforcement learning agents capable of playing the game effectively. An extensive analysis of the optimal choice of hyperparameters will then be conducted. Two deep Q-learning agents (DQN and DDQN) will be compared, and the performance of these learned agents will be evaluated by comparing them against fixed-strategy baseline opponents and each other (DQN vs. DDQN).

The ultimate goal, though not within the scope of this project, is to develop a superhuman reinforcement learning agent that can consistently outperform expert human players, learning solely through self-play.

1.4 Contribution

This project makes the following contributions. First, we implement a custom Schieber Jass environment tailored for reinforcement learning. Then, we include the implementation of two fixed-strategy agents: a greedy agent, which prioritizes short-term gains, and a random agent, which makes decisions without any strategic considerations. Moreover, we implement two deep Q-learning agents: a DQN and a Double DQN (DDQN) agent. The performance of these agents is evaluated against fixed-strategy opponents and each other. Through these contributions, we aspire to establish a solid foundation for future research, particularly in multi-agent reinforcement learning for Schieber Jass.

2 Background

This chapter will first introduce the Schieber variant of Jass, giving a short overview of its rules and explaining the complexity of the game. It will then provide a brief comparison to other related card games. Next, it will give a quick overview of RL, with a focus on Q-learning.

2.1 Schieber Jass

2.1.1 Rules

Schieber Jass is one of the most popular variants of Jass, played by four players in two teams. The two players on a team must sit directly across from each other, with the opponents sitting adjacent to them. The game uses a deck of 36 playing cards, which can be either in the German (Roses, Shields, Acorns, and Bells) or French (Hearts, Spades, Diamonds, and Clubs) suits. The cards used range from 6 to Ace in a standard poker deck. At the beginning of each round, each player is dealt nine cards, which must be kept hidden from the other players. Each round consists of nine tricks, meaning that four cards are played during each trick. Each round awards a total of 157 points. The goal of each team is to accumulate points by winning tricks, which are determined by the cards played in each trick.

The player who starts the first round is the one sitting to the right of the player who dealt the cards. In each round, the starting player can choose the type of game to play. The starting player can choose from six game types: tops-down ("Obenabe" in Swiss German), bottoms-up ("Undenufe"), or either one of the four suits as trump. Alternatively, the player can pass the decision to their partner ("Schieben"), a feature that gives this Jass variant its name. If that happens, their teammate cannot return the decision to the first player and must choose what to play themselves. After making their choice, the first teammate must start the round.

The starting player begins the first trick by playing their first card, after which the other players, moving counterclockwise, each play one card. Once all players have played a card, the player with the highest card wins the trick and leads the next one. After all cards have been played (a total of nine tricks), the points from each won trick are counted and added to the team's total score. The game continues with the previous starting player shuffling the cards and the player to their right starting the second round, and so on. The game finishes either after a fixed number of rounds divisible by four—to ensure each player has an equal opportunity to choose the game type, commonly 12 rounds in tournaments—or when a predetermined score is reached, typically 1000 or 2500 points.

Game types and card ranking

Here is a short description of the six different game types in Jass. A detailed explanation of the card rankings and points distribution can be found on the official Swiss Lottery website (Swisslos, 2024b):

1. **Tops-Down (Obenabe)**: The cards follow their natural ranking, where Aces are the highest, followed by Kings, Queens, Jacks, and so on, down to the 6s. All suits are treated equally and the highest card wins the trick.

2. **Bottoms-Up (Undenufe)**: This is the opposite of tops-down, where the ranking of the cards is reversed with the 6s being the highest, following the 7s, 8s, and so on. Like tops-down, there are no trump suits.

3. **Trump Suit**: The starting player can choose one of the four suits as the trump suit. All cards of the trump suit outrank all cards from the other suits. The ranking of the trump suit is as follows: the trump Jack is the highest, followed by the trump 9, with the natural card ranking (Ace, King, Queen, and so on) applying to the remaining cards down to the 6. The ranking of the non-trump suits follows the natural ranking.

Regardless of the game type being played, the first card played in a trick determines the leading suit, and all other players must follow suit. If a player does not have a card in the leading suit, they can play any card they want, but they cannot win the trick. When a trump is chosen, cards of the trump suit can be played in any trick, regardless of the leading suit. Figure 2.1 shows an example trick, assuming this round is using Bells as the trump suit.

Figure 2.1: This figure is from Swisslos, 2024b. Assuming that Bells is the trump suit and "Mystery 187" leads the round, "meister_ja589" wins this trick because the Under (Jack) of Bells is stronger than seven of Bells. The other cards do not follow suit and are not considered in determining the trick's winner.

There exist additional rules—multiplicative factors for different suits, melds, marriages, matches, and more—which are left out for simplicity in this project. If interested, they can be read on the Swiss Lottery website (Swisslos, 2024a).

2.1.2 Complexity

The following calculations are taken from Niklaus (2019). Since 36 cards are played in each round, there are $36! \approx 3.72 \times 10^{41}$ possible playouts, as each card is played only once without repetition. The number of valid possible playouts was found empirically to be 5.1×10^{16}. The initial hand distribution results in

$$\frac{36!}{(9!)^4} = \binom{36}{9}\binom{27}{9}\binom{18}{9}\binom{9}{9} \approx 2.145 \times 10^{19}$$

different possibilities. After the cards are dealt and a player knows their hand, the possible distributions of cards among the other players reduce to 2.279×10^{11}. The number of states an algorithm must process is then proportional to:

$$5.1 \times 10^{16} \times 2.279 \times 10^{11} \approx 1.16 \times 10^{28}$$

2.1.3 Similar card games to Jass

Since Jass belongs to the family of trick-taking card games, there are many similar card games to Jass that share the same fundamental principles, such as trump suits, partnerships, and hidden information. Examples include Hearts, Bridge, Belote, Skat, and Twenty-eight. Research on Jass can, therefore, be useful for other trick-taking card games as well.

2.2 Reinforcement Learning

2.2.1 RL Basics

Reinforcement learning (RL) has been used repeatedly to play various kinds of games. RL aims to train agents who act in environments to maximize their long-term cumulative rewards. Unlike supervised learning, where neural networks are trained using input-output pairs, RL agents learn by interacting with the environment and adjusting their actions based on feedback through trial and error. Simply put, as shown in Figure 2.2, the reinforcement learning process works as a continuous loop. The agent observes the current state of the environment and receives a reward based on its previous action. Using this information, it decides on its next action, which then changes the state of the environment. Through this iterative process, the agent learns to make decisions that maximize its rewards.

