MASTER'S THESIS

Mastering Cooperative, Incomplete Information Board Games by Self-Play

van der Weij, W.

Award date:
2022

THESIS

MASTERING COOPERATIVE, INCOMPLETE INFORMATION BOARD GAMES BY SELF-PLAY

by

W.J. van der Weij

in partial fulfillment of the requirements for the degree of

Master of Science
in Software Engineering

at the Open University of the Netherlands, Faculty of Science
Master's Programme in Software Engineering
to be defended publicly on Monday May 30, 2022 at 1:30 PM.

Student number:

Course code:	IMA0002	
Thesis committee:	dr. Twan van Laarhoven	Radboud University
	dr. ir. Frank Tempelman	HAN
	dr. ir. Martijn van Otterlo	Open University

Intentionally blank page

ABSTRACT

The goal of this research is to determine how Reinforcement Learning (RL) can be best used for board-card games. Board-card games are marble racing games with imperfect information and team play. These game characteristics bring more complexity after the successes of RL research on perfect information games like Go and Chess.

This research is based on, and part of, broader developments in RL. The study of related work led to the choice to apply the techniques of Deep Q-Networks (DQN) and Deep Monte Carlo (DMC) to a board-card games case study. The agents learn from self-play only. RLCard was chosen as the RL research framework.

The results with DMC are much better than with DQN. DQN scores a maximum win rate of 68% against randomly playing agents, and manual analysis shows that the DQN agents learns the game poorly. The variances caused by the characteristics of board-card games (large action space, multi-agent, imperfect information) are the likely cause of the problems with the incremental nature, ϵ-greediness and Artificial Neural Network (NN) of DQN. DMC scores a win rate of 99% against randomly playing agents and 49% against rule-based agents after 49 days of training. Manual analysis shows that DMC learns gradually. Experiments with the NN size, reward function and the observation model demonstrate that even better results are likely with more training.

DMC seems to be the best technique to learn board-card games. The playing strength is good enough and other properties of RL techniques are interesting enough to be applied to board-card games in practice. Future work on DMC would be worthwhile to determine how strong the agent can become.

PREFACE AND ACKNOWLEDGMENTS

I could not have foreseen that a small idea that I had more than seven years back merged with my interest in RL and initiated the research project that is the subject of this thesis. Back then, the bucket list item was to create an iOs app. Coincidentally, this became an implementation of the board-card game called Keezen. The question arose from here: how can we apply RL techniques to play this type of games? The research objective was straightforward, but the game was more complex than anticipated. After overcoming the challenges, I am pleased with the result described in this thesis.

I like to express my gratitude to my supervisors. Dr. Twan van Laarhoven supported me throughout this project. I am thankful for his support, listening ears, and advice. And it was Twan who dared to take on this project with me. I thank dr. ir. Frank Tempelman for the support and advice, especially in the difficult times of the project. I also thank dr. ir. Martijn van Otterlo for his criticism and enthusiasm, which made me think things over and certainly improved the result.

CONTENTS

List of Figures

List of Tables

1

INTRODUCTION

Scientific research in the area of Artificial Intelligence (AI), and the sub-area Machine Learning (ML) in particular, has shown tremendous progress in recent years. Computers are learning on their own and are becoming better than humans. AI research is the driver behind developments such as image recognition, face recognition, speech recognition, virtual assistants, autonomous cars, robotics, language processing, game playing, and data classification.

ML is a part of AI that studies computer systems that learn from experiences or data. Rules and logic are not manually programmed. A machine recognizes patterns in data or results of actions and learns to apply these experiences.

This research project takes place within a part of ML called Reinforcement Learning (RL). In RL research the actions of agents in an environment and their (optimal) results or rewards are studied. RL is, among others, successfully applied to game-playing agents that learn from self-play.

The research in game-playing plays a prominent role in the development of AI, and RL in particular. Games provide a controlled environment for simulations and experiments with AI algorithms. Games can be considered models of real-world problems. The progress of AI can be measured on game-playing. AI research has brought new and creative insights into game-playing. Especially RL-based game-playing agents came up with strategic novelties. AlphaZero has enriched the playing field of Chess and Go with fascinating moves in beautiful matches [Silver et al., 2018].

After the successes of applying RL to games like Go and Chess, the research on RL has taken on further challenges. Games with incomplete information and cooperative game-play bring more complexity to the table. In games with incomplete information, the current state is not entirely known. In cooperative game-play, agents have to find ways to work together. Games with these properties have specific mechanics, making it worthwhile to research cooperative, incomplete information games. Furthermore, incomplete information and cooperation are common properties in practical applications of AI, such as autonomous cars.

One category of games with incomplete information and cooperative game-play is board games with marbles that are moved by playing cards. This type of games is referred to as **board-card games**. Two teams of two players race their marbles to the finish area. The first team to finish all marbles wins. A deck of cards provides functions (e.g., start, run and

switch) that a player can apply to the marbles on the board. The deck of cards and team-play lead to incomplete information and cooperative game-play. Examples of board-card games are Dog, Tock/Tuck, Sorry! and Keezen.

1.1. Research goal

The goal of this research is to determine how RL can be used for board-card games. This research aims to develop game-playing agents that perform well on imperfect information and cooperative play. The agents are based on the principles of RL, following recent scientific progress on other games like Go, Chess, Shogi, Poker, Bridge, etc. The agents learn from self-play from scratch without any domain knowledge.

1.2. Research questions

The research searches for answers to the following questions:

- **How can an RL-based agent achieve the strongest gameplay in incomplete information and cooperative board-card games?**

  This main research question points to the overall goal of the research. The next questions are sub-questions, supporting the main question.

  (i) What RL research is done in incomplete information and cooperative games?

  (ii) How can these RL techniques be used for a board-card game, with Keezen as a case study?

  (iii) How do RL agents perform in board-card games?

  (iv) Which RL techniques lead to the best results?

1.3. Research design

This research is based on and part of the broader developments in the research area of RL. The research started with the study of existing research, which is described in the next chapter. The choice of techniques originate in former research.

We try to find answers to the research questions by implementing multiple RL-based agents and conducting experiments. This RL research project aims to learn without transferring externally collected data or knowledge to the agents. All data is generated by self-play. The agents start from scratch and only learn from their own experiences, similar to AlphaZero [Silver et al., 2018].

This project does not study all board-card games separately. The game Keezen is taken as a case study. Although the games Dog, Tock, and Sorry! are not investigated directly, the intention and expectations are that the conclusions could be generalized to these board-card games.

The results of the experiments are examined:

- quantitatively: by analyzing the performance of agents in tournaments.
- qualitatively: by manual analysis of the gameplay.

1.4. CONTRIBUTIONS

The following contributions of this research could be identified:

- An overview of RL research on games.
- An RL model of Keezen
- Full implementations to train RL agents for Keezen.
- Agents with good gameplay against people

1.5. THESIS OUTLINE

The remainder of this thesis is structured as follows. The next chapter introduces related work to this research. It handles RL and game-playing research. Chapter 3 describes the board-card games with an analysis of their characteristics. Then the method is described in Chapter 4, followed by the experiments in Chapter 5. The results and discussions are in Chapters 6 and 7. This report ends with the conclusions, future work and reflection in Chapter 8.

2

REINFORCEMENT LEARNING AND GAMES

This chapter describes the related work in the area of RL and game-playing. The first part describes the RL techniques that are relevant for this research. The last section gives an overview of research on games. The main part of the description is based on 2nd edition of the book 'Reinforcement Learning - An Introduction' by Sutton and Barto [Sutton and Barto, 2018], with some additions based on [Graesser and Keng, 2019].

2.1. REINFORCEMENT LEARNING

RL is one of the main areas of research in ML, next to supervised learning and unsupervised learning. In RL agents discover which actions return the highest reward by trying them. An agent tries to behave optimally without supervision or a complete model of the environment. RL is a natural fit for sequential decision-making problems such as games and many real-world problems.

Figure 2.1 shows the basic RL control loop. In RL, the *agent*, the learner and decision-maker, explores the *actions* in the *environment* in a quest for the best *rewards*. Rewards are numerical values from the environment. The rewards may be sparse. *Training* consists of simulations or trial-and-error without an intelligent teacher. The training develops a *policy* that governs the behavior of the agent.

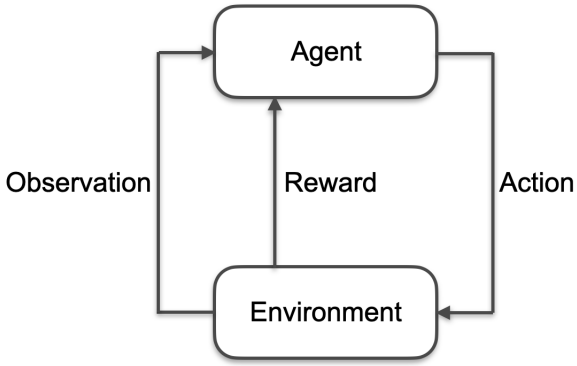

Figure 2.1: Reinforcement learning control loop.

Figure 2.1 also applies to multi-agent environments. The agents are part of the environment, and each agent experiences the environment with the changes that follow from the

actions by other agents.

The Markov Decision Process (MDP) [Bellman, 1957] is used commonly in science to research systems, states, and state transitions. It is a mathematical model for decision-making and a natural fit for games. An MDP is defined by a tuple $\langle S, A, P, R, \gamma \rangle$, where:

- S is a set of states (state space)
- A is a set of actions (action space)
- $P_a(s, s') = Pr(s'|s, a)$, state transition function. A probability distribution that an action a in state s leads to a certain next state s'.
- $R_a(s, s')$, reward function. The immediate rewards for the transition from state s to next state s' due to action a.
- γ is the discount factor, $\gamma \in [0, 1]$. This factor balances between the values of immediate and future rewards.

A process is said to have the *Markov property* if the probability distribution of future states P depends only upon the current state s. In other words: the future of the process follows from the current state s and does not depend on the past. If agents use observations from the past (like already played cards) to come to optimal decisions, then the process does not have the Markov property.

In general, the agent does not know the P and R functions a priori and experiences these by executing actions in the environment. When an agent interacts in a sequential manner and each interaction is a time step, then the *trajectory* begins like $s_0, a_0, r_1, s_1, a_1, r_2, ...$ Every time step is a tuple $\langle s, a, r \rangle$. An *episode* is a trajectory that spans a full game from start to finish. In a finite MDP, which is common for many games, the sets of states, actions, and rewards have a finite number of elements. An episode of a board-card game is finite because the marbles move in the direction of the finish fields.

The return G_t is the sum of all rewards. With rewards r, time steps t and discount rate γ, the return is:

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... \text{ or } G_t = r_{t+1} + \gamma G_{t+1} \tag{2.1}$$

The goal of an RL algorithm is to learn a *policy* that governs the choice of actions to maximize the return G. The policy π is a probability distribution over possible actions in states. $\pi(a|s)$ is the probability of taking action a in state s under policy π. This defines a stochastic policy. A deterministic policy $\pi(s)$ maps one action a to a state s. Policy-based algorithms directly learn a policy. *On-policy* methods evaluate or improve the current policy that generates the trajectories, and *off-policy* methods evaluate or improve a different policy. In general, on-policy methods are more simple than off-policy methods. On-policy and off-policy algorithms differ in the way training data can be used. On-policy algorithms can only utilize the data from the current policy. The policy iterates through versions that have their specific training data that becomes unusable after the training. Examples of on-policy algorithms are Reinforce, Sarsa, Actor-Critic, and Proximal Policy Optimization (PPO). Off-policy algorithms can reuse any data to train. DQN and Monte Carlo Methods (MCM) are, in general, implemented as off-policy algorithms.

A *value function* describes how good states s or state-action pairs (s, a) are related to the future reward under a policy π. A value function $V(s)$ evaluates the value of a state s and

$Q(s,a)$ evaluates the value of a state-action pair (s,a). The value function $V_\pi(s)$ of a state s under a policy π is the expected return G when starting in s and following π thereafter. The value function $Q_\pi(s,a)$ is similar for the action a in state s. An algorithm is called value-based if the agent learns a value function (V or Q). Examples are Sarsa and DQN.

A *model of the environment* predicts the probabilities of the next state s' for each action a in a state s. The model is defined by the transition function $P(s'|s,a)$. Model-based approaches can be said to emphasize planning (a map of the environment) and model-free to emphasize learning (the values of actions or states). Model-based algorithms learn a model that describes the transition dynamics of the environment. Model-free algorithms do not do this and can not make predictions about the next states and expected rewards before taking actions. The agent utilizes the model to predict the behavior of the environment in the next steps. An environment with a large state or action space can be hard to model. A model is only useful if it can reliably predict many steps in the future. MCM and Q-learning are examples of model-free algorithms.

The *optimal policy* π_* gives the best rewards. For a process with finite episodes one or more optimal policies exist with an expected return G in all states greater than or equal to all other policies. All optimal policies share the same value function $V_*(s)$ or $Q_*(s,a)$. If the optimal value function is known, then the optimal policy is the policy that selects the highest scoring actions (greedy). Unlike Q_*, the V_* function requires to calculate one step further. Hence, the estimation of action values is beneficial compared to state values if no model of the environment is available.

The trade-off between *exploration* and *exploitation* is a challenge in RL. The agent should exploit experiences to obtain high rewards and explore new paths to discover better actions. The agent must balance these two to come to the best results. *Epsilon-greedy* is a method to balance exploration and exploitation randomly. Epsilon refers to the probability of choosing exploration over exploitation. Epsilon-greedy prefers exploitation with a small chance of exploration.

The state s can mean two things: the state of the environment and the state observed by an agent. If the observed state matches the environmental state, then the environment is *fully observable* otherwise the environment is *partially observable*. Partial observability is caused by hidden information (such as the cards in board-card games) or limitations in the measure equipment or the data representation. This means many real-world problems are partially observable.

In board-card games, the game state is not fully observable. The agents have to make decisions under uncertainty. The more general (Partially Observable Markov Decision Process (POMDP)) [Astrom, 1965], [Kaelbling et al., 1998] handles this by extending the MDP with:

- Ω is a set of observations
- $O(s,a) = Pr(o'|s',a)$: Observation function. A probability distribution that an action a in state s leads to a next observation o' in the next state s'.

In a POMDP agents make observations $o(s,a)$ based on the actions and the resulting states. The distribution function represents the probability of the state of the environment and forms the basis to act under uncertainty. The probability distributions over the partially observable states make the POMDP more computationally intensive than a MDP. The

POMDP can be seen as an MDP with an extended state space, with so-called belief states. A *belief state* of an agent is a probability distribution over the set of states (S).

The imperfect information adds complexity and requires extra computational resources if a POMDP is to be solved exactly. In board-card games the hidden state space, caused by the playing cards, is huge [1].

In practice, the optimal functions and policies are rarely found due to enormous state spaces (hidden or not). Computational power and available memory are two constraints that often make the use of approximations of value functions, policies and environmental models necessary. Function approximation is a possible solution for large state spaces and can be a solution for POMDPs too. In practice, function approximation can perform surprisingly well when providing the observations and ignoring the unobservable information. Non-linear function approximation is the subject of Section 2.5.

MCM and Temporal Difference learning (TD) [Sutton and Barto, 1987] are two fundamental classes of methods for solving finite Markov decision problems. Both methods have strengths and weaknesses. In general, MCM are simple and do not require a model of the environment, but require more computation. TD also requires no model and is incremental and harder to stabilize. These two types of algorithms are the subjects of the following two sections.

2.2. Monte Carlo Methods

MCM in general are solutions to problems based on repeated random sampling. To learn a state-value function for a policy, values of states and/or actions are estimated by averaging the returns from rollouts (play until the game finishes) from a state s. MCM works with finite episodic tasks.

MCM approximates optimal policies by maintaining an approximate policy and an approximate value function. Algorithm 1 shows an MCM algorithm. The value function $Q_\pi(s,a)$ is continuously improved regarding the current policy π while the policy is greedily improved with the updated value function. Together, these two moving targets try to approach optimality. The policy is replaced by a new policy when the new policy is uniformly better than the old policy. The policy that is used to generate behavior, the behavior policy, is different from the policy that is evaluated and improved, the target policy. The target policy can be greedy and the behavior policy explores all possible actions.

In MCM there are two ways of handling multiple visits to a state in the same episode. First-visit MCM estimates the value of a state based on the first visits to the state, while every-visit MCM averages the returns of all visits to a state. These two methods have small differences. First-visit is more studied and every-visit seems to fit better with function approximation.

MCM do not estimate values based on estimations of successor states, like TD. In other words: MCM do not bootstrap. MCM can efficiently estimate the value of a single state or a subset of states. MCM supports generating many episodes from a start state and only averaging the returns from the states of interest. As MCM works without a model of the environment, the algorithm calculates the state-action value function.

[1] To give an idea of the extra state space: if all 4 players have 5 cards and the other 32 cards are in the stock, then the number of possible combinations of unknown cards in the hands of the other 3 players is: $\binom{47}{5} \times \binom{42}{5} \times \binom{37}{5} = 5.7 \times 10^{17}$.

Algorithm 1 Monte Carlo Methods. The algorithm is first-visit and becomes every-visit by removing the if condition on line 9.

1: **Input:**
   A policy π to evaluate
2: **Output:**
   $Q_\pi(s,a)$ (value function)

3: **Initialize:**
   $Q_\pi(s,a)$, Returns(s, a)
4: **loop**(for each episode)
5: Generate an episode (tuples of $\langle s,a,r\rangle$ until game ends at T)
6: $G \leftarrow 0$
7: **for each** Step of the episode (t=0, t=1,..., t=T-1) **do**
8: $G \leftarrow \gamma G + r_{t+1}$
9: **if** s_t, a_t not in $s_0, a_0, s_1, a_1, ..., s_{T-1}, a_{T-1}$ **then**
10: Append G to $Returns(s_t, a_t)$
11: $Q_\pi(s_t, a_t)$ = avg$(Returns(s_t, a_t))$
12: **end if**
13: **end for**
14: **end loop**

Maintaining sufficient exploration is important in MCM. If the policy is deterministic then many state-action pairs will never be visited. Learning requires continuous exploration, the execution of multiple different actions in a state.

An advantage of MCM is that they may suffer less from violations of the Markov property than bootstrapping methods, like TD or Q-learning. This is because MCM do not update value estimates on the basis of other (incomplete observations of) states.

Monte Carlo Tree Search (MCTS) is a popular algorithm that is an extension of MCM with tree search and is used frequently for searching the best moves in games. The enormous progress of computer playing strength in large, perfect information games such as Go and Chess is largely achieved by using MCTS.

2.3. TEMPORAL-DIFFERENCE LEARNING

TD refers to learning methods that update estimated values based in part on previously calculated estimates. TD can learn directly from experience without a model of the environment, like MCM. TD methods only have to wait for the next step without knowing a final return, while MCM have to finish an episode.

The simpliest form of TD is one-step TD or TD(0).

Algorithm 2 Tabular TD(0).