The environment models an unknown Markov Decision Process (MDP), where it represents the current state and returns a reward for an action the agent took. More formally, the MDP is represented as a tuple $(\mathcal{S}, \mathcal{A}(.), P(s' \mid s, a), r(s, a, s'))$, where $\mathcal{S}$ is a state space, $\mathcal{A}(.)$ is an action space dependent on the state s, $P(s' \mid s, a)$ is a transition operator encoding the transition probabilities between two states s' and s, and $r(s, a, s')$ is a reward function that assigns the immediate reward received after transitioning from state s to state s' by taking action a. An MDP is considered unknown when the transition function $P(s' \mid s, a)$ and the reward function $r(s, a, s')$ are not provided or

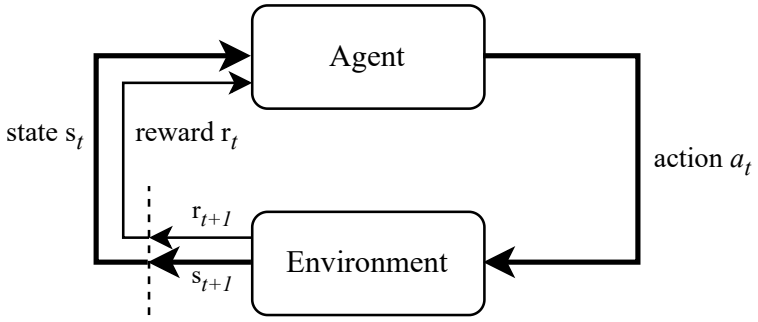

Figure 2.2: Figure describing the agent-environment interaction in RL, where the agent observes the current state, receives a reward, and takes an action that changes the environment's next state.

explicitly defined. For convenience, $r(s, a, s')$ is often simplified to $r(s, a)$, representing the expected immediate reward after taking action a in state s. This is calculated as:

$$r(s, a) = \sum_{s'} r(s, a, s') P(s' \mid s, a)$$

At each time step t, the agent observes a state $s \in \mathcal{S}$, selects an action $a \in \mathcal{A}$ based on a policy $\pi(a|s)$—which models the agent's behavior and can be either probabilistic or deterministic—and receives an immediate reward $r = r(s, a, s')$ from the environment. After taking the action a, the agent observes a new state $s'|s, a$. We denote this sequence of events as the transition tuple (s, a, r, s').

After an action is taken, the environment updates according to the MDP, resulting in a new state s'. Specifically, the transition function $P(s' \mid s, a)$ is evaluated, which defines the probability of transitioning to a new state s' given the previous state s and action a. In deterministic dynamics, this transition function provides a direct mapping with probabilities of 0 or 1. This process continues until a terminal state is reached, if one exists, or until it is stopped after a predetermined number of iterations.

The goal of the agent is to learn an optimal policy that maximizes the expected cumulative reward, which is often represented as the expected value function

$$J(\pi) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r(s_t, \pi(s_t))\right]$$

with γ representing the discount factor. Since the environment is unknown, the agent first needs to explore its environment to find out which actions give him the largest rewards. To solve this credit assignment problem, two RL approaches exist: Model-Based and Model-Free RL. In Model-Based RL we try to learn the MDP, that is, we estimate the transition probabilities $P(s' \mid s, a)$ and the reward function $r(s, a, s')$, and optimize the policy based on the estimated model. By having access to this model, we can simulate the environment and use algorithms like policy iteration and value iteration to determine the optimal policy. However, estimating an entire model in a complex system can be very challenging and computationally very expensive. Therefore, Model-Free RL is often preferred, where we bypass the need for an explicit model and directly estimate the value function or policy. In Model-Free RL, the agent relies on value functions to estimate the expected rewards and guide its actions. The two primary types of value functions are (with fixed policy π):

- State-Value Function: The total expected reward an agent receives in state s if he follows policy π from there on:

$$V^{\pi}(s) = \mathbb{E}\Big[\sum_{t=0}^{\infty} \gamma^t r(s_t, \pi(s_t)) \mid s_0 = s\Big] = r(s, \pi(s)) + \gamma \sum_{s'} P(s' \mid s, \pi(s)) V^{\pi}(s')$$

- Action-Value Function/Q-function: The total expected reward an agent can achieve if he takes action a in state s and from there follows policy π:

$$Q^{\pi}(s, a) = \mathbb{E}\Big[\sum_{t=0}^{\infty} \gamma^t r(s_t, \pi(s_t)) \mid s_0 = s, a_0 = a\Big] = r(s, a) + \gamma \sum_{s'} P(s' \mid s, a) V^{\pi}(s')$$

For an optimal policy, it holds the Bellman equation (for further details, please refer to Sutton and Barto, 2018), where the optimal policy induces the optimal value function, and the optimal value function determines the optimal policy:

$$V^*(s) = \max_a Q^*(s, a)$$

There exist many Model-Free algorithms, which are either used by optimizing the value function (Q-learning, SARSA), policy-gradient methods (REINFORCE, PPO) or a combination of both called actor-critic methods (SAC, DDPG). We will be focusing on one method of Model-Free RL called Q-learning which will be covered in the next section.

2.2.2 Tabular Q-Learning

The core idea behind Q-learning is to approximate the optimal Q-function, which satisfies the Bellman equation. The Q-function represents the expected cumulative future rewards for a given state-action pair. It essentially says, how "good" it is to take an action a being in a given state s. However, we do not have access to the transition probabilities $P(s' \mid s, a)$ of the environment's dynamics, which are required to compute the Q-function directly with the given formula above.

There are two main approaches to solving this problem: one is tabular Q-learning, and the other is deep Q-learning. In tabular Q-learning, the Q-function is represented as a table, where each value corresponds to a state-action pair. These values are called Q-values and represent a real number: $\mathcal{S} \times \mathcal{A} \to \mathbb{R}$. The Q-values for each state-action pair are approximated iteratively. Specifically, after observing a transition tuple (s, a, r, s')—where s is the current state, a is the chosen action, r is the immediate reward, and s' is the resulting next state—the Q-value for the state-action pair $(s, a) \in \mathcal{S} \times \mathcal{A}$ is updated by adding the immediate reward r to the discounted maximum Q-value achievable from the next state s', considering all possible future actions a'. This update rule is expressed as:

$$\hat{Q}^*(s, a) \leftarrow r + \gamma \max_{a'} \hat{Q}^*(s', a') \tag{2.1}$$

However, because this update process has high variance, we use a softer update rule, called **bootstrapping**, which helps reduce this variance. The bootstrapping update rule is expressed as:

$$\hat{Q}^*(s, a) \leftarrow (1 - \alpha_t)\hat{Q}^*(s, a) + \alpha_t[r + \gamma \max_{a'} \hat{Q}^*(s', a')] \tag{2.2}$$

where α_t is the learning rate. We can rewrite this rule to resemble gradient descent by defining a loss for learning:

$$\hat{Q}^*(s,a) \leftarrow \hat{Q}^*(s,a) - \alpha_t[\hat{Q}^*(s,a) - (r + \gamma \max_{a'} \hat{Q}^*(s',a'))] \tag{2.3}$$

with the loss being defined as $\mathcal{L}(s,a,r,s') := \hat{Q}^*(s,a) - (r + \gamma \max_{a'} \hat{Q}^*(s',a'))$. This process is repeated for all state-action pairs, and it eventually converges to the optimal policy. The tabular Q-learning algorithm will then look something similar to this: Initially, the Q-values are assigned arbitrary values (often random). The agent then explores the environment by taking some actions and observes the resulting transition tuple (s,a,r,s'). The Q-values are eventually updated iteratively using the update rule defined in Equation 2.3.

A key concept in Q-learning is the process of selecting actions that balance exploration and exploitation. This is crucial because while we want to exploit the best-known actions to maximize rewards, we must also explore random actions to discover potentially better rewards. This trade-off is handled through the **epsilon-greedy** strategy. Initially, the agent selects actions randomly to explore the environment. Over time, as the agent learns more about the environment, it gradually shifts towards exploiting the best-known actions. This process is often referred to as the exploration-exploitation trade-off.

2.2.3 Deep Q-Learning

The key challenge in tabular Q-learning arises when we deal with large state and action spaces. Updating the Q-values for each state-action pair then becomes expensive and infeasible. A solution to this problem is to use neural networks as function approximators, a method known as deep Q-learning (DQN), introduced in Mnih et al. (2013) and Mnih et al. (2015). Rather than maintaining a table of Q-values, DQN uses neural networks to approximate the Q-function for all possible state-action pairs. This allows the agent to handle much larger and even continuous state and action spaces, as neural networks generalize across similar states, reducing the need to store Q-values for every possible state-action pair.

To further improve the learning process, we use a replay buffer $\mathcal{D}$ to store trajectory tuples (s,a,r,s'). This allows the agent to sample random batches of experiences during training. Moreover, we use two separate networks Q_{train} and Q_{target}. The "train" network is trained to predict Q-values, while the "target" network is used to compute the target Q-values for training. This would change the above update rule to the following:

$$\hat{Q}^*_{train}(s,a) \leftarrow \hat{Q}^*_{train}(s,a) - \alpha_t[\hat{Q}^*_{train}(s,a) - (r + \gamma \max_{a'} \hat{Q}^*_{target}(s',a'))] \tag{2.4}$$

The target network is periodically updated to match the training network. This process helps stabilize the learning process and accelerate convergence.

Despite these improvements, the original DQN algorithm still faces an issue called **overestimation bias**, where the Q-values are overestimated, leading to suboptimal behavior. This is addressed in the Double DQN paper in Hasselt et al. (2015). The solution to this problem is to modify the target calculation by using the current Q-network to select the best action and the separate target Q-network to evaluate its value. In particular, in 2.4 we replace $\max_{a'} \hat{Q}^*_{target}(s',a')$ with $\hat{Q}^*_{target}(s', \text{argmax}_{a'} \hat{Q}^*_{train}(s',a'))$, where argmax selects the action with the highest Q-value. This adjustment reduces overestimation and improves the stability and performance of the learning process.

3 Methods

This chapter will provide details of the Schieber Jass environment implementation, including how the state, action space, and rewards were represented. Then, it will provide details on the implementation of the fixed strategy, deep Q-learning, and human agents.

3.1 Environment Implementation

The environment is designed to simulate multiple rounds of Schieber Jass, accommodating different types of agents. Each episode corresponds to a single round of the game, consisting of nine tricks. At the beginning of an episode, the starting player selects the game type or passes this decision to their teammate. The starting player then begins the first trick by playing their first card, after which the other players play their respective cards. After all cards have been played, the total points for each team are counted, and the winning team is determined. For the following episodes, the starting player is selected cyclically, rotating to the next player in turn. This ensures that, during training, all agents have an opportunity to start a round. Ideally, the number of episodes is divisible by four, ensuring that each agent starts an equal number of times.

All basic rules of Schieber Jass were implemented in the environment, including the use of all 36 cards in Swiss German suits and all game types. However, for simplicity and to focus on the effective training of agents, the additional rules described in Chapter 2.1.1 were omitted. Each round awards a total of 157 points. The environment is designed to allow points to be counted per trick, enabling a more effective reward definition.

To implement the Schieber Jass environment, we first need to define its structure. In reinforcement learning, an environment typically includes two core methods: a "reset" method, which initializes the environment at the start of each episode, and a "step" method, which is called after an agent takes an action, updating the environment accordingly. In the context of our Schieber Jass environment, the reset method sets up a new round by initializing the starting player, dealing the cards to all players, and resetting the scores. It also prepares the environment for the first trick. The step method, on the other hand, is responsible for updating the game state after an agent plays a card. It then checks whether a trick should continue or end, calculates any rewards, sets up the next trick, and determines if the round is complete. The step method returns the updated state, the rewards for each player (the same for both players on the same team), and a flag indicating whether the round is finished.

3.1.1 State Representation

The state captures all the relevant information necessary for the agent to make decisions and learn effectively. In our case, the state is represented as a combination of various important features, each encoding different aspects of the game environment. For our problem, the state is represented by a combination of the following:

1. **Player Hands**: $s_1 \in \mathbb{Z}^{4\times 36}$ is a 4×36 matrix representing the players' hands. Each row corresponds to a player, and each column represents a card. An element in the vector is set to 1 if the card is present in the player's hand and 0 if the card is absent.

2. **Current Trick**: $s_2 \in \mathbb{Z}^{4\times 36}$ is a one-hot encoded 4×36 matrix representing the current trick. Each row corresponds to a player, and the columns indicate the cards. For each player, the card they played is encoded by setting the corresponding element to 1 in the 36-dimensional vector, while all other elements are set to 0.

3. **Leading Player**: $s_3 \in \mathbb{Z}^4$ is a one-hot encoded vector of length 4. It represents the player leading the current trick, which refers to the player who started or is about to start the trick, by setting the index corresponding to that player to 1 and the remaining indices to 0.

4. **Trick History**: $s_4 \in \mathbb{Z}^{4\times 36}$ is a 4×36 matrix. It encodes the history of past tricks, indicating which player played which card. Each row corresponds to a player, and each column represents a card. A value of 1 indicates that the corresponding player has played that card in a past trick, while a value of 0 indicates that the card was not played by that player.

5. **Game Type**: $s_5 \in \mathbb{Z}^6$ is a one-hot encoded vector of length 6. It represents the type of game being played in the current round, which can be one of the following, in this order: tops-down, bottoms-up, roses, shields, acorns, or bells. A value of 1 is set at the index corresponding to the current game type, while all other indices are set to 0.

6. **Decision Passed Flag**: $s_6 \in \mathbb{Z}$ is a scalar value, either 0 or 1, indicating whether the game type decision was passed to the teammate. A value of 1 represents that the decision was passed, while a value of 0 indicates otherwise.

The state is formed by combining the previously defined components, each of which is flattened before concatenation:

$$s = (vec(s_1), vec(s_2), .., vec(s_6)) \in \mathbb{Z}^{443}$$

In our implementation, the environment returns the state as a collection of key-value pairs instead of an encoded vector. This makes handling the information in the game much easier and allows each agent to process the state information differently, depending on its strategy. Additionally, this structure ensures that all agents view the state from the same perspective, which is particularly important for self-play training. To illustrate this, consider the following example: Each player is assigned an index 0 to 3. Player 0 encodes the state such that the first row in s_1 represents their hand. When Player 1 processes the state, the first row in s_1 will encode Player 1's hand, not Player 0's. The same logic applies to the other players, ensuring each agent's view of the state reflects their own role and actions in the game.

An interesting extension to this state representation would be to modify the trick history component s_4 so that it includes information about which player played which card in each specific trick. This could provide valuable additional context for decision-making. However, incorporating this change would increase the size of this component vector by a factor of 9. Due to time constraints, we were unable to explore this extension in this work, but it remains a promising idea to look into in the future.