Input:
A policy π to evaluate
Output:
V_π (value function)

Initialize:
α: step size parameter, $V_\pi(s) = 0, \forall s \in S$
loop(for each episode)
Initialize s
for each step of the episode **do**
$a \leftarrow$ action a for s given by π
Take action a, observe r, s'
$V_\pi(s) \leftarrow V_\pi(s) + \alpha[r + \gamma V_\pi(s') - V_\pi(s)]$
$s \leftarrow s'$
end for
end loop

TD combines the sampling of MCM with bootstrapping. TD updates estimates partly based on other estimates. TD are online and incremental. Especially with long or infinite episodes this is an advantage over MCM. TD learns from each transition. In practice TD seems to converge faster than MCM on stochastic tasks according to [Sutton and Barto, 2018]. In general, TD based algorithms also converge less robustly than MCM.

2.4. Q-Learning

Q-learning [Watkins, 1989] is an off-policy TD algorithm. The introduction of Q-learning can be seen as an early breakthrough in RL. The optimal action-value can be calculated with the recursive Bellman equation (2.2). The maximum future reward $Q(s,a)$ is the current reward r plus the maximum reward for the next state s'. The discount factor γ controls the weight of future states (early rewards are valued higher than late rewards).

$$Q(s,a) = r + \gamma \max_a(Q(s',a)) \tag{2.2}$$

State-action pairs $Q(s,a)$ are more efficient to calculate with than just values of states $V(s)$ if no model of the environment is available. The value of *Q(s, a)* can be approximated iteratively as in equation (2.3), which is the base of Q-learning algorithms.

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1},a) - Q(s_t,a_t)] \tag{2.3}$$

The learned action-value function $Q(s,a)$ directly approximates the optimal action-value function $Q_*(s,a)$, independent of the current policy. The policy still has an effect in steering the visits and updates of state-action pairs. The learning rate α weights the new information. A value of zero means the agent learns nothing and a value of 1 means only new knowledge is considered.

Algorithm 3 Q-learning. Resembles TD with Equation 2.3.

Input:
A policy π to evaluate
Output:
$Q_\pi(s,a)$ (action-value function)

Initialize:
$Q(s,a) = 0, \forall s \in S, a \in a(s)$
loop(for each episode)
Initialize s
for each step of the episode **do**
$a \leftarrow$ action for s derived from Q_π
Take action a, observe r, s'
$Q_\pi(s,a) \leftarrow Q_\pi(s,a) + \alpha[r + \gamma \max_a Q_\pi(s',a) - Q_\pi(s,a)]$
$s \leftarrow s'$
end for
end loop

2.5. NON-LINEAR FUNCTION APPROXIMATION

The simplest problems in RL, with small state and action spaces that fit entirely in arrays or matrices, might have exact solutions. This means that the optimal value function and policy is found. However, most RL problems of interest are more complex and need non-linear function approximation methods. NNs are widely used for these approximations.

NNs are a main component of the algorithms in this research. Non-linear activation functions let the NN approximate highly complex non-linear functions. The network architecture consists of the number of units and how these units are connected. The units are organized into groups, which are called layers. The layers are in a chain structure, with each layer being a function of the layer that preceded it. The main considerations are the depth of the network and the width of each layer. The depth corresponds to the number of layers. The first layer is called the *input layer*, and the final layer is called the *output layer*. Layers between these two are the *hidden layers*. Fully connected layers or dense layers, mean that every output of a layer is connected to every node in the next layer. A feedforward NN has no loops in the network. A NN is called recurrent if it has at least one loop. Connections between units of the NN have associated *weights* that the units typically use to compute a weighted sum of the inputs. The result is applied to a so-called *activation function*, to produce the unit's output or activation.

The NN approximates the value function V or policy π. The input of the NN consists of states s or observations o, which are converted to vectors. The outputs of the NN are the available actions with the corresponding Q-values.

Experiences are fed into the NN in batches. These batches consist of training samples to update the internal model (propagate through the NN). The batch size is a so-called *hyperparameter* that defines the number of samples that is processed before updating the parameters of the NN.

There are several types of networks with particular characteristics that fit different types of input data. Each type is characterized by the type of layers and the way the network is

computed. It is common to combine NN types to form a hybrid type. Three main type of networks can be identified:

- Multi Layer Perceptrons (MLP) or (deep) feedforward network)
  MLPs consist only of fully connected layers. An MLP is a general purpose NN with a single vector as input format. This makes an MLP less feasible for input data that has structure, like the pixels of a 2D image. An MLP has many parameters which involves longer learning. MLPs are stateless, they do not have a history and inputs are not ordered and are all processed independently. MLPs have no feedback connections when no outputs of the model are fed back into itself. If a NN contains feedback connections, they are called recurrent neural networks.

- Convolutional Neural Networks (CNN)
  CNN are specialized in processing image data. They contain one or more convolution layers consisting of convolution kernels. Each kernel may learn a different feature. CNN are stateless and feedforward.

- Recurrent Neural Networks (RNN)
  RNNs are specialized in sequential data. A datapoint in an RNN consists of a sequence of elements or vectors. An RNN assumes that the order of inputs has meaning. RNN are stateful, the input is part of a sequence. The RNN has recurrent layers with a hidden state. The memory lasts for the sequence and is reset for a new sequence. Long Short-Term Memory (LSTM) [Hochreiter and Schmidhuber, 1997] layers are most common.

The type of network depends on the characteristics of the environment. [Graesser and Keng, 2019] describe some guidelines for choosing a network family. The information position plays an important role. The authors identify three categories of POMDPs:

- Fully observable given partial history
- Fully observable given full history
- Never fully observable

Board-card games would fit best in the third category. The history of the played cards does not reveal the internal game state to a player. However, the board-card games have game positions with full observability. If the other players have no cards, which happens at the end of every round, all playing card locations are known.

The shape of the input layer is given by the environment's state space. The shape of the output layer depends on the action space and the algorithm. Examples are Q-values, action probabilities or action probabilities and V-values if the agent learns a Q-function, a policy or both a policy and a value function respectively.

Action and state representation are important factors in the success of RL. States and actions vary greatly from task to task, and how they are represented can strongly affect performance. According to [Sutton and Barto, 2018], in RL, as in other kinds of learning, such representational choices are more art than science. Important factors in the design of a state representation are completeness (sufficient information to solve the problem), complexity (available computational resources) and information loss (representation losses information).

2.6. Deep Q-Networks

DQN are a variant of Q-learning with non-linear function approximation by NNs. The function approximation of DQN should generalize from visited state-action pairs to similar states and actions. If there are parts of the state-action space that are very different from the experienced states and actions, then it is unlikely that the function approximation results in accurate Q-values. DQN have improved performance compared to plain Q-learning with large state spaces, but has decreased convergence and stability. Stability issues are caused by the correlation in sequential observations and actions. Multi-agent environments have more variance than single agent environments, which could influence the learning progress negatively. DQN can suffer from a maximization bias. The algorithm consists of a maximization over estimated values, which can lead to a positive bias if the estimated values are distributed around a low value. If a true value is zero, the maximum of the estimated values is relatively high.

Researchers have proposed several extensions to DQN to gain efficiency and stability. Notable optimisations are Double DQN [Van Hasselt et al., 2016], Prioritized Replay Buffer [Schaul et al., 2015], Dueling DQN [Wang et al., 2016] and Distributional DQN [Bellemare et al., 2017]. Combining these techniques resulted in better performance on Atari games [Hessel et al., 2018]. The domain of classic Atari 2600 games is a common testbed for reinforcement learning, including an interesting article of DeepMind [Mnih et al., 2013].

2.7. Deep Monte Carlo

DMC is MCM combined with NNs for non-linear function approximation. The DMC algorithm was successfully trained for Dou Dizhu, a Chinese card game that has similar characteristics as board-card games such as large action space, multi-player and imperfect information. DMC in this report refers to the implementation that was used for DouZero [Zha et al., 2021]. The DouZero implementation handles large action spaces with legal actions that vary significantly. The authors point out that DMC has three advantages over DQN:

- DMC is, as a MCM based implementation, not susceptible to overestimation.
- With DMC the episode length (length of play) does not impact the convergence.
- DMC's handling of large action spaces with varying legal actions is more efficient.

The authors have developed DouZero to handle a large set of actions with the legal actions varying from turn to turn. DouZero is MCM with function approximation by NNs, action encoding in matrices, and parallel processing. An agent consists of multiple actor processes and one learner process. The learner process maintains Q-networks for the states and updates the networks based on the data from the actor processes. The actors sample trajectories from the game implementation and calculate the rewards for every state-action value. The Q-networks of the actors are synchronized with the global networks periodically. There are three shared buffers between the learner and the actors. The parallel actors make the system scalable.

2.8. Other algorithms

In the dynamic research area of RL, there is a wide range of algorithms. Some notable algorithms that are not described here are:

- Asynchronous Advantage Actor-Critic (A3C) is a further development on such called Actor-Critic methods [Konda and Tsitsiklis, 2000]. The asynchronous nature leads to efficient learning on Atari games, TORCS Car Racing Simulator, MuJoCo Physics Simulator, and a 3D environment [Mnih et al., 2016].
- Counterfactual Regret Minimization (CFR) is an algorithm that is based on counterfactual regret, which exploits the degree of incomplete information [Zinkevich et al., 2007].
- PPO is a policy gradient method. PPO is among others successfully applied to arcade games [Schulman et al., 2017].
- Neural Fictitious Self-Play (NFSP) is was developed for imperfect information games. NFSP is based on DQN and performed better than DQN at poker [Heinrich and Silver, 2016].

2.9. RL IN GAMES

This section describes related research on the use of RL for the turn-based board and card games with imperfect information and cooperation between agents. RL proved very fruitful in game playing. Recent years have shown significant achievements in board games and card games like Chess, Bridge, Poker, and Go by applying self-play. Also in video games, RL has been successful. Just defining the rules of a game and running hours, days, or weeks of self-play turned out to be enough to beat the best humans and the most sophisticated computer programs, depending on the characteristics of the games. Each game has specific properties and dynamics that influence the success of RL techniques.

No prior work was found on board-card games like Keezen, Tock, Dog, or Sorry!. RL research has been applied to *Pachisi* and *Ludo*, simpler dice-based pawn racing games. In this research, the TD algorithm did perform well [Matthews and Rasheed, 2008], as did the Q-learning algorithm [Alhajry et al., 2012].

Table 2.1 shows an overview of researched games with some characteristics, the RL techniques, and the achieved level of play compared to humans. The games and the research are briefly described after the table.

Table 2.1: Overview of characteristics of games with the RL algorithm and achieved level compared to humans. The level of play is divided in amateur (experienced player) and top (potential world champion).

Game	Chance	Information	Players	Team	Algorithm	Level of play
Backgammon	Yes	Perfect	2	No	TD-learning, NN	Slightly below top
Big 2	Yes	Imperfect	4	No	PPO, NN	Better than amateur
Bridge	Yes	Imperfect	4	Yes	MCM, NN	Amateur
Clue(do)	Yes	Imperfect	3-6	No	Q-learning, NN	Amateur
Dou Dizhu	Yes	Imperfect	3	Yes	MCM, NN	Better than top
Go	No	Perfect	2	No	MCTS, NN	Better than top
Hanabi	Yes	Imperfect	3-5	Yes	?	Worse than amateur
Hearts	Yes	Imperfect	3-6	No	MCTS, NN	Amateur
Poker	Yes	Imperfect	2-10	No	MCM, CFR, NN	Better than top
Scrabble	Yes	Imperfect	2-4	No	B*, MCM	Better than top
Settlers of Catan	Yes	Imperfect	3-4	No	Q-learning, NN	?, beats best heuristic
Skat	Yes	Imperfect	3	Yes	MCM	Amateur
Ticket to ride	Yes	Imperfect	2-5	No	MCTS	Amateur

Backgammon is a two-player, perfect information board game with randomness. The program TD Gammon achieved in 1995 a strong level of play in Backgammon by applying the TD algorithm [Tesauro, 1994]. The game has a large branching factor which made brute force not feasible [Tesauro, 1995].

Big 2 is a four-player, imperfect information card game of Chinese origin. The PPO algorithm is used to train a NN, purely learning via self-play [Charlesworth, 2018]. The trained agents have beaten some amateur players and are not tested against expert players.

Bridge, also known as Contract Bridge, is a card game with randomness, imperfect information, team play, and bidding. Two pairs of two players play this game. As the subject of this research, the game is competitive and cooperative. At this moment, the top human bridge players are not outperformed by computers. One complex part of Bridge is the bidding phase [Yeh et al., 2018]. Other complicating factors are team play and imperfect information. The best performance is achieved by MCM with customizations, according to [Khemani and Singh, 2018]. Tournaments between bridge bots are organized since decades. Micro Bridge is the latest tournament winner (2019), using MCM.

Clue is a detective board game. Clue is the name of the game in the United States, the game is also known as Cluedo. Pawns or figures move through a house and find information in rooms to solve a murder. The game is based on a partial observable state. A combination of Q-learning and a Bayesian network gave good results, according to [Cai and Ferrari, 2008].

Dou Dizhu is a three-player card game with bidding that is popular in China. A combinational Q-learning agent outperformed other RL methods and plays well against humans [You et al., 2019]. In the summer of 2021, an even better performing solution was developed: DouZero [Zha et al., 2021]. DouZero implements DMC, which is described in Section 2.7.

Go is a two-player perfect information board game that has been a major challenge in AI research because of the enormous state space. In 2015 and 2016 AlphaGo was the first computer program to defeat professional Go players on a full-size board [Silver et al., 2016]. AlphaGo uses MCTS with a value network and a policy network. It was trained by human and computer play. AlphaGo contained Go-specific logic. AlphaGo Zero succeeded

AlphaGo in 2017. AlphaGo Zero was based on reinforcement learning by self-play only. It mastered the game of Go without applying human knowledge [Silver et al., 2017b]. AlphaZero is the successor of AlphaGo Zero. AlphaZero generalized the algorithm of AlphaGo Zero to the games of *Chess* and *Shogi*, a Japanese more complex chess variant [Silver et al., 2017a], [Silver et al., 2018].

Hanabi is a cooperative card game for 3 to 5 players. The company DeepMind proposed The Hanabi challenge in 2019 as a testbed game to drive research to cooperative and imperfect information games [Bard et al., 2020]. Regarding the current performances of agents on Hanabi, there is no preferable algorithm yet.

Hearts is a multi-agent, imperfect information card game. MCTS resulted in strong play according to [Sturtevant, 2008].

Poker is a family of multiplayer card games with randomness, imperfect information, bluffing, and bidding. The poker program Libratus has defeated top players in no-limit Texas hold 'em, one of the most popular variants of poker, in a tournament in 2017. Libratus applies three different algorithms. The blueprint strategy depends on Monte Carlo Counterfactual Regret Minimisation with self-play [Brown and Sandholm, 2018]. Another successful agent is DeepStack that uses deep counterfactual value networks [Moravčík et al., 2017].

Scrabble is a multiplayer, words-based board game. The program Maven plays at a superhuman level [Sheppard, 2002], with game-specific logics and the slightly improved B* tree search algorithm [Berliner, 1981] that could be considered as a truncated Monte Carlo simulation.

Settlers of Catan is a multiplayer, non-deterministic, turn-based, imperfect information game with trading between agents. Several MCTS-based agents have been developed, but apparently, a Q-value function-based recurrent neural network outperforms them [Konstantia et al., 2019].

Skat is a three-player, bridge-like game that was successful with the Recursive Imperfect Information Monte Carlo search algorithm [Furtak and Buro, 2013].

Ticket to ride is a two to five-player game with randomness, pathfinding, and imperfect information. An MCTS agent performed well according to [Huchler, 2015].

3

CASE STUDY BOARD-CARD GAMES

This chapter describes the board-card games and their characteristics. The game Keezen, as subject of the case study, is described in detail.

3.1. BOARD-CARD GAMES

The board-card games of this research are marble racing games. Players run their four marbles from the start to the finish fields. The goal is to be the first to get all marbles in the finish area.

A setup with four players is the most common. The players across the table form teams. Each player starts with their own four marbles. When a player's marbles have reached the finish fields, the player continues with the marbles of the teammate. The first team to run all marbles to the finish fields wins.

Marbles can pass, hit or block other marbles depending on the location on the board. At every turn, players make decisions, balancing between supporting the team's marbles and thwarting opponent marbles.

The movement of marbles on the board is steered by the functions of playing cards. The playing cards bring imperfect information to the game. Every turn, a player throws a card with a function and changes the position of one or more marbles. For example, a three (of any suit) allows a marble to run three steps,

3.2. BOARD-CARD GAME VARIANTS

The most notable board-card games are *Tock (or Tuck)*[1], *Dog*[2], *Sorry!*[3] and *Keezen*[4]. Tock originates from Canada, Dog is particularly played in Swiss and Germany, Sorry! in England and Keezen in the Netherlands. The foundation of these games is analogous, but there are variations in boards, rules, cards, and actions:

- *Keezen* is played by four players in two teams with a French deck of cards without the jokers. The board has 96 fields: four wait fields and four finish fields per player and a round of 64 fields. The rules of Keezen are described in detail in Section 3.3.

[1] https://en.wikipedia.org/wiki/Tock
[2] https://de.wikipedia.org/wiki/DOG_(Spiel)
[3] https://en.wikipedia.org/wiki/Sorry!_(game)
[4] https://nl.wikipedia.org/wiki/Keezen

- *Tock* is played with the same stock of cards as Keezen, but the actions per card differ. The 5 moves the marble of any player 5 steps. The 7, like Dog, is distributed over all marbles. Black Jacks swap marbles, while red Jacks move a marble 11 fields. The King is used as a starter or to run 13 fields.

- *Dog* is played on almost the same board as Keezen, with more cards and a different number of cards per round. Teammates exchange one card at the start of each round. The cards are different. The King moves 13 fields, the 7 splits 7 steps over all the players' marbles, and the 4 moves forward or backward. Marbles may have to pass the finishing area to run another round. The board has the start field located in front of the finish fields.

- Two main variants of *Sorry!* exist. The classic variant involves a proprietary deck of 45 cards. The 6 and 9 are left out and four Sorry! cards are added. The actions of cards are slightly different from those of Keezen. A Sorry!-card sends a marble of an opponent back to the start field. In 2013, a Sorry!-version was introduced that is played with only three marbles per player and introduced a notion of Fire and Ice to change rules for specific marbles.

3.3. Rules of Keezen

This research follows the Keezen rules as defined for the national Keezen championship of The Netherlands [5]. These rules are chosen for clarity and in the belief that these are the standard rules.

Figure 3.1 shows the Keezen-board. The figure shows definitions for fields and player positions as used further in this report. The forward running direction is clockwise.
A *step* means moving a marble to an adjacent field.
Starting means putting a marble from a waiting field to the start field.
Finishing means running a marble to the furthest finish field.
Hitting means landing a marble on a field that is already occupied by a marble (if allowed), sending the other marble back to a waiting field.
Switching means swapping two marbles.
Blocking means marbles can't pass another marble. A marble on a field of the same color (start or finish field) is blocking other marbles.
Playing a card means throwing the card in the middle of the board and executing one of the defined actions of the card.

The marbles are moved by the functions of 52 playing cards from a classical deck without the jokers. Each type of card has one or two actions (independent on the suit):

The flow of the game:

- The game starts with all marbles at the waiting fields.

- Each round, the dealer gives cards from the stock to the players. In the first round five cards, then four cards, and again four cards per player. Every three rounds, the stock is reset. This means all cards are in the stock and shuffled. Note that at this moment the game state is entirely known to all players.

[5] https://nk.keezbord.nl/officiele-spelregels

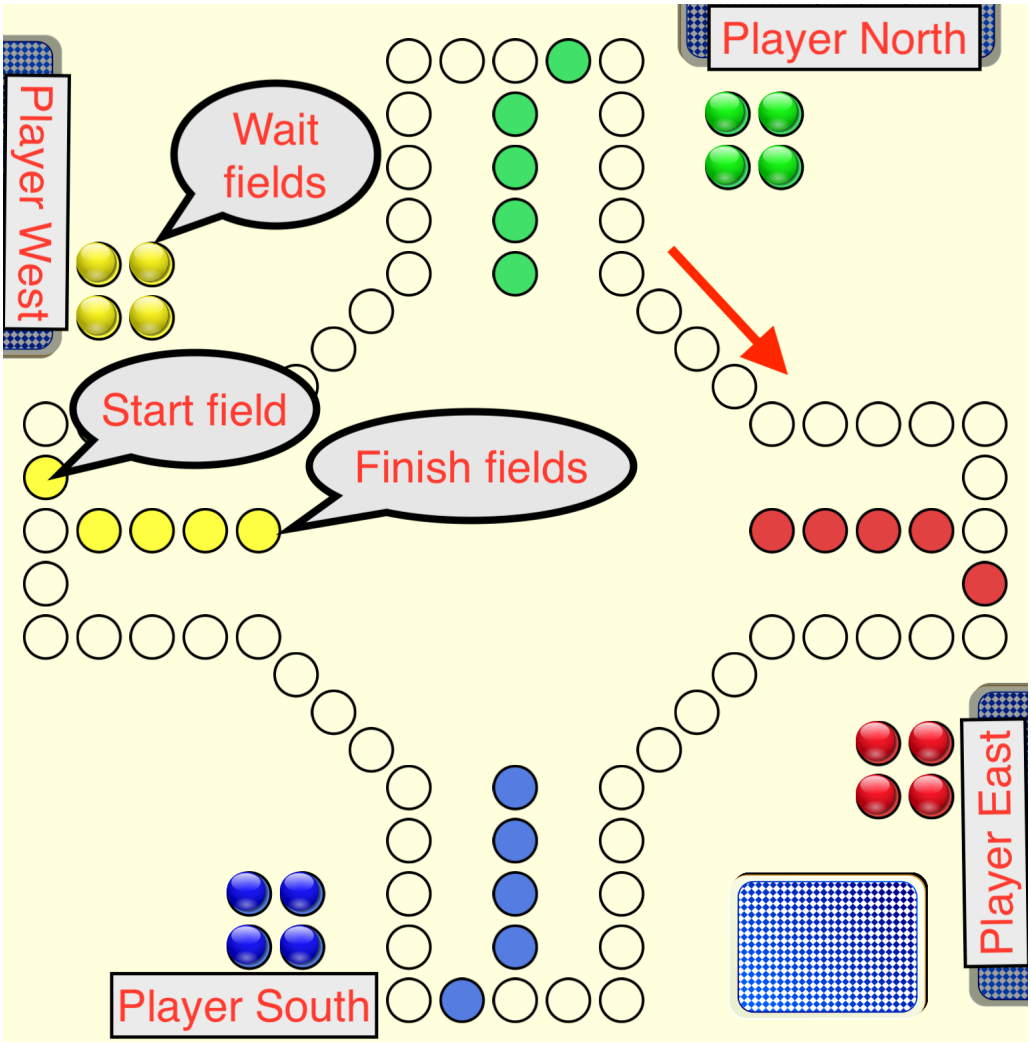

Figure 3.1: Keezen board with all marbles at the waiting fields.

Table 3.1: The cards and their functions in Keezen.

Card	**Function(s)**
Ace	Run 1 step or set up a marble (starting).
2, 3, 5, 6, 8, 9, 10	Run 2, 3, 5, 6, 8, 9 or 10 steps respectively.
4	Run 4 steps backward. A marble cannot run backward directly into or out of the finish area. A marble can run backward past the finish area and finish forward in the next move.
7	Run 7 steps in total with one or two marbles.
Jack	Swap own marble with other players marble (not allowed from the start fields).
Queen	Run 12 steps.
King	Set up a marble (starting).

- The player next (clockwise) to the dealer starts. If a player has a starter (Ace or King), then a marble can move to the start field to start running to the finish area.
- If the round is finished, then the next (clockwise) player becomes the dealer.
- If a player finishes all four marbles, then the player continues with the marbles of the teammate. The action of card 7 may be split between the last finishing marble and one marble of the teammate.
- If a team finishes all eight marbles, then the team wins and the game is over.

Rules:

- Each turn, the execution of an action is mandatory. If a player has no valid action to execute with his cards, then the player discards all cards and waits for new cards in the next round.
- If a marble ends up at a field that already contains a marble, then the other marble is hit and returned to a waiting field. It does not matter if it is your marble or not.
- A marble on a field with the same color (start or finish field) blocks other marbles and can not be hit or swapped. Posting a marble on the start field effectively opposes other marbles. Moving marbles into the finish area might require precise maneuvering because they cannot pass each other.

Several direct variations on Keezen exist. Most consist of small variations in the rules, regarding the preconditions for swapping marbles and running backward. The game Kruiskeezen has an extra cross in the center of the board to support shortcuts. It is likely that the findings of this research are also valid for small variations in the Keezen rules.

3.4. Game characteristics

3.4.1. Definition

In science, it is commonplace to identify, describe and categorize subjects of research. Observed similarities lead to an arrangement in groups. An obvious example is the detailed classification system for plants and animals.

Work has been done for classification systems for games [Elias et al., 2012]. Considering the short history and current dynamic nature of the research area, one cannot expect a classification system in place like it is in Biology. Even if it was there already, one might still wonder how to apply the characteristics of the hybrid board-card games.

It is hard to define a meaningful classification system for games in general. There are many different games with many different features or characteristics. It is even hard to come up with a definition of a game [Elias et al., 2012]. One can argue that the definition should cover everything from word search puzzles, to video games, to professional soccer.

There are at least two good reasons to analyze the characteristics of board-card games:

- Compare games
  Classification helps to compare the characteristics of a game with other games.

- Match RL algorithms to games
  The characteristics could help to determine which (RL) techniques are effective to develop game-playing agents.

These reasons make determining the characteristics of board-card games beneficial. Inspired by [Elias et al., 2012], based on other games literature, and based on the experiences in this research, we come to the following relevant characteristics of board-card games:

- **Number of players**
  The number of players to play the game.
- **Roles of players**
  A game can have multiple roles for the players. If so, agents have to learn different strategies for those roles.
- **Teamplay**
  Teamplay implies that one or more players work together.
- **Turn-based or real-time**
  Games can be turn-based or real-time (sequential or parallel). Most board and card games are turn-based. Players execute their moves sequentially.
- **Length of play**
  The length of play is an indicator of the number of actions to be executed during an episode (gameplay from start to finish) of a game. Turn-based games can be measured in a turn or move number. In Keezen each action or turn of a player is counted as a move. Throwing all cards is a move and skipping the turn after throwing all cards is not counted as a move.
- **Chance events**
  Chance events bring randomness into the game. Throwing dice or dealing cards are chance events that control the state of a game that is otherwise manipulated by the actions of agents. This is just a boolean indicator without an indication to what extent a game is random.
- **Information position**
  In a perfect information game, all players know the full state of the game. There is no hidden information. In an imperfect information game, agents miss information to make their decisions. The information position influences the level of assurance of decision-making by agents. This is just a boolean indicator. It would be interesting to have a factor that indicates to what extent a game has hidden information. This factor could be calculated for a game position or over the average of all moves of an episode.
- **Branching factor**
  In a generic sense, the branching factor is the number of children at a node. If each node is a valid game position then the branching factor at that point is equal to the number of valid moves (sub-nodes). The average branching factor is a characteristic that indicates the exponentially increasing number of nodes of a game. The branching factor is an important indicator of the required resources to learn to play a game

well. In general, the games that are subject to glsAI research have a branching factor and length of play that lead to a combinatorial explosion that is not feasible to solve by brute force.

- **Action space**
  The action space comprises all available unique actions that the game supports. The legal actions at a game position form a subset of the action space or, if all actions are valid, a set equal to the action space. The value for the action space size is Low (<=10), Medium (>10 and <=50), or High (>50).

- **Combinational actions**
  This characteristic expresses the dynamic nature of legal actions as part of the action space. An indication can be calculated by dividing the number of legal actions by the size of the action space. One can calculate this value as an average over a full episode. This indication is satisfactory to the purpose of this research although it uses the average number of legal actions and ignores the distribution of legal actions in the action space.

  The value for the combinational actions is High (<=0.33), Medium (>0.33 and <=0.66) or Low (>0.66). These values seem reasonable based on the values of other games [6]. Games with allowed actions that are more or less independent of the state, such as poker, have a value Low for combinational actions. All board-card games have a value High for combinational actions.

- **State space**
  The state space is the set of all possible states of a game. The size of the state space is an indicator of the required computational resources like the branching factor. The value for the state space is Small (<=10^5), Medium (>10^5 and <=10^{10}), or Large (> 10^{10}). In RL research many games have large state spaces.

- **Outcomes**
  There are competitive and non-competitive games. Most games end in a win, a loss, or a draw. The board-card games end with a win or a loss for the teams.

- **Symmetry**
  Symmetry means that a game is (partially) invariant under some transformations[7]. Symmetry in games regarding roles, states, or actions opens possibilities to optimize search algorithms [Schiffel, 2010]. For example, if multiple board positions are identified as the same but mirrored or rotated, that reduces the search tree and limits the calculations required. Board-card games are symmetric regarding the roles of players, the state of the board, and actions.

3.4.2. VALUES

In this section we estimate the values for the game characteristics for the board-card games (Dog, Keezen, Tock and Sorry!).

The values of the following characteristics are estimated:

[6] https://en.wikipedia.org/wiki/Game_complexity
[7] https://en.wikipedia.org/wiki/Symmetry

- **Length of play**
  The average length of play of Keezen is determined by counting during a tournament of 500 matches with rule-based clients. The result: 207 moves on average. The minimal number of moves of a full game was 103 and the maximum number of moves of a full game was 342. The lengths of plays of other board-card games are estimated to be similar.

  The possible values for the length of play are Low (<=50), Medium (>50 and <=100), and High (>100). These values seem reasonable based on the values of other games. All board-card games have a high length of play.

- **Branching factor**
  The average branching factor of Keezen is determined by counting during a tournament of 500 matches with rule-based clients. The result: 6. This means that on average a player has 6 possible actions to choose from in a state s. The branching factors of other board-card games are estimated to be slightly higher (Dog), equal (Tock), and less than Keezen (Sorry!) based on the number of cards in the hands and the number of functions of the cards.

  The possible values for the branching factor are Low (<=10), Medium (>10 and <=100), and High (>100). These values seem reasonable based on the values of other games [8]. The branching factors of Tick-tac-toe, Chess, and Go are 4, 35, and 250 respectively. All board-card games have a low branching factor.

- **State space**
  The state-space of Keezen consists of the positions of the marbles on the board and the cards. A rough estimation:

  - Board:
    The total number of 16 marbles of the four players can occupy 72 possible fields. Ignoring overlap: $(72x71x70x69)^4 = 3.7x10^{29}$
  - Cards:
    There are six positions for each card: four players, stock or played: $6^{52} = 2.9x10^{40}$
  - Total:
    Board positions x Card positions = $1.1x10^{70}$, hence: Large. To compare: chess has $2x10^{40}$ positions according to [Steinerberger, 2015]. The other board-card games have large state spaces too. Dog and Tock have a similar board with 16 marbles. Sorry! has 12 or 16 marbles on a board with fewer fields. The number of cards of Dog is more than Keezen (102), Tock has the same number of cards and Sorry has fewer cards (45). These calculations are disputable because of multiple reasons (for example overlap of marble positions and equal card ranks are ignored), but the conclusion stands that the state spaces are large.

Table 3.2 shows the characteristics of the board-card games, and two other games to compare with, filled in. The values for Chess and Go are from Wikipedia [8].

The analysis of the characteristics of the board-card games shows that all board-card game have the same values for all characteristics. This suggests that the conclusions regarding the case study are valid for all board-card games.

[8] https://en.wikipedia.org/wiki/Game_complexity

Table 3.2: Values of the characteristics of the board-card games and two other games to compare with.

	Dog	**Keezen**	**Tock**	**Sorry!**	**Chess**	**Go**
Number of players	4 - 6	4 - 6	4 - 6	4	2	2
Roles of players	Equal	Equal	Equal	Equal	Equal	Equal
Team play	Teams of 2	Teams of 2	Teams of 2	Teams of 2	No	No
Turn-based?	Yes	Yes	Yes	Yes	Yes	Yes
Length of play	High (207)	High	High	High	Medium (70)	High (150)
Chance events	Yes	Yes	Yes	Yes	No	No
Information pos.	Imperfect	Imperfect	Imperfect	Imperfect	Perfect	Perfect
Branching factor	Low	Low (6)	Low	Low	Medium (35)	High (250)
Action space	High	High (271)	High	High	High	High
Comb. actions	High	High	High	High	High	High
State space	Large	Large (10^{70})	Large	Large	Large (10^{44})	Large (10^{170})
Outcomes	Win-loss	Win-loss	Win-loss	Win-loss	Win-draw-loss	Win-loss
Symmetry	Yes	Yes	Yes	Yes	Yes	Yes

On a higher level, the results could be generalized, with reservations, to situations outside the game world. All situations have their characteristics. The characteristics of Section 3.4 will not fit completely in other situations, but it is a start. There are situations of imperfect information, team play, and large action space. The research into games with these characteristics can be beneficial to those situations, probably in real-life.

3.5. SKILL AND CHANCE

The characteristics of board-card games influence the expectations one can have of the learning outcomes. The *skill-factor* for a game type is a factor that indicates the extent to which skill determines the outcome of a game. Game characteristics like imperfect information, chance events, branching factor, and length of play steer to a lesser or greater extent the determining factor of choices during a game. In Chess, with perfect information, no chance events, and a higher branching factor, the skill of the player has a greater influence on the outcome than it has in board-card games with shuffled cards and imperfect information. This goes two ways. Bad moves can turn out to be good moves and brilliant moves can turn out to be bad moves.

Skill and chance in games have been researched for the legal distinction between games of skill and gambling. In [Borm and van der Genugten, 2001] the relative skill-factor is defined as:

$$Skill = \frac{\text{Learning effect}}{\text{Learning effect+Random effect}} = \frac{\text{Result Real-average-player - Result Beginner}}{\text{Result Virtual-average-player - Result Beginner}} \quad (3.1)$$

The value of the skill factor lies between zero (pure chance) and 1 (pure skill). The *learning effect* indicates the difference in result that comes from learning better moves. The three types of players that are defined to measure skill are:

- *Beginner*: somebody who has just mastered the rules of the game and plays in a naive way.
- *Real-average-player*: represents the vast majority of players.
- *Virtual-average-player*: an average player who is told the outcome of chance events before an action is chosen.

The skill-factor influences the expectations we may have concerning the gameplay of agents. In games of Chess or Go, with high skill-factor values, we can enjoy brilliant moves that determine the outcome of matches. In addition to brilliant moves, one can also talk about a playing style. AlphaZero brought excitement in the world of Go and Chess with tactical novelties, stunning moves, and a distinct playing style [Sadler and Regan, 2019]. Not surprisingly, brilliant moves in a game with chance events like Ludo will not lead to the same excitement. To a greater or lesser extent this will also apply to board-card games.

The skill-factor of board-card games is unknown. The outcomes of board-card games depend not solely on the skill of the players, but it is hard to determine to what extent chance and skill influence the outcomes of board-card games. In general, the role of skill in games is determined relatively by considering the results of different types of players ([Borm and van der Genugten, 2001]) or the rating of players ([Duersch et al., 2020]). Unfortunately, there are no board-card game references to compare. To this research, we just keep in mind that the expectations regarding brilliant play in board-card games are lower than games like Chess and Go. Further research into the relation of the characteristics of games and the role of skill and chance in the context of RL might be interesting.

4

METHOD

This chapter describes the technical implementations and the choices we made. All implementations are available for future research.

An RL-framework provides base functionalities for RL experiments. A custom game environment runs episodes of the Keezen game implementation with specific state and action model designs. RL-agents execute actions following algorithms in the environment to learn how to play the game. The training progress is measured by playing auto-play tournaments against randomly playing agents and rule-based agents, and by analyzing the gameplay manually. We will cover these subjects in more detail in this chapter.

4.1. SCOPE

The research focuses on the four-player composition with two teams of two players, which is the most common way to play board-card games. Other configurations, with more or fewer teams, bring other game dynamics. Two equal agents (two agents from the same training stage) team up. Combining different agents with different strategies in one team would add complexity and is out of scope.

Following 1.3, the game Keezen will be implemented and experimented with as a case study. The game is learned from zero, without initial knowledge. There is no expert input to the learning process. The imperfect information position is respected without exception. Each agent is aware of its own cards and the played cards but unaware of other cards.

4.2. RL FRAMEWORK

Several frameworks aimed at researching game agents exist, most notably OpenAI Gym[Brockman et al., 2016][1], OpenSpiel[Lanctot et al., 2019][2] and RLCard [Zha et al., 2019][3].

- **OpenAI Gym**
  OpenAI Gym is a well-known general-purpose RL research framework. OpenAI is a great source of information, tutorials, algorithms, and environments (Cartpole) to

[1] https://gym.openai.com
[2] https://deepmind.com/research/open-source/openspiel
[3] http://rlcard.org

play with. OpenAI's Spinning Up[4] is a recommended resource. OpenAI Gym is also used for research on (video) games.

- **OpenSpiel**
  OpenSpiel is a framework for RL research on games. DeepMind is the company behind OpenSpiel that has made notable contributions to RL and games in recent years. Among others, they have made headlines with their achievements in Chess and Go with MCTS based algorithms (AlphaGo and AlphaZero [Silver et al., 2017a]).

- **RLCard**
  RLCard is a framework that aims at RL research on card games. RLCard is developed at Texas A&M University.

All three frameworks were explored. OpenAI Gym is great, but not specifically made for games. For example, it does not support multi-agent games out of the box. This research could have been executed with OpenAI Gym, but the generality made it less suitable than the two game-oriented frameworks. OpenSpiel and RLCard are both very capable frameworks to research board-card games. Both support multi-agent games and there was no clear preference for one over the other. The learning curve of RLCard is less steep than OpenSpiel. The ease of getting started with a custom game was ultimately the deciding factor. The case study experiment was running with DQN. RLCard has implementations of other algorithms such as NFSP.