3.1.2 Action representation

In Schieber Jass, an action corresponds to the card the agent chooses to play during a trick. In our case, the action space is defined as a single integer

$$a \in \{0, 1, 2, ..., 35\}$$

Each number uniquely represents a specific card. The agent outputs a number within this range to indicate its chosen action. The set of allowed actions is dynamically constrained based on the current state, in line with the Schieber Jass rules. This means that the action set changes within each trick depending on which cards are allowed to be played in that trick. In our implementation, this is achieved by ensuring that each agent is only allowed to play valid moves during each trick.

3.1.3 Rewards representation

Rewards in the Schieber Jass environment are tracked for both individual players and teams. Since the reward structure is sparse when rewards are provided only at the end of the round, we return the points won in each trick as rewards throughout the game. This approach significantly helps training, giving better feedback on the move of the agent. The rewards are normalized by dividing them by 157, the total points available in a round. More explicitly, the reward function is defined as:

$$r = \mathbb{1}_{won\_trick} \cdot \frac{1}{157} \sum_{card} p(card, game\_type)$$

where $\mathbb{1}_{won\_trick}$ is an indicator function that equals 1 if the agent's team won the last trick and 0 otherwise. The term $p(card, game\_type)$ represents the point value of each card based on the game type, and the summation is taken over all cards in the last trick. This immediate reward lies in the interval $[0, 1)$, i.e. $r \in [0, 1)$. The total reward across all teams and tricks by the end of the game sums to 1.

3.1.4 State Transitions

In our RL training framework, the environment dynamics $P(s' \mid s, a)$ are influenced not only by the agent's actions but also by the actions of the other players (opponents and the agent's partner). These players are treated as part of the environment, and their behavior, whether random, greedy, or part of a self-play scenario, is predefined and dictates how the environment evolves between states. A crucial aspect, as illustrated in Figure 3.1, is that the state is internally updated multiple times before transitioning to the next state. For example, after the agent plays a card, the remaining players take their turns until the trick is completed. The updated state also reflects the beginning of a new trick, resulting in a fully updated state. By the time the agent receives this updated state for the next time step, it encapsulates all events that occurred since the agent's last action. The final reward, however, is always computed based on the last trick, representing only a portion of the updated state.

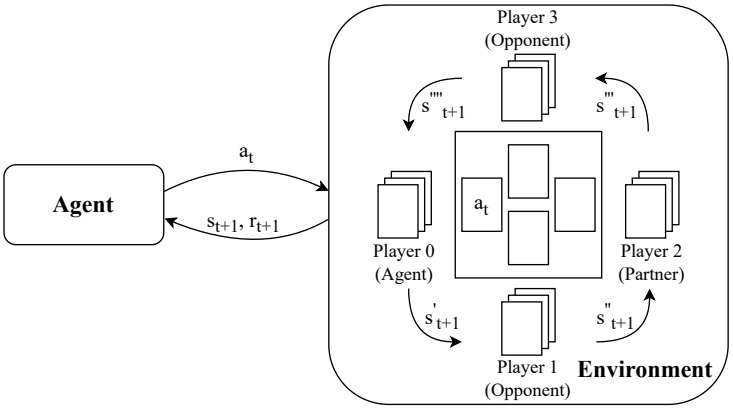

Figure 3.1: This figure illustrates the state transition process. After receiving state s_t, the agent selects an action a_t, which is applied to the environment. The environment updates its state multiple times until it is the agent's turn again. The fully updated state s_{t+1} and the reward r_{t+1} are then returned to the agent.

3.2 Agent Implementation

We implement three types of agents to play the game of Schieber Jass. Fixed-strategy agents rely on predefined rules and heuristics, primarily serving to test the environment and acting as simple opponents for training the deep Q-learning agents. The deep Q-learning agents were trained and evaluated to learn optimal strategies through reinforcement learning. Additionally, a human agent was implemented, allowing humans to play against other agents to evaluate their strengths and performance.

3.2.1 Fixed-Strategy Agents

We implemented two fixed-strategy agents, one of which is the random agent. When starting a new round, the random agent selects the game type randomly from the seven available options, including the option to pass the decision to a teammate. During gameplay, the random agent plays a randomly selected card from the allowed options in its hand. The agent never attempts illegal moves, as these are prevented in advance.

The second fixed-strategy agent is the greedy agent, which prioritizes short-term gains in each trick and follows a strictly defined strategy. At the start of a new round, the greedy agent selects the game type based on the strength of its hand. If it has many high-value cards (five or more cards that are Jack or higher), it chooses tops-down, while a hand of low-value cards (five or more cards that are Nine or lower) leads to a bottoms-up selection. For more mixed hands, the agent selects the trump suit for which it holds the most cards. Moreover, to incorporate some variability in passing, the greedy agent passes the game type decision to its teammate with a 14% probability. The distribution of the game type decision is balanced, as depicted in Table 3.1.

The greedy agent tries to win each trick whenever possible. For example, if it has a card in hand that can secure the trick, it plays the highest possible card to reduce the opponent's chance of winning the current trick. If winning the trick with its hand is not possible, the agent plays the lowest card in its hand. The greedy agent does not strategize for future tricks, nor does it consider the cards held by other players or the

Game Type	Probability
Tops-down	0.106
Bottoms-up	0.106
Roses	0.162
Shields	0.162
Acorns	0.162
Bells	0.162
Pass	0.140

Table 3.1: This table presents the frequency of game type selections by the greedy agent, based on empirical simulations of 1 million rounds.

cards already played in previous tricks. The strategies of the random and greedy agents are detailed in Figure 4.1.

3.2.2 Deep Q-Learning Agents

The primary focus of this section is on the implementation of the two deep Q-learning agents: a standard DQN agent and a Double DQN (DDQN) agent. Initially, we describe the foundational components shared by both agents, followed by the differences between the DQN and DDQN approaches. Further details on the training of these agents are provided in Chapter 4.

The core task of the learning agent is to make a decision on which card to play, given the current state of the environment. The agent learns from its actions by observing the rewards it receives, allowing it to improve its decision-making over time. This learning process is facilitated using neural networks.

State representation The agent's first step is to encode the state information into a numerical format suitable for neural network input. The environment represents the state as key-value pairs, which we transform into a flattened numerical vector for this purpose. We incorporated the features outlined in Section 3.1.1 to create this input representation. To facilitate experiments with both full and partial information scenarios, we introduced an adjustable flag that determines whether the opponent's and partner's hands are included:

- **Full information**: All features were included, except the trick history, resulting in a state vector of length 299.
- **Partial information**: We used the agent's own hand along with the trick history feature, excluding information about the opponent's and partner's hands. This setup resulted in a state vector of length 335.

We added the trick history feature only in the partial information setup to ensure comparable input sizes between the two setups, enabling fair comparisons. Once appropriately encoded, the state vector is fed into the neural network.