4.3. GAME IMPLEMENTATION

A custom game can be added to RLCard by implementing the game logic, wrapping it with an environment, and then registering it to the framework. The game and environment use the action model and the state model that are the subjects of the next two sections.

The Keezen implementations were taken from an iOs app (that we developed before) and migrated from Swift code to Python code. The game logic differed from the suggested RLCard design. The adapter class `KeezenGameAdapter` overcomes the incompatible interface.

Environment

The environment `KeezenEnv` class extends the RLCard `Env` class. This class is registered by the RLCard framework as 'keezen'. The `KeezenEnv` class has the following method implementations, imposed by RLCard:

- `reset()` returns the state (observation o) of the game and the index of the player to move.
- `step(action)` performs an action (a) in the game. The action is an index of the array with 271 actions in the GameActions class.
- `_extract_state(state)` extracts the state (observation o) as an array or matrix, formatted as input for the algorithm. See the state model description.
- `get_payoffs()` returns the rewards ($R_a(s, s')$) as an array with for each player the result. 1 for win and 0 for loss.

[4] https://spinningup.openai.com

- `_decode_action(action)` returns the action string representation for an action index. See the action model description.
- `_get_legal_actions()` returns the legal actions for the current state as an array of integers.

Game
The central classes of the game implementation are `Game` and `KeezenGameAdapter`. `Game` implements the game logics without keeping a game state. This class refers to other classes like `Board`, `Marble`, `Field`, `Card`, `Player`, `Move`, and `GameState`. The design deviates from the structure that is suggested in RLCard documentation with `Round`, `Dealer`, `Judger`, and `Player`. That is ok as long as the `Game` interface of RLCard is respected. We choose to preserve the original design as it has already been extensively tested, even though it was with another programming language.

Methods of the Keezen `Game` class are:

- `init_game()` initializes the game and returns an initial game state (observation o) and the player to move.
- `step(move, state)` performs a move (action a) in the game. Returns the game state (o), rewards (R) for all players, and if the game finished.
- `is_over(state)` returns if the game is over and the rewards (R) for all players.
- `render()` prints the game position to the standard out.

The `KeezenGameAdapter` class was introduced to convert the incompatible interface to the `Game` interface of RLCard. The adapter class wraps the Keezen `Game` implementation class and calls its methods, while maintaining the game state. The methods of the `KeezenGameAdapter` are:

- `init_game()` initializes the game and returns an initial state and the index player to move.
- `step(action)` performs an action in the game. Returns the game state (observation o) and the player to move next.
- `get_state(player_id)` returns the game state (observation o) from the perspective of a player.
- `get_player_id()` returns the index of the player whose turn it is.
- `get_player_num()` returns the number of the players: four.
- `is_over(state)` returns if the game is over.
- `render()` prints the game position (s) to the standard out. Used during the manual analysis.

Register
The Keezen game is registered to the RLCard framework by setting the name and entry point in `rlcard/envs/__init__.py`.

4.4. STATES AND ACTIONS

The state model represents the state of the game as observed by the players. Each player has its own unique observations of the game. The incomplete information position means the observations do not reveal the cards in the hands of other players. The other players are denoted as Opponent A, Teammate and Opponent B.

Figure 4.1 shows an example of an observed game state. The observation consists of entirely observable board positions and partially observable card positions.

The board position is described by the field positions of the marbles. The fields are identified by unique numbers. The marbles are identified by the color and a unique number with a 0 referring to the furthest marble and a 3 to the last marble.

The cards in the hand of the player and the played cards in the middle of the board are observable. The other cards, in the hands of other players and in the stock are not observable. The game state also includes an indication which player's turn it is and the move number.

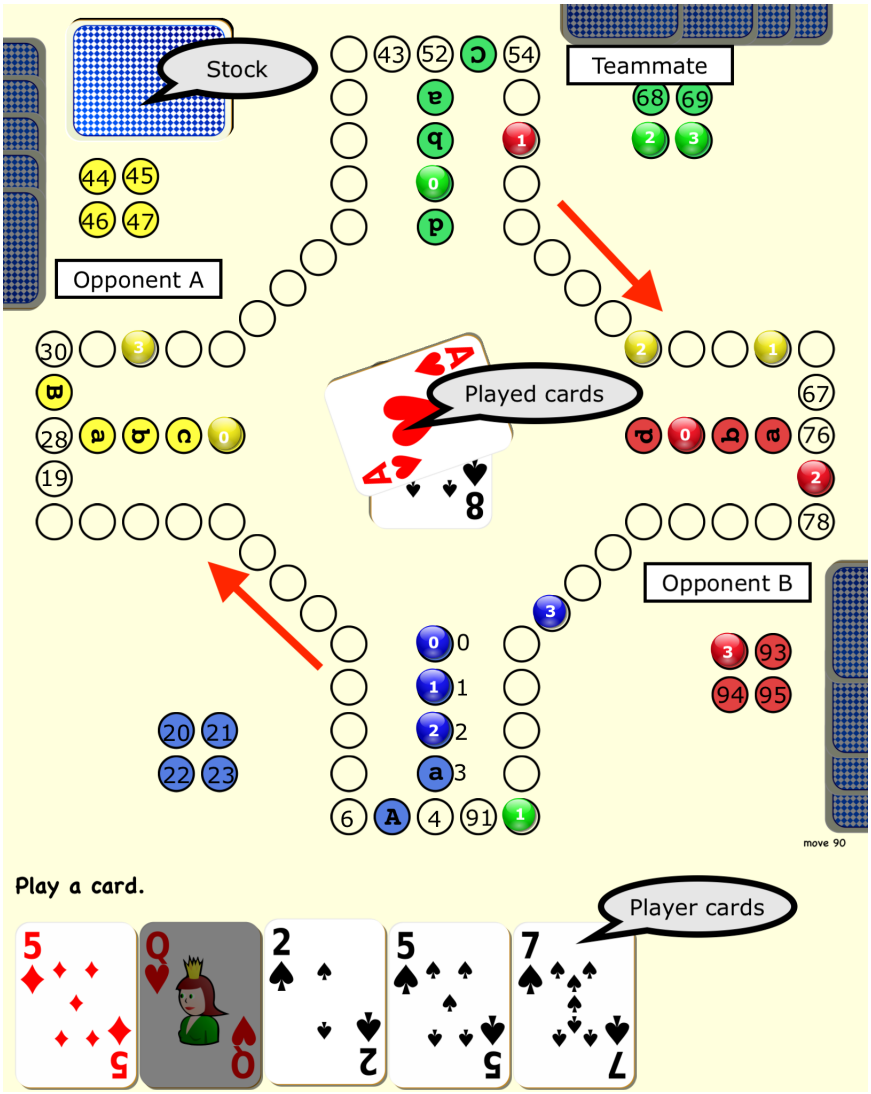

Figure 4.1: Example of a Keezen game state observed by player south. Note the numbering of the fields (0-95) and the marbles (0-3 for each color).

The cards in the hand of the player in a state s determine the allowed actions of a player. In general, an action a consists of a card that moves one or two marbles. In Figure 4.1 player South may choose to throw card Spades 2 and run with the blue marble with number 3 from field 85 to field 87. Player South may instead choose to throw a card 5 (Diamonds or Spades) to run the marble to field 90, sending the green marble (1) of his teammate back to the waiting fields. Note that each action of a player is fully observable by the other players. The card becomes a played card that all players can see.

The next two subsections describe the representations of the state model and the action model.

4.4.1. Observation model

The observation model that we also less-precisely call the state model, encompasses all possible observations of the game. Refer to Figure 4.1 for the field and marble numbering. The cards are identified by unique numbers from 0 to 12, corresponding with the ranks from Ace to King.

The state model is represented by a matrix with 4 rows with only 0 and 1 values that represent the observations to the players. The state matrix consists of the following blocks:

- **Move player [0]**: One column with a 1 for the player that has the turn, 0 otherwise.
- **Player cards [1-13]**: 13 columns with the current player's cards.
  The indexes of the 13 columns correspond with the ranks: Ace, Two, Three, Four, Five, Six, Seven, Eight, Nine, Ten, Jack, Queen, and King. The suits play no role. Multiple cards of the same rank result in multiple 1 values in one column. The player in figure 4.1 has the cards Diamonds 5, Hearts Queen, Spades 2, Spades 5, and Spades 7. This results in the matrix to represent the cards:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

  In which the columns refer to the following cards:

$$\begin{bmatrix} A & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & J & Q & K \end{bmatrix}$$

- **Played cards [14-26]** 13 columns with the played cards.
  Identical representation model as with the current player's cards.
- **Board [27-122]** 96 columns with the board state.
  The board state consists of rows of 96 columns. Each column matches a field as seen from the perspective of the player. The field numbers are shown in Figure 4.1. Each row contains four 1 values that indicate the field positions of the marbles of one player (and 88 zero's). The first row contains the players own marble positions and the next rows contain the marble positions of the other player in clockwise direction.
- **Card counts [123-127]** 5 columns representing the number of cards in the hands of the players. Each card of a player is represented by a 1 from left to right. For example,

this matrix indicates from top to bottom zero, two, three, and three cards in the hands of the players:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \end{bmatrix}$$

The matrix is flatten to an array (the rows are put one after the other) to confirm to the RLCard framework DMC interface. The model is used on both DQN and DMC.

We experimented with several variations on the observation model. These experiments are described in Section 7.2.6. We experimented with and without a fifth row that contains a 1 if the column contains a 1 in any of the other 4 rows and a 0 otherwise, we experimented with and without the block that holds the card count, and we experimented with two different perspectives: *player perspective* and *generic perspective*. The difference between these perspectives is in the order of the rows with information that is known to all players: the move player, the positions of marbles on the board, and the number of cards in the hands of the players.

We did not experiment with an explicit history of actions in the model like DouZero [Zha et al., 2021]. In DouZero, the most recent 15 actions are encoded and fed into an LSTM and then from the last cell into the NN. Adding a history with the actions per player during the three rounds of the stock might improve the performance.

4.4.2. Action model

The action model encompasses all possible actions of a player during the game. The action model uses the same field, marble, and card numbering as the state model defined in the previous section. Every action specifically defines which marble the action is applied to. Running ten fields with the first marble is not the same action as running with the second marble. Analysis and tests resulted in a set of 271 unique actions. These action definitions are in `GameActions.ALL_ACTIONS_271`. It seems that an action space with less than these actions is at the expense of information or adds more complexity.

The action model is formed by an array of 271 elements that is used by the NN. Like the state model, the array contains just 0 and 1 elements. Each index of the array represents an action that is in an array with unique string representations of actions (or moves). The string representations are mapped to the move objects that the game implementation receives from the agents to update the game state.

The string representations of actions are structured by action types and marble references. There are seven different action types with unique two character long identifiers:

- Run: **RU**
- Split: **SP**
- Start: **ST**
- Switch: **SW**
- Throw cards: **TC**
- Deal: **DL**

- No move: **NO**

The first four action types require field and marble references to uniquely identify the movement of marbles on the board. A 'P' refers to a marble of the current player, 'T' to a marble of the teammate, and 'A' and 'B' to the marbles of the two opponents. The string representation starts with the move type followed by the marbles to move. The actions are defined as follows:

- **Run actions (40x)**: "RU$\langle steps \rangle$ $\langle marble-id \rangle$"
  Run 1 to 12 fields (excluding 7 and 11) with marble 0, 1, 2 or 3 (P0 is the furthest marble, P3 is the latest).
  Examples: "RU08P0": Run 8 steps with the furthest marble (P0), "RU08P1": Run 8 steps with the second-placed marble (P1).

- **Split actions (172x)**: "SP07$\langle marble-id \rangle$" and "SP07$\langle 1^{st}-marble-id \rangle$ $\langle steps \rangle$ $\langle 2^{nd}-marble-id \rangle$"
  Run 7 steps with one or two marbles.
  Example: "SP07P32P2": Run 2 steps with the last marble (P3) and 5 steps (the remainder) with the third marble (P2).
  There are 4 shorter split actions, which are just moves with a single marble: "SP07P0", "SP07P1", "SP07P2", and "SP07P3". The other split actions that move two marbles, do 1 to 6 steps by one of the four marbles, with the remaining steps moved by one of the three other own marbles or one of the four marbles of the teammate. This results in a total of $4 + 6 \times 4 \times 7 = 172$ split actions.

- **Start actions (8x)**: "ST01P0" to "ST01P3" (Ace) and "ST13P0" to "ST13P3" (King)
  Start with the first marble (P0) or the next marbles (P1, P2, and P3).
  Example: "ST01P0": Start with the first marble (P0).

- **Switch actions(48x)**: "SW11P0A0" to "SW11P3B3":
  Swap an own marble with another players marble.
  Example: "SW11P3A2": Swap the last marble (P3) with the third marble of opponent A (A2).

- **Special actions (3x)**: "TC" (Throw cards), "NO" (No move) and "DL" (Deal)
  The actions "NO" and "DL" are the only actions that are no moves in the game. "NO" is a no-operation and "DL" is a deal event.

This results in the set of all possible actions that is contained in Table 4.1. The action model itself is an array of one row that contains only 0 or 1 values for indexes that match actions in this table.

The idea behind the action model was to balance between unique actions and the number of actions. Another manner to define the action model is by defining the actions per field which would have resulted in more than 6000 action definitions. According to [Zha et al., 2021], DMC supports thousands of actions, which might open the door for field-bound actions.

Table 4.1: All possible actions from the action model.

Action type	**Actions**
No action	'NO'
Deal	'DL'
Run (40)	'RU01P0', 'RU01P1', 'RU01P2', 'RU01P3', 'RU02P0', 'RU02P1', 'RU02P2', 'RU02P3', 'RU03P0', 'RU03P1', 'RU03P2', 'RU03P3', 'RU04P0', 'RU04P1', 'RU04P2', 'RU04P3', 'RU05P0', 'RU05P1', 'RU05P2', 'RU05P3', 'RU06P0', 'RU06P1', 'RU06P2', 'RU06P3', 'RU08P0', 'RU08P1', 'RU08P2', 'RU08P3', 'RU09P0', 'RU09P1', 'RU09P2', 'RU09P3', 'RU10P0', 'RU10P1', 'RU10P2', 'RU10P3', 'RU12P0', 'RU12P1', 'RU12P2', 'RU12P3'
Split (172)	'SP07P0', 'SP07P01P1', 'SP07P01P2', 'SP07P01P3', 'SP07P01T0', 'SP07P01T1', 'SP07P01T2', 'SP07P01T3', 'SP07P02P1', 'SP07P02P2', 'SP07P02P3', 'SP07P02T0', 'SP07P02T1', 'SP07P02T2', 'SP07P02T3', 'SP07P03P1', 'SP07P03P2', 'SP07P03P3', 'SP07P03T0', 'SP07P03T1', 'SP07P03T2', 'SP07P03T3', 'SP07P04P1', 'SP07P04P2', 'SP07P04P3', 'SP07P04T1', 'SP07P04T2', 'SP07P04T3', 'SP07P05P1', 'SP07P05P2', 'SP07P05P3', 'SP07P05T0', 'SP07P04T0', 'SP07P05T1', 'SP07P05T2', 'SP07P05T3', 'SP07P06P1', 'SP07P06P2', 'SP07P06P3', 'SP07P06T0', 'SP07P06T1', 'SP07P06T2', 'SP07P06T3', 'SP07P1', 'SP07P11P0', 'SP07P11P2', 'SP07P11P3', 'SP07P11T0', 'SP07P11T1', 'SP07P11T2', 'SP07P11T3', 'SP07P12P0', 'SP07P12P2', 'SP07P12P3', 'SP07P12T0', 'SP07P12T1', 'SP07P12T2', 'SP07P12T3', 'SP07P13P0', 'SP07P13P2', 'SP07P13P3', 'SP07P13T0', 'SP07P13T1', 'SP07P13T2', 'SP07P13T3', 'SP07P14P0', 'SP07P14P2', 'SP07P14P3', 'SP07P14T0', 'SP07P14T1', 'SP07P14T2', 'SP07P14T3', 'SP07P15P0', 'SP07P15P2', 'SP07P15P3', 'SP07P15T0', 'SP07P15T1', 'SP07P15T2', 'SP07P15T3', 'SP07P16P0', 'SP07P16P2', 'SP07P16P3', 'SP07P16T0', 'SP07P16T1', 'SP07P16T2', 'SP07P16T3', 'SP07P2', 'SP07P21P0', 'SP07P21P1', 'SP07P21P3', 'SP07P21T0', 'SP07P21T1', 'SP07P21T2', 'SP07P21T3', 'SP07P22P0', 'SP07P22P1', 'SP07P22P3', 'SP07P22T0', 'SP07P22T1', 'SP07P22T2', 'SP07P22T3', 'SP07P23P0', 'SP07P23P1', 'SP07P23P3', 'SP07P23T0', 'SP07P23T1', 'SP07P23T2', 'SP07P23T3', 'SP07P24P0', 'SP07P24P1', 'SP07P24P3', 'SP07P24T0', 'SP07P24T1', 'SP07P24T2', 'SP07P24T3', 'SP07P25P0', 'SP07P25P1', 'SP07P25P3', 'SP07P25T0', 'SP07P25T1', 'SP07P25T2', 'SP07P25T3', 'SP07P26P0', 'SP07P26P1', 'SP07P26P3', 'SP07P26T0', 'SP07P26T1', 'SP07P26T2', 'SP07P26T3', 'SP07P3', 'SP07P31P0', 'SP07P31P1', 'SP07P31P2', 'SP07P31T0', 'SP07P31T1', 'SP07P31T2', 'SP07P31T3', 'SP07P32P0', 'SP07P32P1', 'SP07P32P2', 'SP07P32T0', 'SP07P32T1', 'SP07P32T2', 'SP07P32T3', 'SP07P33P0', 'SP07P33P2', 'SP07P33P1', 'SP07P33T0', 'SP07P33T1', 'SP07P33T2', 'SP07P33T3', 'SP07P34P0', 'SP07P34P1', 'SP07P34P2', 'SP07P34T0', 'SP07P34T1', 'SP07P34T2', 'SP07P34T3', 'SP07P35P0', 'SP07P35P1', 'SP07P35P2', 'SP07P35T0', 'SP07P35T1', 'SP07P35T2', 'SP07P35T3', 'SP07P36P0', 'SP07P36P1', 'SP07P36P2', 'SP07P36T0', 'SP07P36T1', 'SP07P36T2', 'SP07P36T3',
Start (8)	'ST01P0', 'ST01P1', 'ST01P2', 'ST01P3', 'ST13P0', 'ST13P1', 'ST13P2', 'ST13P3'
Switch (48)	'SW11P0A0', 'SW11P0B0', 'SW11P0A1', 'SW11P0B1', 'SW11P0A2', 'SW11P0B2', 'SW11P0A3', 'SW11P0B3', 'SW11P0T0', 'SW11P0T1', 'SW11P0T2', 'SW11P0T3', 'SW11P1A0', 'SW11P1B0', 'SW11P1A1', 'SW11P1B1', 'SW11P1A2', 'SW11P1B2', 'SW11P1A3', 'SW11P1B3', 'SW11P1T0', 'SW11P1T1', 'SW11P1T2', 'SW11P1T3', 'SW11P2A0', 'SW11P2B0', 'SW11P2A1', 'SW11P2B1', 'SW11P2A2', 'SW11P2B2', 'SW11P2A3', 'SW11P2B3', 'SW11P2T0', 'SW11P2T1', 'SW11P2T2', 'SW11P2T3', 'SW11P3A0', 'SW11P3B0', 'SW11P3A1', 'SW11P3B1', 'SW11P3A2', 'SW11P3B2', 'SW11P3A3', 'SW11P3B3', 'SW11P3T0', 'SW11P3T1', 'SW11P3T2', 'SW11P3T3'
Throw cards	'TC'

4.5. RL-AGENTS

From the literature, the decision was made regarding the techniques to apply. The study of related work in Section 2.9 led to the conclusion that Q-Learning and MCM based solutions with non-linear function approximation were the best options. These techniques were the most promising choices as they are the subject of many research projects and applied successfully to many gameplay agents. More than other techniques, like A3C and PPO. Both Q-Learning and MCM are model-free and fit environments with finite episodes. We decided in favor of Q-Learning to start researching DQN agents and investigate DMC based agents depending on the results with DQN and the available time (although Table 2.1 shows MCM and MCTS are more popular than DQN). The non-linear function approximation with NN has to address the imperfect information position of the agents.

The experiments with DQN and DMC are run on the same hardware with different software versions. The version difference in RLCard between DQN and DMC is significant. The framework is not backward compatible and requires changes in the model and environment classes between versions. There were many adjustments in the configuration of the DQN experiments in the search for success. These are described in Section 6.3. The configuration of the DMC experiments was varied to compare changes in NN sizes and state representations and are described in Chapter 7.

The DMC agents are trained by running the `run_dmc_keezen.py` script. This script instantiates and starts the `DMCTrainer`. This trainer class creates the models, buffers, and optimizers before starting the actor and learner processes. The agents are trained as 4 separate players that learn individual behavior without sharing anything.

4.6. RANDOM AGENT

The random agent or more precisely the randomly playing agent plays, as the name suggests, random moves. The agent receives a set of allowed moves in a state s and returns one of these moves randomly. The agent has no clue regarding positional values or strategy.

A reasonable player like the rule-based agent that is described in the next section defeats the random agent easily with a win-rate of 99% or more. Obviously the policy of the random agent results in a bad playing strength. This does not mean the randomly playing agent can never win. Randomly playing agents get lucky sometimes, getting good cards and taking the best moves accidentally. Hence, winning against any opponent, no matter how strong, is not impossible. Especially when playing thousands of games.

4.7. TRADITIONAL RULE-BASED AGENT

A traditional rule-based agent forms a benchmark for the RL agents by competing in autoplay of tournaments. We developed the rule-based agent for an iOs app and migrated the code to Python for this research. There were no other known agents available to compete. The policy π of the rule-based agent is to select the allowed action a that results in the state s with the highest value V_s.

The rule-based agent plays as strong as average, experienced human players, but weaker than top players. Feedback from human players that is received via reviews in the iOs App store shows that the playing strength of the rule-based agent is appreciated. There are people that find the agent too strong and others say the agent is too weak, but on average the players seem to appreciate the playing strength. Some users are annoyed by infrequent,

awkward moves. Although the awkward actions are rare, they may have a disproportionate influence on the appreciation of the game play.

The traditional rule-based agent is quite simple but plays reasonably well. The win rate against agents that play random moves is >99%. The win rate of the rule-based agent against humans is unknown. There are rare game positions where the calculated best action seems awkward, for example unnecessarily blocking or even hitting the teammate. These actions have only a small effect on the overall playing strength.

The rule-based agent evaluates board positions based on rules and values that are manually programmed. The calculations to value a game position are based on the programmer's biased view on the values of making progress, blocking opponents, hitting opponents, and helping the teammate.

The rule-based agent assigns values to marble moves. For example, a marble close to the finish has more value than far away. Blocking and hitting also have values. Based on those values, the move 4-steps-backward-from-start is played. The agent has no rule 'go 4 backward from start'. This also applies to the other observations shown in figure 5.1. The evaluation of marble moves leads to values that induce expected behavior. Hence, the agent needs no rules such as 'use the ace to start instead of running', 'finish smart', 'hit opponents if you can', 'don't hit the teammate' or 'swap pawns advantageously'.

The remainder of this section describes the implementation of the rule-based agent briefly. The implementation of the `RuleBasedAgent` class is located in `\rlcard\games\keezen\agent.py`.

The agent uses the following parameters while judging game positions:

- PROGRESS_OTHERS, PROGRESS_TEAMMATE, PROGRESS_SELF value the progress made by the other team, the teammate and the own position.
- HIT_OTHERS, HIT_TEAMMATE, HIT_SELF estimate the chances of hitting the marbles of the opponents, of the teammate and of the chance of being hit.
- BLOCK_OTHERS, BLOCK_TEAMMATE, BLOCK_SELF value the blocking of the other team, the teammate and the chances of being blocked.
- SWITCHING calculates advantages and disadvantages of swapping marbles and chances of own marbles being swapped.

The logic of the rule-based agent is shown in Algorithm 4. The value function of the agent loops over all marbles that take part in a move. Each marble position receives a value that is a combination of the listed parameters and if it is the current players marble, an opponents marble or a teammate marble. The parameters lead to values that depend on the marble positions, known card positions and possible card positions. Note that the terms *move* and *action* are interchangeable in this context.

The evaluation function uses the values of all fields on the board that are calculated per player according to Algorithm 5. The field values are progressively higher in the direction of the FINISH fields.

Algorithm 4 Calculation of the best move in a state by the rule-based agent.

Input:
o_s Observation of state s
Input:
M_s Set of legal moves in state s
Output:
m_s Best move in state s

Initialize:
$m_s \leftarrow None$
Initialize:
$v_m \leftarrow 0$
for each move in M_s **do**
    Execute move in s
    $tempv_m \leftarrow 0$
    for each $marble$ of $move$ **do**
        for each $parameter$ in (PROGRESS, HIT, BLOCK, SWITCHING) **do**
            $tempv_m \leftarrow tempv_m + Evaluate(o_s, marble, parameter)$
        end for
    end for
    if $tempv_m > v_m$ **then**
        $m_s \leftarrow move$
        $v_m \leftarrow tempv_m$
    end if
end for

Algorithm 5 Calculation of the field values from the perspective of a player.

Input:
F All fields ordered for the player, starting with WAIT fields and then clockwise from START to FINISH.
Output:
V Dictionary that maps all fields to values.

Initialize:
$value \leftarrow 1$
Initialize:
$delta \leftarrow 1$
Initialize:
$V \leftarrow \{\}$
for each f in F **do**
    if f is WAIT-of-player **then**
        $V[f] = 0$; **continue**
    end if
    if f is START-of-player **then**
        $V[f] = 50$; **continue**
    end if
    $V[f] = value$
    if f is START-of-other-player **then**
        $delta \leftarrow delta + 1$
        $value \leftarrow value + 15$
    end if
    if f is FINISH-of-player **then**
        $delta \leftarrow delta + 2$
        $value \leftarrow value + 50$
    end if
    $value = value + delta$
end for

The field values for all marbles directly lead to the values for the PROGRESS parameters during a move. The progress of the own marbles and the marbles of the teammate are positive and progress of the marbles of opponents is negative.

The values of the progress of marbles are adjusted by the opportunities and risks of blocking and hitting. The own cards, the played cards and the number of cards in the hands of the other players are input to chance estimations of possible actions of other players. The values for the HIT parameters depend on the field values and the chances that the opponents have a required card to hit the marble. The values for the BLOCK parameters depend on the length of the free path of the marbles and on the number of marbles the player has on the board. If the free path of a marble is 12 or less, then the value is $path\_length * 100/12$. If a player has a marble on the START field, a waiting marble, and a start card then a blocking value of -50 is calculated.

4.8. TRAINING PARAMETERS

Table 4.2 shows the parameters to train the DQN-agent. Experiences can be stored in a replay memory and sampled randomly to train the network. This is called experience replay.

Table 4.2: Parameters of the DQN agent.

Parameter	Description	Default	Value
replay_memory_size	Size of the replay memory.	20000	20000
replay_memory_init_size	Number of random experiences to sample when initializing the replay memory.	100	100
update_target_estimator_every	Copy parameters from the Q estimator to the target estimator every N steps.	1000	1000
discount_factor	Gamma discount factor.	0.99	0.99
epsilon_start	Chance to sample a random action.	1.0	1.0
epsilon_end	Final value value after epsilon is decayed.	0.1	0.1
epsilon_decay_steps	The steps to decay epsilon over.	20000	20000
batch_size	Size of batches to sample from the replay memory.	32	32
num_actions	The number of actions.	2	**271**
state_shape	The shape of the state space.	None	**[5x123]**
train_every	Train the network every N steps.	1	1
mlp_layers	The layers numbers and dimensions of the MLP.	None	**[4x512]**
learning_rate	Learning rate.	0.00005	0.00005
device	Whether to use the CPU or the GPU.	None	GPU

Table 4.3 shows the parameters to train the DMC-agent. The values that deviate from the default values are shown in **bold**. The value for the unroll_length was increased to 400 to prevent OutOfMemory errors. This parameter is related to the game length. The RMSProp parameters (alpha, momentum and epsilon) steer the Root Mean Square Propagation, which is a gradient descent optimization algorithm.

Table 4.3: Parameters of the DMC agent (`DMCTrainer`).

Parameter	Description	Default	Value
xpid	Experiment id.	dmc	**exp7**
save_interval	Interval in minutes to save the model.	30	**120**
num_actor_devices	The number of devices (GPU's).	1	1
num_actors	Number of actors for for each device.	5	5 or **6**
training_device	The index of the GPU used for training models.	0	0
total_frames	Total frames to train.	10^{11}	10^{11}
exp_epsilon	The probability for exploration.	0.01	0.01
batch_size	Learner batch size.	32	32
unroll_length	The unroll length (time dimension).	100	**400**
num_buffers	Number of shared-memory buffers.	50	50
num_threads	Number learner threads.	4	4
max_grad_norm	Max norm of gradients.	40	40
learning_rate	Learning rate.	0.0001	0.0001
alpha	RMSProp smoothing constant.	0.99	0.99
momentum	RMSProp momentum.	0	0
epsilon	RMSProp epsilon.	0.00001	0.00001

5

EXPERIMENTS

This chapter describes the experiments that produced the results in the next two chapters.

5.1. PLAN

The experiments are conducted to answer the research questions about the techniques and the performance of RL-agents at board-card games. The goals of the experiments are:

- Determine if board-card agents could be trained effectively using DQN or DMC.
- Measure the performance of the agents.
- Optimize the performance.

The experiments should indicate as early as possible if the technical setup actually works and if the training will deliver useful results. The importance of this step should not be underestimated. The first experiment is designed to fail fast to minimize the time to come to a working setup.

The measuring of the performance of the board-card agents, mainly the playing strength and the processing time, gives insight in the training process and is described in Section 5.3.

We tried to improve the performance of the DMC agent by conducting experiments with the NN size, the reward function, and the state representation. The DMC setup, like any RL environment, has many configurations or components to play with. The measuring methods provide a great benchmark for optimizing the performance.

The expectations of the experiments were that randomly playing agents would be defeated easily and that the rule-based agents could (be proven to) be defeated.

5.2. SETUP

The experiments were based on learning by self-play and the goal was to play full board-card games without simplifications. These two starting points lead to a configuration of four self-learning agents at all four board positions. These agents learn from their limited observations of the game state and the end rewards only.

The training takes place in an environment with four separate agents. The agents at opposite locations of the board form teams. All agents observe the game from their own

perspective. They do not observe any hidden information and can not communicate with their teammates. The only observables are defined by the rules of the game. Every agent has a separate NN that is fed with the game experiences of episodes with observables (o) and actions (a). During the training, each configured interval a total of 4 models (one for each agent) is saved in the designated experiment directory.

The models are loaded from disk back into memory for evaluation during tournament play. To evaluate the models of a stage of the training process, the models for two board locations (for example North at index 0 and South at index 2) and two other agents (random, rule-based or other trained agents) for the remaining board locations are initialized and play a tournament against each other.

5.3. MEASURING METHODS

The results are measured according to five methods. The first three methods are based on determining the win rate in tournament play, the fourth method focuses on in-game behavior, and the last method is all about effectiveness.

1. Play against randomly playing agents

   Trained agents play tournaments against agents that play random moves. The teams are formed by two identical agents, as stated in Section 4.1. Each tournament consists of 1000 matches. This number was chosen as a trade-off between resource consumption and noise reduction. The tournaments ran in parallel with the training processes. Every generated model competes in pairs in a tournament against a pair of randomly playing agents. The models from board positions North and South were taken to compete with agents at East and West.

   Defeating randomly playing agents should be an easy barrier to take. Random move selection makes the agent very weak (see Section 4.6). The win-rate against randomly playing agents is a good indicator especially during the first stages of the training process. The performance of untrained models is comparable to randomly playing agents en should increase quickly with training. One would expect a win-rate of around 99% like the rule-based agent (see Section 4.7) against randomly playing agents.

2. Play against other trained agents

   Tournaments between agents of different stages of the training process provide insight into the learning process. Expectations are that agents from early stages are weaker than agents from later stages in the learning process. Each tournament consists of 4000 matches.

3. Play against rule-based agents

   Rule-based agents play a decent game against humans and defeat randomly playing agents easily. Awareness of positional value is key to reaching the finish area faster than the opponents. Especially moving four steps backward from the start field is an effective accelerator that a strong agent is expected to apply often. Each tournament consists of 4000 matches. Models are chosen to compete in tournaments at reasonable intervals of 5×10^8 frames.

4. Manually analyze the play

   Manual analysis of the gameplay provides insights into the learning process and the playing strength and style of the agents. The analysis is executed by spectating the play of two teams of the same version of the agent. All moves of all agents are examined for interesting situations and overall style.

   The interesting situations are positions where real choices are made. Some choices generally have a worse outcome than alternatives. For example: running four steps backward from the start field is almost always clearly better than alternative moves. The choices are categorized into six aspects of play: *Starting*, *Running*, *Finishing*, *Hitting*, *Blocking*, and *Switching*. Some style-related observations, like 'plays the highest card first', are taken into account as well. The reason is that it might be disadvantageous if the play gets predictive. Figure 5.1 shows examples of observations.

   The observations of the manual analysis and the rule-based agent cannot be seen as fully independent from each other. I observed, interpreted, and chose the situations, while I also made the rule-based agent. As a result, my view might be biased and there is a risk that the manual analysis judges the RL agents according to the rule-based implementation. A consequence is that it might be difficult to spot better play than the behavior of the rule-based agent. It is precisely in complex situations for the rule-based agent (and me) that RL agents could excel. It is good to be aware of this while trying to observe games without prejudices to form a useful addition to auto-play tournaments.

   Manual analysis of observations and selected actions of trained agents at different stages of the training process provides insights in the learning process. The increased playing strength is expected to go hand in hand with better judgment in the situations shown in Figure 5.1.

5. Measure the processing time

   The time a training takes is an important part of the performance. One wants good results with minimal processing time and low resource consumption. The processing speed is measured by the processing time per frame. The win rate progress in time to determines the most efficient solution.

The playing strength is measured by the mutual performance in tournament play and expressed in a win rate percentage. The use of existing indicators of the playing strength, such as ELO [1] and Microsoft TrueSkill [2], was considered. Both indicators measure the relative skill levels. Although these rating systems have their value, the extra value in this project is minimal because of the lack of references or benchmarks.

The tournament play has advantages and disadvantages. An advantage of automatic tournament play is that it is a clean quantitative measurement. The speed of automatic play allows a large number of matches to be played, reducing the noise generated by randomness and incidental errors. A disadvantage is that it is just a relative measurement. It measures the playing strength between the tournament participants without absolute values. During tournament play, biased agents or fouls may stay unnoticed.

[1] https://en.wikipedia.org/wiki/Elo_rating_system
[2] https://www.microsoft.com/en-us/research/project/trueskill-ranking-system/

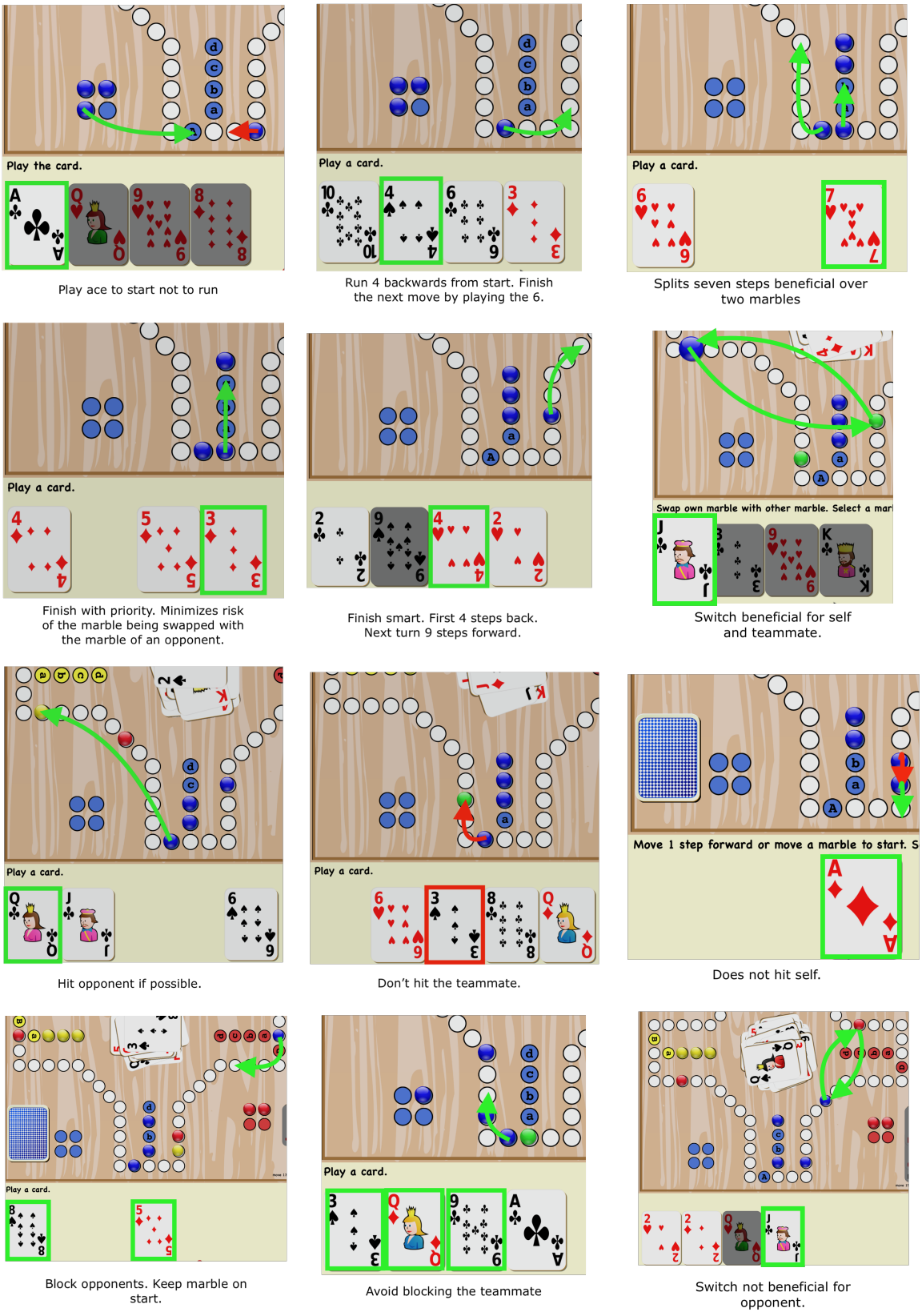

Figure 5.1: Examples of observations during manual analysis. Green cards and arrows indicate good choices and red indicates bad choices.