Q-networks We implemented customizable neural networks, referred to as Q-networks, to estimate the Q-values for each possible action in the action space. The input size of the Q-network is 299 or 335 (depending on the information setup) and the output size is

fixed at 36, corresponding to the number of possible actions. The Q-network architecture was designed to be flexible, allowing configurability in the number of hidden layers, their sizes, and the activation functions. Each learning agent employed two separate Q-networks with identical architectures: a training network, used to predict Q-values and select policy actions, and a target network, used exclusively to compute targets for loss calculation, as described in Section 2.2.3. The training network updates its weights after every episode, while the target network adjusts its weights more gradually by employing a soft update mechanism controlled by the target update rate parameter τ.

Invalid action masking Since the rules of Schieber Jass restrict which cards can be played during a trick, it was necessary to ensure that the agent only selected valid actions. Two approaches were considered to enforce this: the first involved heavily penalizing the agent with a high negative reward for making an invalid move. While this theoretically incentivizes the agent to avoid illegal actions, it still allows them to occur during play, which is undesirable. The second approach, which we adopted, was to mask invalid actions, ensuring that the agent could only choose valid moves at all times. This approach has been shown to be more effective as highlighted by Huang and Ontañón (2022). Once we implemented masking, the agent's performance improved significantly, and invalid moves no longer occurred. This resolved the issue entirely, and we did not investigate the problem further. Our invalid action masking mechanism works as follows (refer to Figure 3.3): after computing the Q-values using the Q-network, we assigned a very high negative value (specifically -1e7) to the Q-values corresponding to invalid actions. During action selection, the agent then chooses the action with the highest Q-value. Since valid actions are never assigned extremely negative Q-values, the agent is effectively constrained to playing only legal moves.

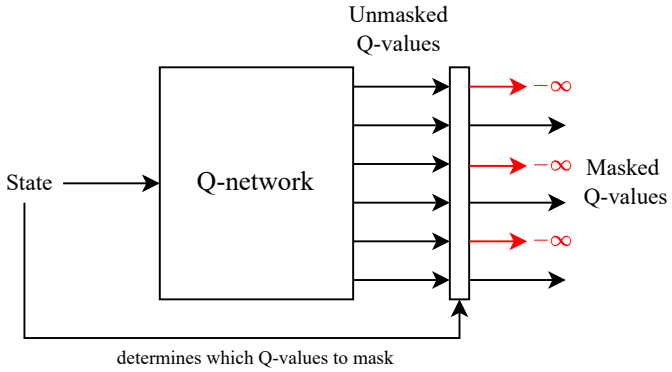

Figure 3.3: The Q-values for invalid actions are assigned a very large negative number, with information on which Q-values to mask provided in the state.

Game type selection We also need to explain how the deep Q-learning agents select the game type. The agent uses its Q-network to evaluate the initial game state for each possible game type. It then determines the game type associated with the highest maximum Q-value. If the decision to choose the game type was passed by the teammate, the agent ensures it does not pass again, as this option is excluded. To incorporate exploration in game type selection, we apply the epsilon-greedy strategy using the same hyperparameters as those used for action selection.

Difference between DQN and DDQN Finally, we outline the main and only difference between the DQN and the DDQN agents. To compute the target expected values, the DDQN agent selects the best action for the next state using the training network, but it retrieves the corresponding Q-values from the target network. This approach effectively decouples action selection from action evaluation. In contrast, the DQN agent directly calculates the target expected values using the maximum Q-value from the target network for the next state without decoupling the selection and evaluation.

3.2.3 Human Agents

We implemented a human agent to enable a human player to participate in the game. This agent utilizes the Python "inquirer" module, which allows players to select their moves through a command-line interface. Only valid moves are displayed, as illegal moves are filtered out beforehand.

3.3 Implementation Details

The full source code for this project is available on GitHub[1]. The entire implementation is written in Python. The environment is designed to be adaptable and could be extended to adhere to the Gymnasium interface (Towers et al., 2024), enabling easier integration with standard RL tools in the future. Training and evaluation parameters for the agents can be configured through the command-line interface. The learning agents were implemented using the PyTorch deep learning framework. The project runs via the command-line interface, and an optional print flag provides step-by-step information about the game, including the cards played by each agent. A graphical user interface (GUI) was not implemented due to time constraints and a focus on more important aspects of the project.

[1] https://github.com/albinkoshy/jass-ai

Algorithm 1 Random Agent's Strategy

Input: State s
From s get all necessary information and define them in variables.
if *is_starting_trick* **then**
  Select a random card from *hand*:
  card ← random.choice(*hand*)
else
  Determine valid cards:
  valid_hand ← get_valid_hand(*hand*)
  Select a random card from *valid_hand*:
  card ← random.choice(*valid_hand*)
end if
Return: *card*

Algorithm 2 Greedy Agent's Strategy

Input: State s
From s get all necessary information and define them in variables.
if *is_starting_trick* **then**
  Select the highest card from *hand*:
  card ← get_highest_card(*hand*)
else
  Determine valid cards:
  valid_hand ← get_valid_hand(*hand*)
  if *valid_hand* **is not empty then**
    Select the highest card from *valid_hand*:
    highest_card ← get_highest_card(*valid_hand*)
    if can_win_trick_with(*highest_card*) **then**
      card ← *highest_card*
    else
      Select the lowest card from *valid_hand*:
      card ← get_lowest_card(*valid_hand*)
    end if
  else
    Get trump cards:
    trump_cards ← get_trump_cards(*hand*)
    if *trump_cards* **is not empty then**
      Select the highest card from *trump_cards*:
      card ← get_highest_card(*trump_cards*)
    else
      Select the lowest card from *hand*:
      card ← get_lowest_card(*hand*)
    end if
  end if
end if
Return: *card*

Figure 3.2: This figure illustrates the strategies of the random and greedy agents.

4 Experiments

This chapter presents the experiments conducted with the implemented environment and agents. It details the training and evaluation processes for the DQN and DDQN agents. We then compare various aspects, including performance with different loss functions, variations in network architecture, and the impact of partial versus full information settings. We also compare the agents' win rates against random, greedy, and opposing DQN/DDQN agents.

4.1 Training Setup for Deep Q-Learning Agents

We experimented with 2 hidden layers, using configurations (128, 128), (256, 256), and (512, 512). The tested activation functions included ReLU, sigmoid, and tanh. To train our Q-networks, we utilized a replay memory to store transition tuples (s, a, r, s'). The replay memory was configured with a capacity of 50,000, ensuring that it always retained the most recent transitions by discarding older entries once the limit was reached. The training was performed using a batch size of 512. While we found this setup effective, the memory size and batch size were not extensively tuned and could be explored further in future work. We employed an epsilon-greedy strategy to balance exploration (random actions) and exploitation (policy actions). The epsilon decay was configured such that the exploration probability remained above the minimum value for 60% of the training duration. To optimize the Q-network, we experimented with both Mean Squared Error (MSE) and Mean Absolute Error (MAE) as loss functions. The learning rate was set to 0.00005 and the target update rate was configured to 0.005 after extensive testing. The discount factor γ was set to 1, as each round consists of only nine tricks, and the rewards from each trick are of similar magnitude. The finalized range of hyperparameters used during training is summarized in Table 4.1.