Manual analysis is a sensible addition to tournament play to identify biases and fouls. This analysis is a qualitative and resource-intensive method. Although a small number of games are analyzed, these games consist of a decent amount of moves that provide trustworthy insights into the playing strength and style of the agents. There is a lot of nuance in the manual analysis. Players do not always have one right move. Moves often have nuances because multiple considerations regarding progressing, blocking, hitting, and finishing are made in parallel. In some positions running four steps backward might not be chosen to block an opponent. In general, it is better to play an Ace to start than to run 1 field, but if a marble of the opponent is hit or during a finishing maneuver, then running with an Ace might be smarter. The interesting events are moves with a choice that has clear wins or losses. These events are counted as plus or minus in case of a good or bad choice. The moves with no clear outcome were ignored.

5.4. Running Experiments

The experiments with DQN and DMC run in separate configurations but are very similar. The DQN training is started with the script `keezen_dqn.py` and the DMC training is started with the `run_dmc_keezen.py`. Every experiment has a unique identifier and each training run outputs a log file and the results in a dedicated folder. Agents are instantiated for each of the four player positions. Then the trajectories are generated by running games in the environment and feeding them to the agents. The progress of the training was measured by running tournaments of the trained agents against randomly playing agents.

The evaluation tournaments against random agents differs somewhat between DQN and DMC. The DQN experiments run the tournaments as part of the training script every 10k trained episodes. The DQN experiments save the performances in the tournaments at every stage of the training, and a graphical representation (a .png file). The DMC evaluations are run by a separate cron job that administrates all processed tournament results in a text file (`processed.csv`). Every run, the job checks for not processed models and starts a tournament with the latest one.

5.4.1. Tournaments against randomly playing agents

The tournaments against randomly playing agents are run by a cron job that runs the `evaluate_keezen_rnd.py` script. The script reads the already processed models from the file `processed.csv`. It reads all generated models from the same path, picks the latest, and runs a tournament of 1000 matches against randomly playing agents. The script `create_graph_rnd.py` reads all results from the the file `processed.csv` and generates the Figure 7.1.

5.4.2. Tournaments against other trained agents

The tournaments against other versions of agents are started manually. The script evaluate_keezen_versions.py loads the agents of two teams and runs a tournament of 4000 matches. The agents load the models by specifying the number of trained frames. The resulting values are collected in Table 7.1.

5.4.3. TOURNAMENTS AGAINST RULE-BASED AGENTS

The tournaments against rule-based agents are run by the script `evaluate_keezen_rb.py`. Every $5x10^8$ frames two DMC-agents team up and play 4000 matches against a team of rule-based agents. Figure 7.2 is created with the script `create_graph_values.py`.

5.4.4. MANUAL ANALYSIS

The manual analysis is run by the script `evaluate_keezen_man.py`. The script loads the four DMC-models and outputs the game position, including the cards of all players, to the screen. The player that has the turn is indicated by the "<->" next to the cards. By pressing Enter, the next move is executed by a DMC-agent. Figure 5.2 shows the output during the manual analysis.

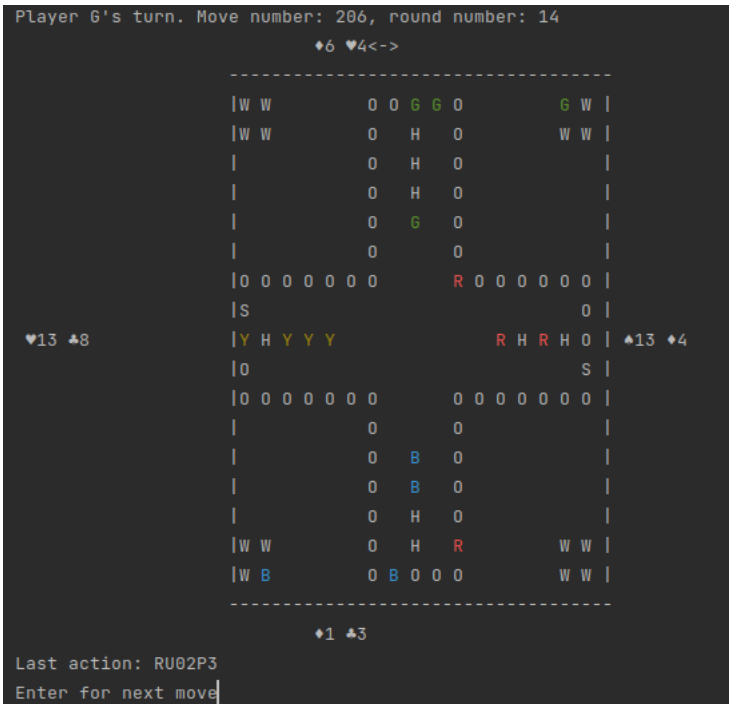

Figure 5.2: Output during manual analysis.

5.4.5. PROCESSING TIME

The processing time is measured by collecting the time of the creation of the generated models. The processing speed is the number of frames per unit of time. The script create_graph_rnd_exp7_exp20_speed.py finds the models and plots the modification dates versus the number of frames.

During training, the goal is to achieve a high win rate in the least possible time. The processing speed (processed frames per unit of time) tells not the entire story, as the win rate per processed frame is not constant. The win rate per unit of time plots tells a completer story. For example, the higher processing speed due to a smaller NN should not decrease the win rate per precessed frame (too much) to be beneficial.

5.5. SYSTEM

Table 5.1 specifies the hardware and operating system used during the research.

Table 5.1: Hardware and operation system

Processor	AMD Ryzen 7 3700x 8-core
Memory	32GB
GPU	NVIDIA GeForce RTX 2070 Super
OS	Ubuntu 20.04.3 LTS

6

DQN

This chapter discusses the attempts that were taken to train board-card agents with DQN. We briefly describe the experiments, the results, and a discussion. Due to the disappointing results, the discussion of DQN is limited compared to the description of the more successful DMC in the next chapter.

6.1. Experiments

The experiments with DQN consist basically of training DQN agents, evaluating the agents in tournaments against randomly playing agents, doing manual analysis of the playing style to track causes of the weak performance and making changes to the experiment.

A lot of effort has gone into experiments to improve the playing strength of the DQN implementation without success:

- We experimented with parameters of the algorithm, such as learning rate, discount factor, and epsilon (exploration versus exploitation).
- We experimented with technical parameters, such as batch size, update period, buffer size, replay size, and network size.
- We experimented with changes in the model:
  - We used marble identifiers and we used marble indexes based on board positions.
  - We tried different perspectives: birds-eye perspective and player perspective.
  - We tried models without the cards, just the board.
  - We changed the input shape from numbers to only zeros and ones.
  - We experimented with perfect information in the game.
- We tried with longer training. Several training sessions lasted for two weeks without any improvements after two or three days. There was no indication that continuing would make sense.
- We experimented with simplifications of the game. We tried:

  - games with fewer marbles: two or three marbles per player instead of four.
  - games with fewer cards/actions: fewer cards and no 7-card to decrease the number of actions to 25 instead of 271.
  - games without team play (stop when one player's marbles are finished).
  - a combination of the above.
  - an implementation of Ludo (Mens-erger-je-niet) (25 actions).
- We have validated the experiment setup:
  - We tested extensively.
  - We debugged extensively.
  - We have done code reviews.
- We did more literature study (action shaping, DQN limitations such as overestimation).
- We have analyzed the play manually.
- We have tried to introduce specific rewards:
  - We have experimented with the values of rewards: higher values for winning and lower values for losing.
  - We have experimented with extra rewards: reward for moving four steps backward from the start field
  - We have experimented with higher rewards for shorter games.
- We contacted an expert from the RLCard team. He confirmed that DQN could be problematic with large action spaces.
- We tried alternatives. NFSP did not improve. We failed to get A3C and PPO working with Keezen. DMC performed much better.

6.2. Results

Training DQN agents at the Keezen game gives the results as shown in Figures 6.1, 6.2 and 6.3. These figures are three samples to show that the trained DQN agents have a maximum win rate of only 68% against randomly playing agents. This score is low, considering random play already scores a win rate of 50%. These are the results of many attempts that did not get better than this. The next section discusses a possible explanation. In DQN, the timestep represents the number of times a batch of training data is processed.

The results of the DQN implementation against randomly playing agents are the reason to skip the tournaments during training and against rule-based agents. Not taking the first barrier makes those two tournament types of little value. Expect a win rate of about 3% against rule-based agents, based on the results of a short-trained DMC agent of comparable strength.

Manual analysis confirms that DQN agents perform poorly. The agents did not learn the move to go four steps backward from the start field, effectively skipping a complete run

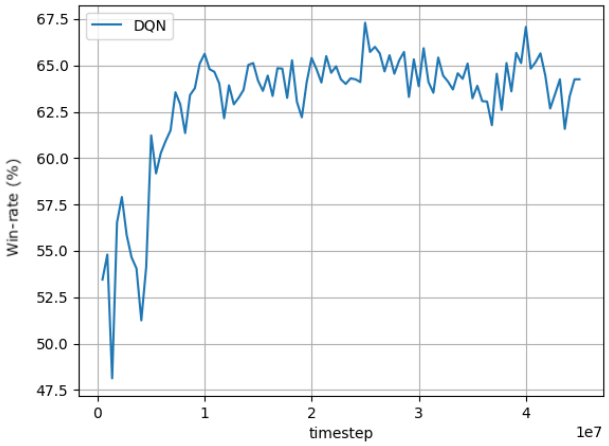

Figure 6.1: Tournament results sample of DQN agents against randomly playing agents.

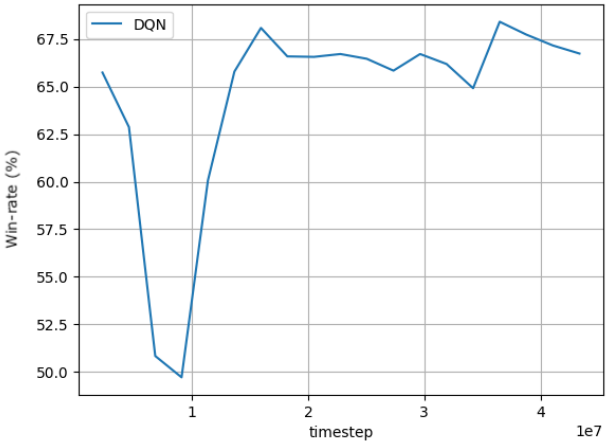

Figure 6.2: Tournament results sample of DQN agents against randomly playing agents.

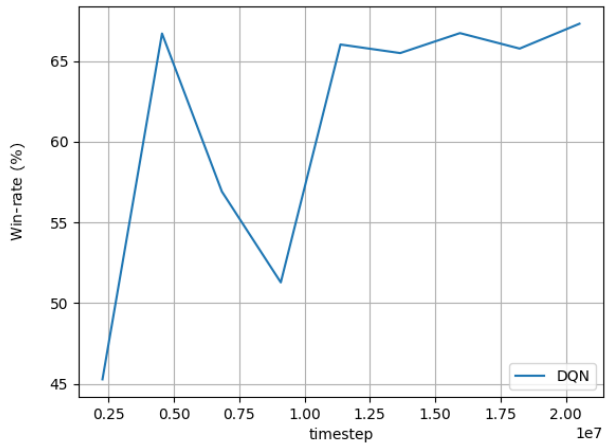

Figure 6.3: Tournament results sample of DQN agents against randomly playing agents.

of a marble on the board. This opportunity manifests itself at least once in most episodes. During manual analysis of the moves, it is hard to point out what the DQN agents have learned better than randomly playing agents.

6.3. Discussion

The results, as presented in the previous section, show that the DQN performance on board-card games is much worse than anticipated. The expectations based on results with a simple rule-based agent were that the DQN implementation would defeat the random agents with a win rate of at least 95%. After many experiments, the maximum tournament result is a 68% win rate against randomly playing agents.

The results confirm that the playing strength of the DQN agents is very weak. Automatic tournament play and manual analysis show that the DQN agents do not learn the game well. An obvious move like four steps backward from the start field, effectively skipping a full run of the marble around the board, is not performed. Even interim rewards for this move did not help.

More experiments and changes did not significantly improve the performance. In general the later results as seen in Figures 6.2 and 6.3 were only slightly better than the early results in Figure 6.1.

The characteristics of board-card games result in variances in the DQN training. The characteristics were discussed in Section 3.4 and show that board-card games have many action definitions (Keezen: 271). Only a small subset of these actions in a state is legal, depending on the cards in the hand of the player and the position of the marbles on the board. Often there are no legal moves at all. Variances are also introduced by the multi-agent and imperfect information characteristics.

The variances that are inherent to the characteristics of board-card games are likely to cause difficulties in training board-card games with DQN. Chapter 2 discussed that TD is incremental and harder to stabilize than MCM and that DQN has decreased convergence and stability compared to TD.

According to [Heinrich and Silver, 2016], the DQN algorithm learns a deterministic and greedy strategy that is sufficient for single agent MDPs, but not for large multi-agent imperfect information games. This type of games require more stochastic policies. If agents are ϵ-greedy, with smaller chances on exploring, the generated experiences are highly correlated with a narrow distribution of states. The authors introduce NFSP to achieve better results at imperfect information games and show improvement over DQN at playing poker (Limit Texas Hold'em). The policy of NFSP agents to generate experiences changes more slowly. This results in smoother variances in the experiences and more stable NNs. Experiments with NFSP on board-card games did not indicate any better performance than DQN. The characteristics of board-card games are worse for DQN, such as a larger action space and much longer games.

Literature describes similar problems of the card game Dou Dizhu with DQN [You et al., 2019]. The authors propose combinational Q-learning to handle the combinational action space. In DouZero this problem is tackled by the implementation of DMC [Zha et al., 2021]. DMC is more efficient than DQN in handling large action spaces.

The failure to achieve decent results with DQN does not mean board-card games can not be trained with DQN. This research shows it is harder to get good results with DQN than DMC, but the belief remains that it is possible to achieve better results with DQN.

7

DMC

This chapter discusses the experiments and the results with DMC.

7.1. Experiments

All experiments that were described in Chapter 5 were conducted with DMC:

- The playing strength was measured against randomly playing agents, against other versions of the agents, and against rule-based agents.
- Games played by DMC agents from several stages of the training process were manually analyzed.
- The processing time (and cost) of the training was determined.
- Experiments with several possible training optimizations were conducted.

7.2. Results

7.2.1. Tournaments against randomly playing agents

Figure 7.1 shows the tournament results of two trained DMC agents against two agents that play random moves. The figure shows the number of frames or game positions and the win percentage against random agents. The models at board positions North and South played a tournament of 1000 games against two randomly playing agents at board positions East and West. The agents achieve a win rate in tournaments of over 99%.

The NN size is $[5 \times 512]$, the model has no card count information, and the model is created from the *generic perspective* (see Section 4.4.1). The training took 49 days (cycle time).

7.2.2. Tournaments against other trained agents

Table 7.1 shows the tournament results of several versions of DMC agents during the training. Each row in the table shows the tournament results of that agent against the other agents in the column headers.

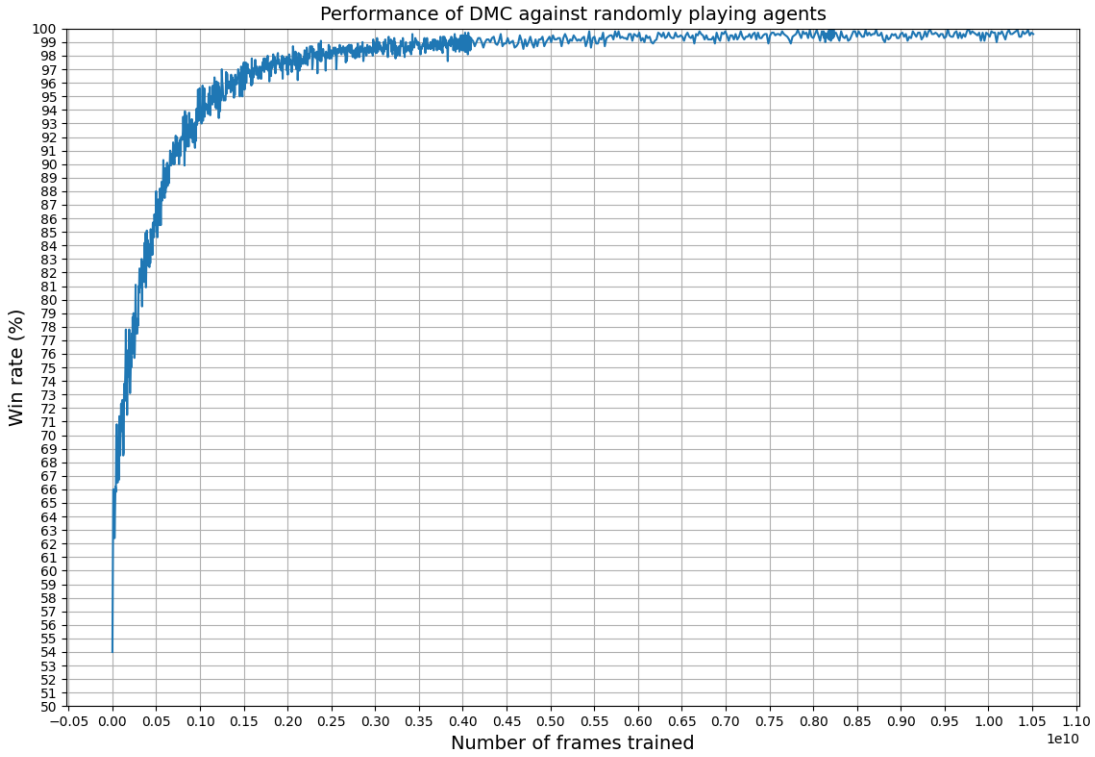

Figure 7.1: Tournament results of DMC agents against randomly playing agents. The thinner line after 4×10^9 frames is due to a change in the save interval of the models from 30 minutes to 2 hours.

Table 7.1: Win-rate of DMC agents during training.