Category	**Hyperparameter**	**Tested Values**
General Settings	Episodes ($N_{episodes}$)	$500,000$
	Agent Type	`dqn`, `double_dqn`
Network Settings	Hidden Layer Sizes	(128, 128), (256, 256), (512, 512)
	Activation Function	`relu`, `sigmoid`, `tanh`
Training Dynamics	Batch Size	512
	Learning Rate	0.00005
	Target Update Rate (τ)	0.005
	Epsilon (ϵ)	$1 \rightarrow 0.01$
	Discount Factor (γ)	1
	Replay Memory Size	$50,000$
	Loss Function	`smooth_l1`, `mse`

Table 4.1: Candidate hyperparameters tested in the experiments.

The training loop used to train the agents is described in Algorithm 3. The algorithm initializes the environment and players and then iterates through each episode. To train the learning agent, we set the first player as the learning agent, with the other three agents as greedy agents, forming an opposing greedy team and a greedy partner. The starting player rotates to the next player at the beginning of each episode. The learning agent's policy is optimized at the end of each episode, and this process continues until all episodes are completed.

Algorithm 3 Training Loop for Schieber Jass Agents

Input: Environment env, Players $players$, Number of episodes $N_{episodes}$
for episode = 1 to $N_{episodes}$ **do**
  Reset environment env and obtain initial state s
  Starting player chooses game type, optionally passes to teammate
  Set game type in env and initialize current player
  Initialize state-action tracking for all players
  while game not done **do**
    if state-action pair exists for current player **then**
      Store transition (s, a, r, s') for current player to its replay memory
    end if
    Current player selects action a using its policy
    Update state-action tracking for current player
    Execute action a in env, observe next state s', rewards r, and done flag
    Update current player and set $s \leftarrow s'$
  end while
  for each player $p \in players$ **do**
    Store final transition (s, a, r, s') with $done = True$
    if p is a learning agent **then**
      Perform model optimization for p
    end if
  end for
end for

Figure 4.1: The training loop used to train the DQN and DDQN agents.

The training was carried out on the ETH Euler cluster, which primarily uses AMD EPYC CPUs. Training was conducted for approximately 500,000 episodes, utilizing a single CPU core with 2 to 4 GB of memory. Completing 500,000 episodes took roughly 10–12 hours. Therefore, using GPUs to speed up the process is recommended. However, GPUs were not available for our project, so training was carried out only on CPUs instead.

4.2 Training Loss and Reward Comparisons

In this section, we compare the training and reward plots for different hyperparameter settings, based on the hyperparameters outlined in Table 4.1. The activation function was fixed to ReLU, as we did not observe a significant difference compared to the other activation functions. We measure the total reward for the agent's team after each round (i.e., after nine tricks) and calculate the average total reward over 1,000 episodes. All experiments were done with three different seeds and we depict the mean of these and the standard deviation. The plots were smoothed using a moving average filter to make the lines more distinguishable. In general, if the curves cross the 0.5 reward boundary on the y-axis (dashed line), it indicates that the agent's team outperforms the opponent team. This was observed across all the hyperparameter experiments evaluated below.

4.2.1 Hidden Network Size Comparison

First, we compare the influence of the size of the hidden layers using the DQN agent with MAE loss. As seen in Figure 4.2, the network sizes (128, 128) and (256, 256) exhibit similar performance, while (512, 512) performs worse. The (256, 256) configuration may slightly outperform the (128, 128) setting, although this could be due to chance. Moreover, comparing the loss and reward curves for (512, 512) suggests overfitting.

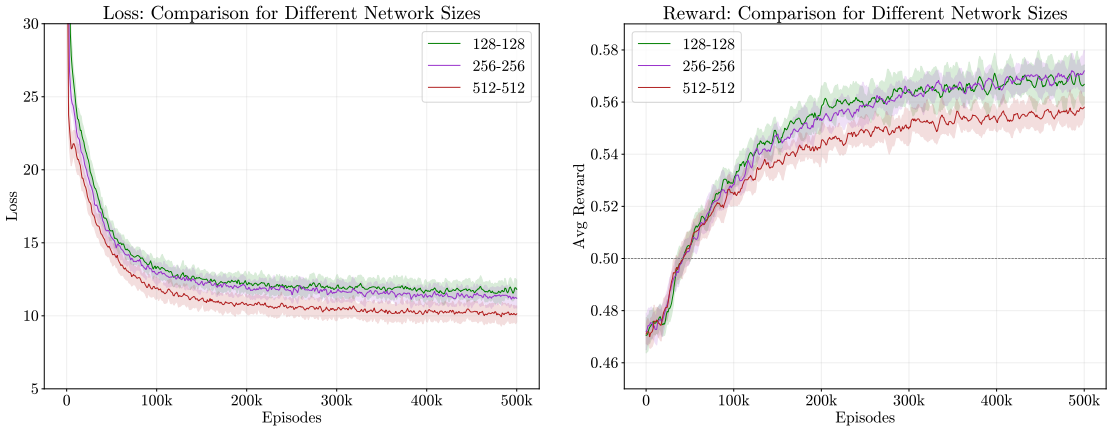

Figure 4.2: Loss and reward curves for different hidden network size configurations

4.2.2 Loss Function Comparison

Now, we compare the performance of the DQN agent using MSE and MAE loss functions with a hidden layer size of (128, 128). The curves are shown in Figure 4.3. Based on the performance, we conclude that MSE loss might be a better choice, as it outperforms the MAE loss agent. The loss curves don't provide significant insight, as the loss functions themselves differ substantially. The MSE loss might be better, as it is more sensitive to large errors compared to MAE, where larger errors have less impact on the loss calculation.

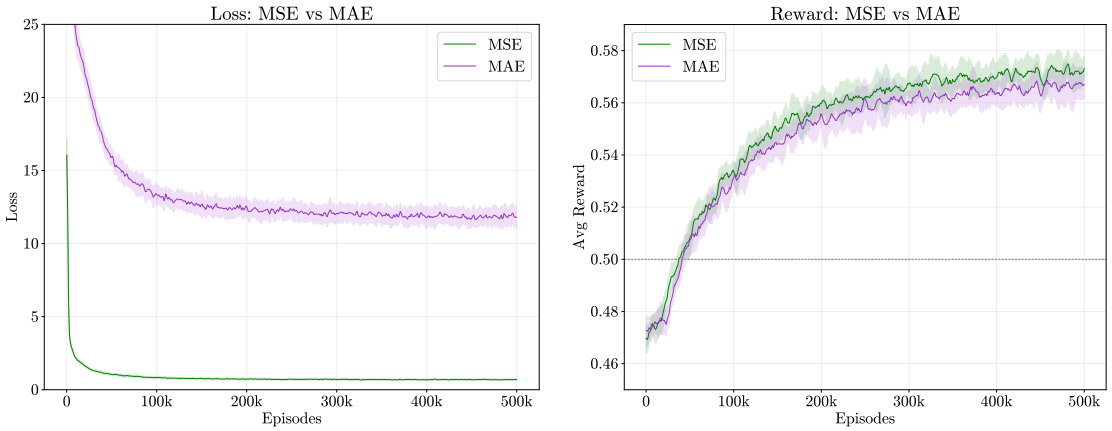

Figure 4.3: Loss and reward curves for different loss functions

4.2.3 DQN vs. DDQN

We fixed the hidden network size configuration to (128, 128) and used the MAE loss to compare the performance curves (shown in Figure 4.4) of the DQN and DDQN agents. As observed, there is no significant difference between the two agents. The lack of distinction could be due to a suboptimal learning rate or other misconfigured hyperparameters. It might also result from insufficient training time, as the advantages of DDQN often become more evident during longer training periods with higher $N_{episodes}$. Furthermore, the observed similarity in performance could also be because the differences between the two agents are inherently subtle.