Agent (frames)	0 (random)	$1x10^9$	$2x10^9$	$3x10^9$	$4x10^9$	$7x10^9$	$10x10^9$
0 (random)	X	0.058	0.034	0.019	0.008	0.005	0.004
$1x10^9$	0.942	X	0.313	0.238	0.194	0.138	0.109
$2x10^9$	0.966	0.687	X	0.416	0.348	0.274	0.226
$3x10^9$	0.981	0.762	0.584	X	0.436	0.342	0.295
$4x10^9$	0.992	0.806	0.652	0.564	X	0.411	0.355
$7x10^9$	0.995	0.862	0.726	0.658	0.589	X	0.426
$10x10^9$	0.996	0.891	0.774	0.705	0.645	0.574	X

7.2.3. TOURNAMENTS AGAINST RULE-BASED AGENTS

Figure 7.2 shows the results of tournaments of 4000 games of DMC agents against rule-based agents.

The increasing win-rate values from left to right indicate that the DMC agents get stronger with more training. The DMC agents come on par with the rule-based agents and the curve suggests that the trained agents will surpass the rule-based agents.

The agents are the same as the agents that played against the random agents. The NN size is $[5 \times 512]$, the model has no card count information, and the model is created from the *generic perspective* (see Section 4.4.1).

7.2.4. MANUAL ANALYSIS

The agents of three separate training stages are evaluated manually according to the described method. Table 7.2 shows the analysis of the DMC agents after training $1.5x10^9$, $4x10^9$, and $10x10^9$ frames. The results are based on two games (428 moves), three games (721 moves), and three games (793 moves), respectively. The values are calculated by dividing the number of positive events by the total of events. The values are rounded to 1 decimal place. Hence, the values are between zero and one, and higher values are better. If two or fewer events occurred in total, this resulted in a N/A (not applicable).

The values of the manual analysis in Table 7.2 lead to the learning status of the three DMC-agent versions. Table 7.3 shows the analysis of DMC after training $1.5x10^9$ frames, Table 7.4 shows the analysis of DMC after training $4x10^9$ frames, and Table 7.5 shows the analysis of DMC after training $10x10^9$ frames. After more training, the DMC agent learned to start, run, finish, hit, block and switch much better. The latest agent seems to have learned to anticipate on hitting opponents and avoiding to get hit.

The second agent performed an interesting move. Instead of directly running four steps backward from the start field, the agent first moved two steps forward and ran four steps backward the next turn. This maneuver seems smarter because of decreased chances of opponents swapping the marble to prevent a fast finish. It is not clear if this was the result of the learning process or just a coincidence. The third agent used the Ace to run in the finish area more than the second agent, resulting in a lower score. The latest agent's primary focus seems hitting the opponents. The agent has learned better not to hit the own marbles and helps the teammate when swapping marbles.

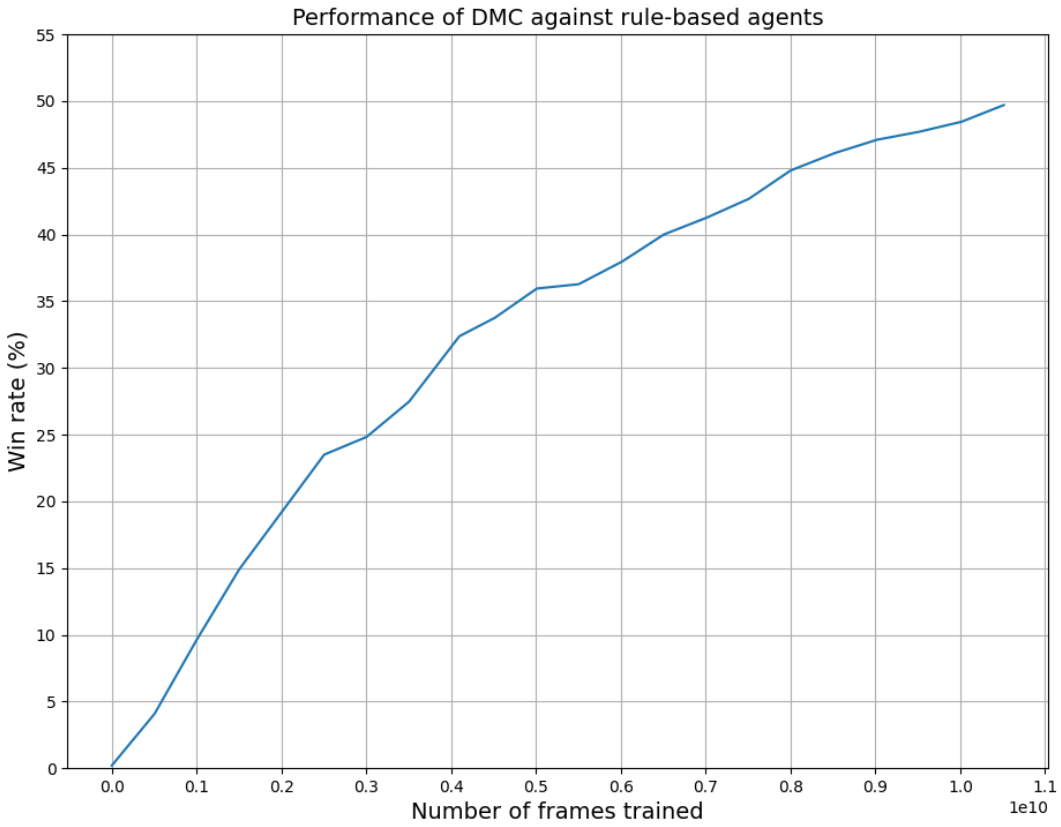

Figure 7.2: Tournament results of DMC agents against rule-based agents. After 10.5×10^9 the DMC agents play as strong as the rule-based agents.

Table 7.2: Manual analysis of DMC-agents after training $1.5x10^9$, $4x10^9$, and $10x10^9$ frames.

	$1.5x10^9$	$4x10^9$	$10x10^9$
Starting			
Use ace as starter, not run	0.4	0.9	0.8
Moves marble from start if agent has other starter card	0	0.4	0.5
Running			
Runs 4 backward from start field first marble	0.6	1	0.9
Runs 4 backward from start field other marbles	0	0.8	0.8
Runs not just with front marble	0.9	1	1
Plays not always the highest card first	1	1	1
Splits seven steps beneficial over two marbles	0.9	0.9	0.9
Finishing			
Finishes with priority	0.8	1	1
Finishes smart	0.6	0.9	0.9
Hitting			
Hits opponents if possible	0.9	0.9	1
Does not hit the teammate	0.3	0.7	0.7
Does not hit self	0.1	0.3	0.8
Runs with chance on hitting opponents	N/A	N/A	1
Decreases chances of being hit	0.4	0.6	0.6
Blocking			
Keeps marble on own start field to block opponents	0.7	1	0.9
Avoids blocking the teammate	0.2	1	1
Tries to avoid being blocked	N/A	N/A	0.8
Switching			
Switches beneficial for self	0.6	0.9	0.9
Switches beneficial for teammate	0	0.4	0.8
Switches not beneficial for opponents	0.4	1	0.9

Table 7.3: Learning analysis of DMC after training $1.5x10^9$ frames.

Aspect	**Learning status at $1.5x10^9$ frames**	**Still to learn**
Starting	Prefers running one step with a marble over putting a new marble on the start field.	More starting with ace.
Starting	Does not free the start field to start with new marbles if applicable.	Free the start field for waiting marbles.
Running	Plays 4 backward with the first marble, not with the other marbles.	Run 4 backward with all marbles.
Running	Runs marbles and plays cards without being predictive.	Ok
Finishing	Finishes with priority and quite smart.	Get smarter at finishing.
Hitting	Hits opponents almost perfectly, but hits the teammate and own marbles too.	Don't hit teammates and own marbles.
Blocking	Blocks opponents and blocks the teammate too.	Don't block the teammate.
Switching	Switches the own marbles ok. Switching marbles of teammate is bad and the opponents a little better.	Switch more effective, especially teammate and opponents.

Table 7.4: Learning analysis of DMC after training $4x10^9$ frames.

Aspect	**Learning status at $4x10^9$ frames**	**Still to learn**
Starting	Prefers putting a new marble on the start field over running one step with a marble.	More starting with ace.
Starting	Frees the start field in some occasions to start with new marbles.	Free the start field more often for waiting marbles.
Running	Always plays 4 backward with the first marble, in most occasions with next marbles.	Run 4 backward with all marbles.
Running	Runs marbles and plays cards without being predictive.	Ok
Finishing	Finishes with high priority and smart.	Get smarter at finishing.
Hitting	Hits opponents almost perfectly, but hits teammates and own marbles unnecessary on some occasions.	Don't hit the teammate or own marbles.
Hitting	Seems to have some notion of chances of being hit, but seems not to avoid other start fields.	Learn to prevent being hit.
Blocking	Blocks opponents and avoids blocking the teammate.	Ok
Switching	Switches own marbles, and those of the opponents effectively. On some occasions switching the teammates marbles could be better.	Switch more effective, especially the teammate.

Table 7.5: Learning analysis of DMC after training $10x10^9$ frames.

Aspect	**Learning status at $10x10^9$ frames**	**Still to learn**
Starting	Prefers putting a new marble on the start field over running one step with a marble.	Ok, prefers finishing occasionally.
Starting	Frees the start field in some occasions to start with new marbles.	Free the start field more often for waiting marbles.
Running	Always plays 4 backward with the first marble, in most occasions with next marbles.	Run 4 backward with all marbles.
Running	Runs marbles and plays cards without being predictive.	Ok
Finishing	Finishes with high priority and smart.	Get smarter at finishing.
Hitting	Hits opponents with high priority, still hits the teammates marble on some occasions and hits fewer own marbles.	Don't hit the teammate or own marbles.
Hitting	Seems to have some notion of chances of being hit, but seems not to avoid other start fields.	Learn to prevent being hit.
Blocking	Blocks opponents, avoids blocking the teammate and tries to avoid being blocked.	Ok
Switching	Switches own marbles, and those of the opponents and teammates effectively.	Learn even better switching the teammate.

7.2.5. PROCESSING TIME AND COST

Table 7.6 gives an indication of the training time given the setup described in Sector 5.4. Two periods of the training took much longer. A job was running in parallel to gather the tournament results between $3x10^9$ frames and $4x10^9$ frames, and there was a problem with a resource-consuming job after $8x10^9$ frames. Without those two delays, the training processed roughly $9.7x10^6$ frames per hour.

The training time results in costs. There are many dependencies and assumptions in a cost calculation. For example, dealing with depreciation, produced (useful) heat and fluctuating energy prices. We keep it simple and calculate the direct costs of the training without taking depreciation and other factors into account. The power consumption of the training system with 100% GPU load is 200 Watt. A kWh costs Euro 0.24. This electricity price is low given the current market prices, but the system also works on solar panels during the day. These considerations lead to the following calculation:
$Costs$ = Consumption [kW] × Time [h] × Price [EUR/kWh] = 0.2 × 1185 × 0.24 = 56.88 EUR

Table 7.6: Training time of DMC.

Frames trained	Delta (hours)	Total (hours)	Total (days)
$1x10^9$	103	103	4.3
$2x10^9$	103	206	8.6
$3x10^9$	102	308	12,8
$4x10^9$	192	500	20.8
$5x10^9$	100	600	25
$6x10^9$	102	702	29.3
$7x10^9$	102	804	33.5
$8x10^9$	100	904	37.7
$9x10^9$	179	1083	45.1
$10x10^9$	102	1185	49.4

7.2.6. OPTIMIZATION EXPERIMENTS

We experimented with the NN size, the reward function, and the observation representation to try to optimize the training process and improve the evaluation results. Table 7.7 shows the experiments with the variables.

Table 7.7: Experiments with DMC to optimize the results: id, description and variables. The bold values (that are not in the first row or first column) indicate a change compared with the experiment above. The color column indicates the line color in the graphs.

Id	Description	NN Size	Reward	Card count	Rows	Perspective	Color
A	Base experiment	[5 × 512]	[0, 1]	No	5	Generic	Blue
B	NN size	**[2 × 512]**	[0, 1]	No	5	Generic	Red
C	Rewards	[2 × 512]	**[-26, 26]**	No	5	Generic	Purple
D	Card count	[2 × 512]	[-26, 26]	**Yes**	5	Generic	Black
E	No redundancy	[2 × 512]	[-26, 26]	Yes	**4**	Generic	Green
F	Perspective	[2 × 512]	[-26, 26]	Yes	4	**Player**	Orange

Experiment **A** forms the basis that we try to optimize with the other experiments. It is

the DMC experiment that is already described with the results in this chapter. The experiments **B** up to and including **F** build on experiment **A** by cascading the extra variables.

Figures 7.3 and 7.4 show the results of the experiments that are listed in table 7.7. The first figure shows the win rate against randomly playing agents during the training of the experiments. The second figure shows the same data as the first figure with a higher zoom level.

Figure 7.5 shows the processing speed of the experiments. Note that three groups of experiments are formed: Experiment 1) **A**, 2) Experiments **B** and **C**, and 3) Experiments **D**,**E** and **F**.

Figure 7.6 compares the win rate during a training of experiment **F** with base experiment **A**. The training of **F** shows a higher win rate per frame. The training of experiment **F** also had a higher speed than experiment **A**. Due to different training circumstances it is hard to measure the speed difference exactly. An estimation is a speed gain of a little less than 15% based on Figure 7.5.

NETWORK SIZE

The NN of the DMC agents in experiment **A** has a size of $[5 \times 512]$. A smaller network is expected to speed up the training process at the cost of some possible information loss. Experiment **B** utilizes a NN size of $[2 \times 512]$ to test how much faster the training process would be and if the playing strength would suffer. One extra (6th) actor process was configured to achieve a 100% GPU utilization again.

Figure 7.7 shows the results of experiments **A** and **B** with the two different network sizes after training more than ten days. The win rate against randomly playing agents during both learning processes largely overlap. Figure 7.5 shows that the processing time per frame between experiments **A** and **B** decreased due to the smaller NN. Processing of an equal number of frames of 2.75×10^9 took 240 hours in stead of 283 hours, meaning 15% faster processing per frame.

REWARD FUNCTION

Experiments **A** and **B** have a simple reward function: 1 for the winners and 0 for the losers. A more aggressive reward function is expected to show some difference in the scoring results and probably shorter games. Experiments **C** and further utilize a reward function that rewards wins by a big lead higher than wins by a small small lead. The reward function is inspired by the scoring system at the national Keezen championship in the Netherlands [1] and is calculated as follows:

- Each finished marble is 2 points.
- The winners get 10 points bonus.
- The losers get the negative value of the reward of the winners.

The winning team will always have 8 finished marbles. This is $8 \times 2 + 10 = 26$ points. Each of the finished marbles of the losing team counts as 2 points and is subtracted from 26. So if the losing team has 5 finished marbles then the winning team gets a reward of $26 - 5 \times 2 = 16$. The losing team gets a reward of -16.

[1] https://nk.keezbord.nl/officiele-spelregels/

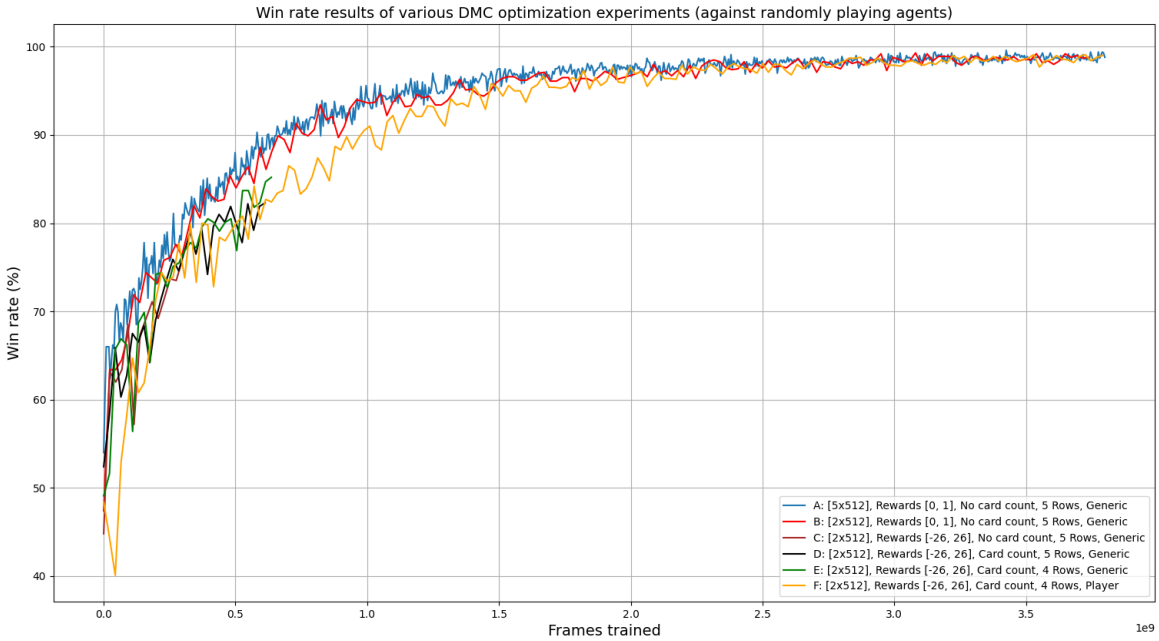

Figure 7.3: Tournament results of DMC agents of optimization experiments **A** to **F** against randomly playing agents.

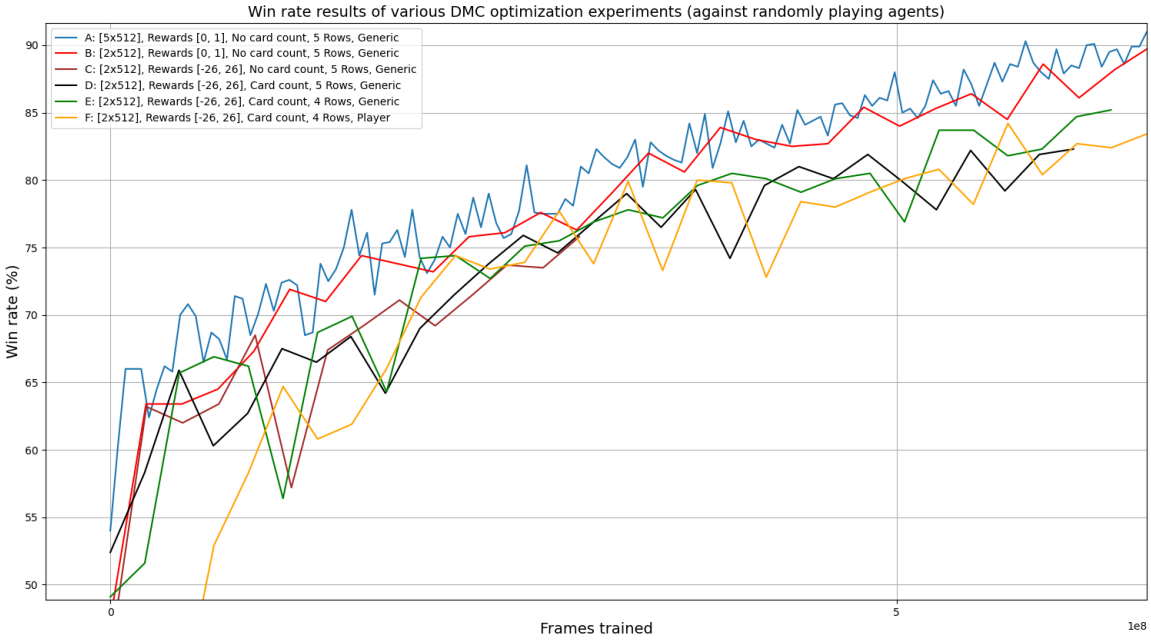

Figure 7.4: Zoomed in at figure 7.3.