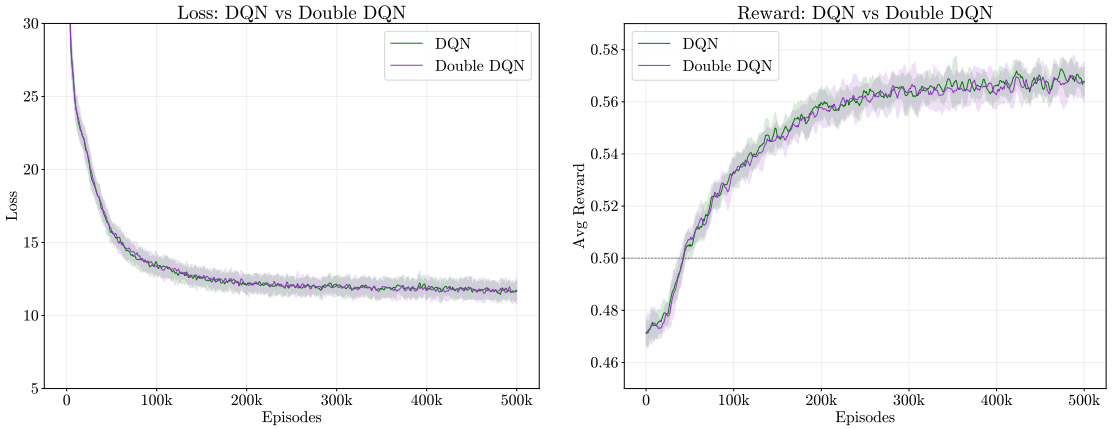

Figure 4.4: Loss and reward curves for DQN and DDQN

4.2.4 Full and Partial Information Setting Comparison

Finally, we compare the performance difference between the full and partial information settings for the DQN and DDQN agents in Figure 4.5. As in the previous comparisons, we fixed the hidden network size configuration to (128, 128) and used MAE as the loss function. As expected, the partial information setting performs worse than the full information setting. Surprisingly, the agent's team surpasses the opposing greedy team on average after around 80,000 episodes. This demonstrates that the agent has sufficient

information to learn effectively and win games, even in the partial information setting. Consistent with the earlier experiment, there is no noticeable performance difference between the DQN and DDQN agents in this comparison.

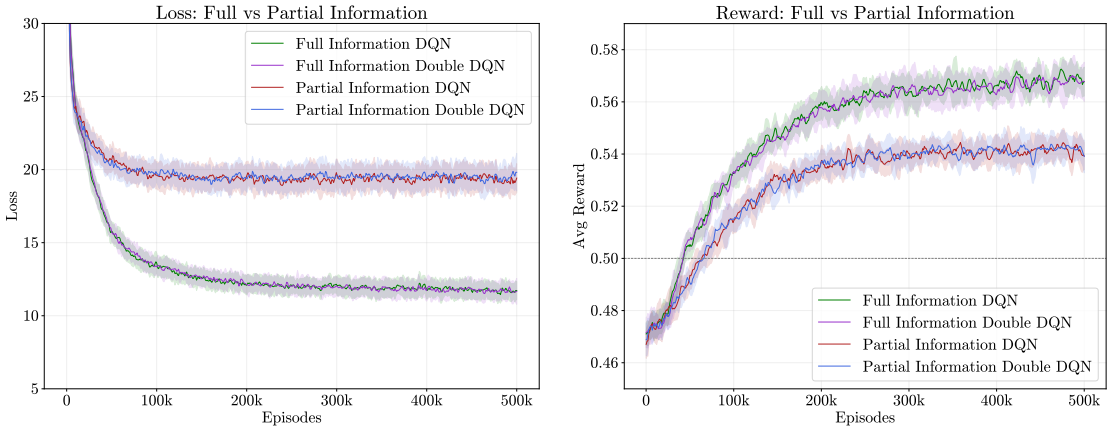

Figure 4.5: Loss and reward curves for full and partial information settings

4.3 Win Rate Comparison

An important experiment involves evaluating the performance of our implemented agents against each other. We conduct experiments using the random, greedy, DQN, and DDQN agents. For the DQN and DDQN agents, we compare their performance under full information (FI) and partial information (PI) setups. Each agent is partnered with a greedy agent, and we simulate 1,000 Schieber Jass rounds. The winning rates are presented in Table 4.2.

Agent	**Random**	**Greedy**	**DQN-FI**	**DQN-PI**	**DDQN-FI**	**DDQN-PI**
Random	50 ± 0	44 ± 1	33 ± 1	37 ± 1	33 ± 1	37 ± 1
Greedy	56 ± 1	50 ± 0	38 ± 2	43 ± 3	38 ± 2	43 ± 2
DQN-FI	67 ± 1	62 ± 2	50 ± 0	55 ± 1	51 ± 3	55 ± 2
DQN-PI	63 ± 1	57 ± 3	45 ± 1	50 ± 0	44 ± 3	50 ± 3
DDQN-FI	67 ± 1	62 ± 2	49 ± 3	56 ± 3	50 ± 0	56 ± 3
DDQN-PI	63 ± 1	57 ± 2	45 ± 2	50 ± 3	44 ± 3	50 ± 0

Table 4.2: Mean and standard deviation of the winning rates in percent out of 1,000 games across 4 seeds. The winning rates for the agent's team in the left column are against the opponent agent's team in the top row. Each agent plays with a greedy partner.

The agents behave as expected, with the full information DQN and DDQN agents winning against all other agents. Comparing DQN-FI and DDQN-FI, their performance is similarly strong. The partial information DQN and DDQN agents are competitive, easily beating the greedy agents but naturally having a disadvantage against the full information agents.

4.4 DQN Game Type Selection

It would also be interesting to observe, how the trained agents select the game types when they start a round of Schieber Jass. To investigate this, we experimented similar to the one in Table 3.1, where we simulated rounds of Schieber Jass and recorded how often the DQN agents selected each game type. The results are presented in Table 4.3. This experiment was performed for both the DQN agents trained under full information (DQN-FI) and partial information (DQN-PI) settings. Interestingly, the DQN-FI agent passes its game type decision to its greedy teammate approximately 50% of the time. In contrast, the DQN-PI agent does so much less frequently. This suggests that the DQN-FI agent has learned to rely on its teammate, recognizing that the teammate might have better cards and is better positioned to make decisions when the DQN-FI agent has access to the full card distribution. Both agents prefer to take the bottoms-up game type over tops-down, a pattern that remains unexplained. Further, the DQN-FI agent selects trump game types in a relatively balanced manner, while the DQN-PI agent shows a clear preference for roses, choosing it much more frequently than acorns.

Game Type	Probability DQN-FI	Probability DQN-PI
Tops-down	0.038	0.210
Bottoms-up	0.151	0.307
Roses	0.069	0.043
Shields	0.073	0.022
Acorns	0.080	0.019
Bells	0.069	0.035
Pass	0.519	0.363

Table 4.3: This table presents the frequency of game type selections by the full information (DQN-FI) and partial information (DQN-PI) DQN agents based on empirical simulations of 1 million rounds.

4.5 Experiment Discussion

The experiments conducted in this chapter highlight the successful training of DQN and DDQN agents for Schieber Jass in both full and partial information settings. All trained agents successfully learn effective strategies and outperform the fixed-strategy teams after approximately 100,000 episodes.

We found that overly large networks are undesirable, as they lead to overfitting. We also observed that the MSE loss performs better than the MAE loss. Interestingly, we observed no significant difference in performance between the DQN and DDQN agents. This could be due to subtle differences in the algorithms, potential hyperparameter tuning issues, or the limited number of training episodes. Despite this, both agents performed well.

The analysis of game type selection revealed notable patterns. The DQN-FI agent often deferred game type decisions to its greedy teammate, likely recognizing the teammate's advantage in situations where the full distribution of cards among players was available. This behavior was also observed in the DQN-PI agent, albeit to a lesser extent. These observations suggest that the greedy agent employs a near-optimal strategy when selecting the game type.

5 Conclusion

In this project, we explored the application of RL techniques to the Swiss card game Schieber Jass. We implemented a custom Schieber Jass environment with random, greedy, and deep Q-learning agents. This environment can be easily modified to train learning agents through self-play. We recommend masking invalid actions, as this significantly improves both training and performance. We focused on training DQN and DDQN agents in both full and partial information settings, evaluating their performance through extensive experiments. Our results demonstrate that both DQN and DDQN agents are capable of learning effective strategies in Schieber Jass, with full-information agents performing consistently better due to the absence of information uncertainty. Furthermore, we identified a suitable set of hyperparameters for training these agents.

In conclusion, this work highlights the potential of reinforcement learning in competitive games like Schieber Jass, paving the way for more complex and collaborative RL applications in games involving multiple agents and partial information.

6 Outlook

While this project has laid a solid foundation for training and evaluating RL agents to play Schieber Jass, several exciting opportunities for future improvements remain. One significant next step would be to improve and explore the partial-information aspect of the game, by adjusting the state and action spaces to incorporate more detailed information between tricks and rounds. For example, probabilistic models could be used to estimate the cards held by other players, providing agents with a more detailed understanding of the game state.

Another important step is the implementation of self-play, enabling agents to train solely by playing against themselves. Self-play has proven effective in other games and could help create stronger and more adaptive agents in Schieber Jass. Moreover, exploring alternative agent methodologies, such as policy-gradient and actor-critic methods, could potentially outperform the value-based methods currently used.

Furthermore, since Schieber Jass involves team-based play, which, alongside the partial-information aspect, is one of the main challenges, advancing towards multi-agent RL methods would be highly relevant. In such methods, agents share the same resources and collaborate to optimize team performance. As observed with expert human players, coordination is key to winning. For example, agents could learn to coordinate strategies dynamically and adapt to their partners' playstyles, similar to the teamwork displayed with human players. Such extensions would likely result in major improvements and align closely with the collaborative dynamics inherent in Schieber Jass.

These next steps would not only advance the capabilities of RL agents in Schieber Jass but also contribute valuable insights to the broader field of strategic decision-making in team-based, imperfect-information games. This could further fuel research into solving larger and more complex strategic problems.

Bibliography

Brown, N., & Sandholm, T. (2019). Superhuman AI for multiplayer poker. *Science*, *365*(6456), 885–890. https://doi.org/10.1126/science.aay2400

Charlesworth, H. (2018). Application of Self-Play Reinforcement Learning to a Four-Player Game of Imperfect Information. https://arxiv.org/abs/1808.10442

De Jong, M. (2021). Lost Cities: Using Deep Q-Networks to Learn a Simple Card Game [Accessed: 2024-12-23]. https://maxwelldejong.com/machine_learning/lost_cities/

Hasselt, H. v., Guez, A., & Silver, D. (2015). Deep Reinforcement Learning with Double Q-learning. https://arxiv.org/abs/1509.06461

Huang, S., & Ontañón, S. (2022). A Closer Look at Invalid Action Masking in Policy Gradient Algorithms. *The International FLAIRS Conference Proceedings*, *35*. https://doi.org/10.32473/flairs.v35i.130584

Jaquet, B., & Kutirov, M. (2024). Reinforcement Learning Agents for the Game of Differenzler Jass. https://github.com/Huo12345/reinforcment-learing-differenzler-jass/blob/master/Report.pdf

Kita, H., Koyamada, S., Yamaguchi, Y., & Ishii, S. (2024). A Simple, Solid, and Reproducible Baseline for Bridge Bidding AI. https://arxiv.org/abs/2406.10306

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing Atari with Deep Reinforcement Learning. http://arxiv.org/abs/1312.5602

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., & Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, *518*(7540), 529–533. http://dx.doi.org/10.1038/nature14236

Moravčík, M., Schmid, M., Burch, N., Lisý, V., Morrill, D., Bard, N., Davis, T., Waugh, K., Johanson, M., & Bowling, M. (2017). DeepStack: Expert-level artificial intelligence in heads-up no-limit poker. *Science*, *356*(6337), 508–513. https://doi.org/10.1126/science.aam6960

Niklaus, J. (2019). *JassTheRipper – A High-Human Artificial Intelligence for the Swiss Card Game Jass* [Master's thesis]. University of Bern, University of Fribourg, University of Neuchatel. https://niklaus.ai/files/theses/Master_Thesis.pdf

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., Sifre, L., van den Driessche, G., Graepel, T., & Hassabis, D. (2017). Mastering the game of Go without human knowledge. *Nature*, *550*, 354–. http://dx.doi.org/10.1038/nature24270

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement Learning: An Introduction* (Second). The MIT Press. http://incompleteideas.net/book/the-book-2nd.html

Swisslos. (2024a). Introduction to Schieber Jass rules [Accessed: 2024-12-09]. https://www.swisslos.ch/en/jass/informations/jass-rules/schieber-jass.html

Swisslos. (2024b). Introduction to the basic principles of Jass [Accessed: 2024-12-09]. https://www.swisslos.ch/en/jass/informations/jass-rules/principles-of-jass.html

Towers, M., Kwiatkowski, A., Terry, J., Balis, J. U., De Cola, G., Deleu, T., Goulão, M., Kallinteris, A., Krimmel, M., KG, A., et al. (2024). Gymnasium: A Standard Interface for Reinforcement Learning Environments [GitHub: https://github.com/Farama-Foundation/Gymnasium]. https://arxiv.org/abs/2407.17032

Zha, D., Lai, K.-H., Cao, Y., Huang, S., Wei, R., Guo, J., & Hu, X. (2020). RLCard: A Toolkit for Reinforcement Learning in Card Games. https://arxiv.org/abs/1910.04376