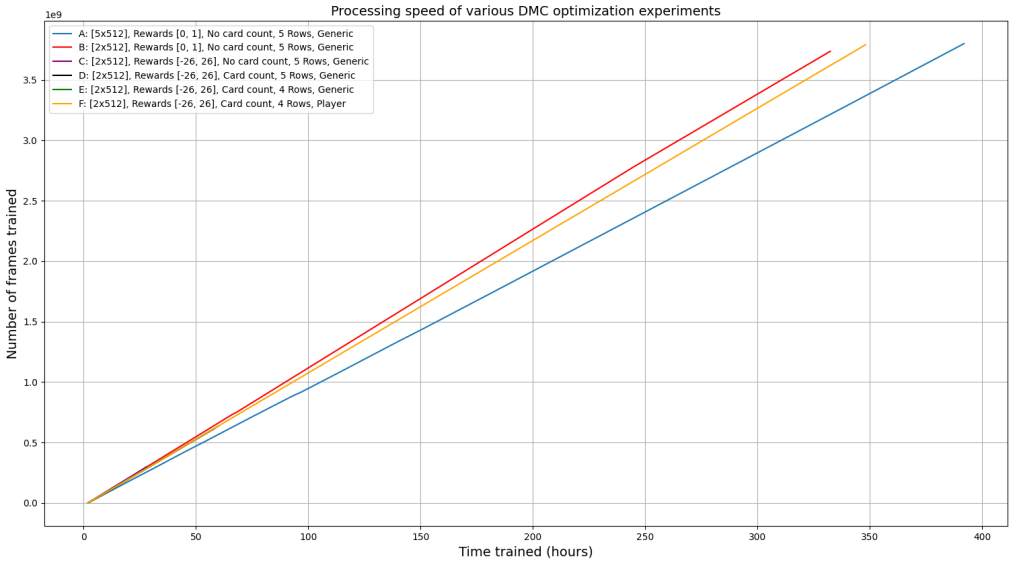

Figure 7.5: Processing speed of training DMC agents of optimization experiments **A** to **F**. Experiments **B** and **C** overlap and experiments **D**,**E** and **F** overlap.

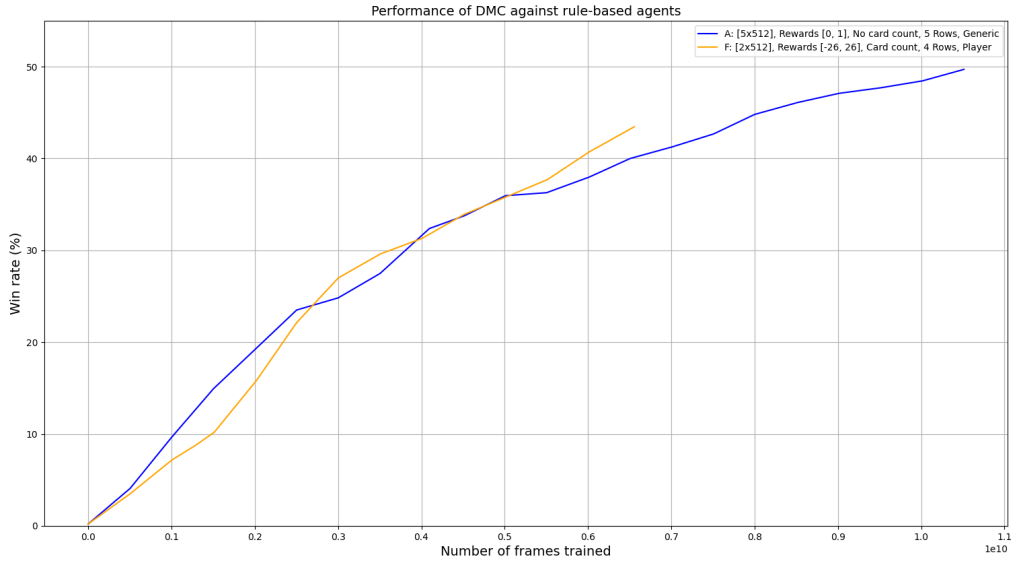

Figure 7.6: Performance of training DMC agents of experiments **A** to **F** against rule-based agents.

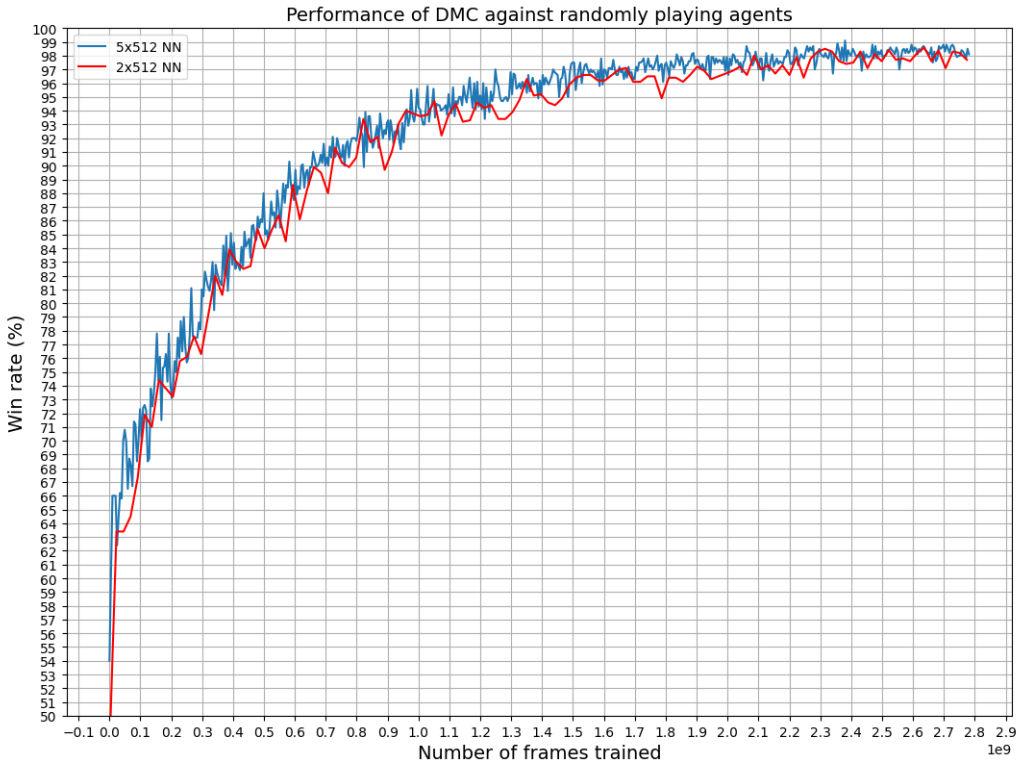

Figure 7.7: Tournament results of DMC agents with different network sizes against randomly playing agents. Experiment **A**: $[5 \times 512]$ and **B**: $[2 \times 512]$

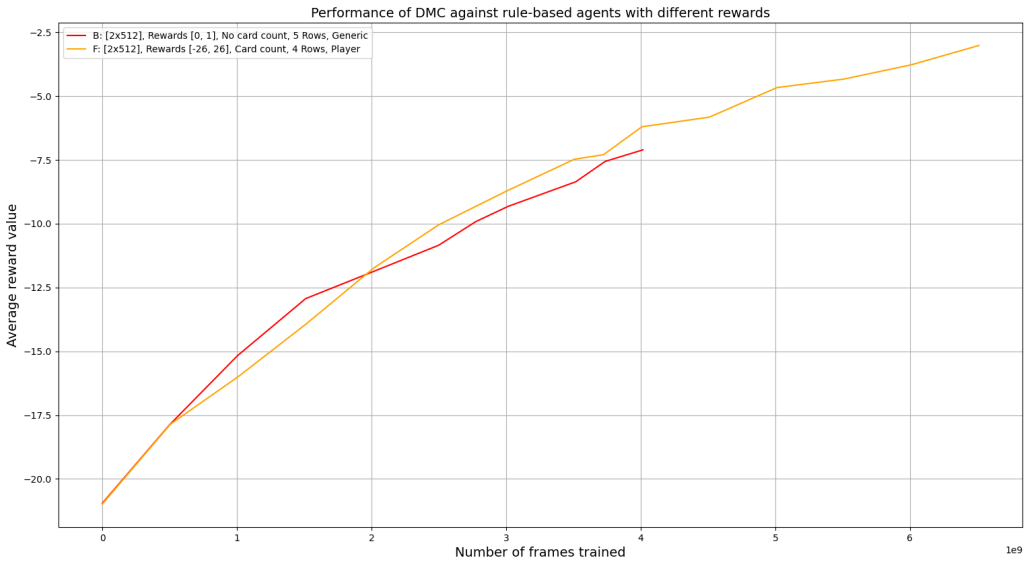

Figure 7.8: Average rewards for DMC agents, that are trained with different reward functions, against rule-based agents. The DMC agents were trained with two different reward functions: $[0,1]$ and $[-26,26]$.

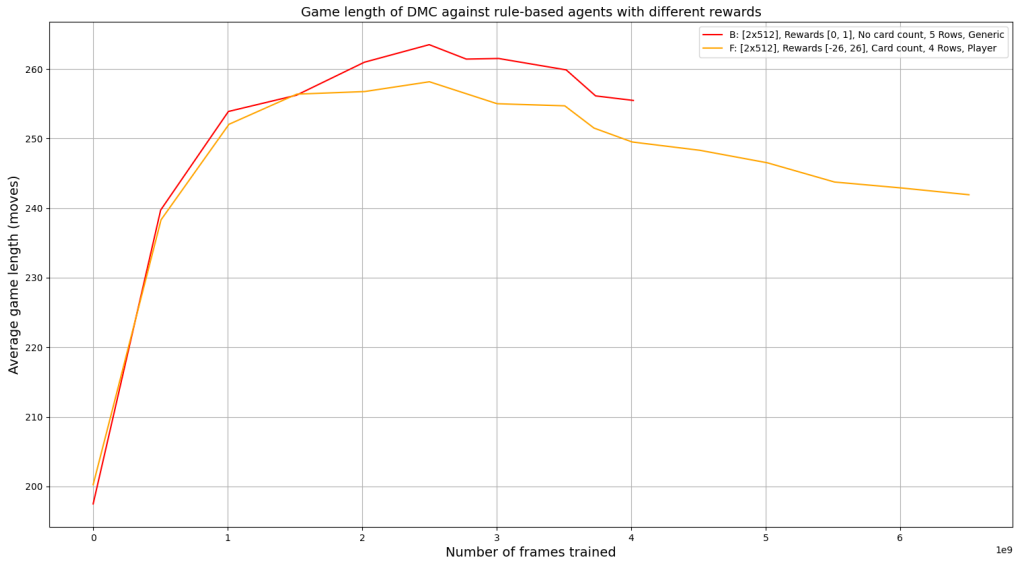

Figure 7.9: Average number of moves of DMC agents, that are trained with different reward functions, against rule-based agents. The DMC agents were trained with two different reward functions: $[0,1]$ and $[-26,26]$.

Figures 7.8 and 7.9 show the results of experiments **B** and **F**. Although there is no direct comparison of the win rate and game length between experiments **B** and **C**, with the results against the randomly playing agents (Figure 7.4), there is a trend to recognize. All experiments with the more aggressive $[-26,26]$ reward function start with worse win rates than the experiments with the $[0,1]$ reward function and seem to recover after 2×10^9 frames and achieve higher win rates after that. The agents need a little more time to understand the more aggressive reward function and benefit from this later. The average length of games shows a comparable pattern.

The average length of games depends on the playing strength of both teams. A team of not trained DMC-agents is beaten fast by a team of the rule-based agents. During training the DMC-agents learn to play better. Better game-play consist of two game facets: finish faster (shorter games) and thwart the opponents better (longer games). The manual analysis can give an explanation. Table 7.2 shows that hitting the marbles of opponents is learned early in the training process (at $1.5 \times 10^9 frames$). On average, this makes games longer. The manual analysis also shows that the moves to finish quickly are learned later. Example moves are 'running 4 steps backward from the start field', finishing (smart and with priority), and switching beneficially. This explains the curve starting with a game length of 200 moves, with a top around 260 moves and then gradually declining. We can also speculate that the more aggressive reward function stimulates to finish faster more than to thwart the opponents, which results in shorter games and the orange curve in Figure 7.9.

Observation model

The observation model was changed by three variables (see Table 7.7): the card count, a 5th row with redundant information, and the perspective.

Card count
The number of cards in the hands of other players is observational information that helps judging some positions. A player without cards to play will not interfere with the current board position. During one or a couple of turns, the other players with cards have no risk of being hit or blocked by this player and have chances to hit or block the player without intervening. Adding the card count to the model is expected to improve the playing strength of the agents. The models of experiments **A**, **B**, and **C** did not include the number of cards in the hands of all players. In experiments **D**, **E**, and **F** this information is included.

No redundancy
The observation models of experiments **A**, **B**, **C**, and **D** contain a fifth row with redundant information because we thought this could be beneficial for the NN. This fifth row contains a 1 at each index of a column that contains a 1 in any of the other 4 rows, and a 0 otherwise. Since the other four rows contain the marble positions per player, this row accumulates all fields that are occupied by marbles. The drawback is that the model with a fifth row is 25% larger and requires extra resources.

Perspective
The perspective means from which point of view the representation is created. In one perspective, the *generic perspective*, the first row in the model represents the move player, the marble positions, and the card counts of the player with index 0 (player North), the second row the player with index 1 (player East) and so on. A second perspective, the *player perspective*, starts with the currently observing player at index 0, the next clockwise player at index 1 and so on.

Agents that are trained with the *generic perspective* are board position specific. An agent trained at position North has position specific knowledge that results in bad moves at the other board positions. This is unnecessary given the symmetry of the board-card games. A *player perspective* agent should be able to play at all for board positions.

Another reason to experiment with two perspectives is that the evaluation results show that the North-South team is stronger than the East-West team. The mean episode return values for both teams are 0.55 against 0.45. We did not find a cause for this and probably this difference would not exist when using the *player perspective*.

7.3. Discussion

7.3.1. Tournament results

The DMC agents of experiment **A** score a win rate of more than 99% against randomly playing agents and more than 49% against rule-based agents. This is achieved after 49 days of training. Figure 7.2 suggests that the playing strength will increase with more training. It is hard to estimate how the win rate will develop further and where it will end.

The playing strength of the DMC agent of experiment **A** is good enough to play against humans. The rule-based agent is strong enough for experienced human opponents and is matched by the strength of the DMC agent.

7.3.2. MANUAL ANALYSIS

The manual analysis of some games played by DMC agents of experiment **A** at $1.5x10^9$, $4x10^9$, and $10x10^9$ frames gives insights into the training process. The agent learns to play the most effective moves gradually. The earlier agent has learned some beneficial moves and the later agent has progressed further. For example, the agents at $1.5x10^9$ frames know to apply the effective 4-steps-backward-from-start move for the first marble, the two later agents know to perform the effective 4-steps-backward-from-start move for the next marbles too.

The manual analysis results from Table 7.2 show that the agent learns some types of moves faster than others. The first agent has already learned to hit opponents effectively. Later, the agent learned to hit the teammates and their own marbles less. The same can be said about blocking and switching. The agents learned to look further and play more as a team during the learning process. The agents help the teammate and thwart the opponents. The agent at $4x10^9$ has improved to value positions and team play. The latest agent matches the rule-based agent by making hitting opponents a higher priority, hitting fewer own marbles and bringing more anticipation in the game. Better anticipation is expected with more training and with observation models that take the number of cards in the hands of other players into account.

7.3.3. OPTIMIZATIONS

The optimization experiments have a positive effect on the processing speed and the win rate progress. The processing speed of experiment **F**, compared to experiment **A**, increased a little less than 15%. The win rate increased around 3% at the same stage of the training. This estimation of the win rate has more uncertainty as it is unknown how it will develop in later stages of the training.

NETWORK SIZE

The most straight-forward optimization is the size of the NN. Experiments **A** and **B** in Figures 7.5 and 7.7 show that a smaller NN improves the speed, apparently without impact on the win rate. Training with a NN of $[2 \times 512]$ is 15% faster than with a NN of $[5 \times 512]$.

REWARD FUNCTION

A more aggressive reward function with values at an interval of $[26, -26]$ as described in 7.2.6 causes rougher and less performant starts of the trainings that catch up later on. After 2×10^9 frames the reward values show a positive trend that seems to hold. The average game length is decreased by around 5 moves. It is hard to draw conclusions regarding the decreased game length as this can be caused by the other optimizations as well.

OBSERVATION MODEL

Adding the card count is not expected to improve the playing strength in the interval of Figures 7.3 and 7.4 drastically. The manual analysis of the learning process shows that at these stages the agent is still learning important actions that have more practical value, such as preventing to be hit and moving 4 steps backwards, than the number of cards of the other players. It is quite possible that the number of cards of other players is less decisive in the outcome of the games. The hope is that the card counts will make some difference at later stages, especially if players have no cards. To draw conclusions longer training with card counts is required.

Experiment **F**, with the state representation from the player perspective in Figure 7.3, has better balanced mean episode return values. Although the agents of experiment **F** are more board position independent, for yet unknown reasons (start player is selected randomly) they still prefer the board position that they were trained for. The mean episode return values are 1.7 versus -1.7 in favor of the North-South team. A tournament with agents of experiment **F** trained at North and South and positioned at East and West score only 7.6% against rule-based agents (on positions North and South the score is 43%). This is better than the experiments with the *generic perspective*, but not good enough.

Figure 7.5 shows the processing speed of the experiments. One can see that experiment **A** (5 layer NN) is the slowest and experiments **B** and **C** are the fastest. Experiments **D**, **E** and **F** are, not clearly visible due to overlapping, in the middle without much differences. Hence, the smaller NN helps efficient processing the most, while the cart count costs some processing power. This may be caused by the extra numpy `concatenate` and `flatten` operations. Preventing these operations would be worthwhile. Note that a representation of 4 or 5 rows (experiment **D** versus **E**), does not seem to make any difference in processing speed.

7.3.4. Overall

Overall the agent of experiment **F** is the best because it is faster, has a better win rate and the expectations are better on longer trainings (card count and reward function). More experiments might still improve the results:

- Changing parameters of the training, such as learning rate, alpha, exploration
- Changing technical settings, such as number of devices, number of actors, number of threads, batch size, buffer size, replay size, network size
- Changing the action model or other observation model representations
- Steering the training by adding knowledge or extra, intermediate rewards
- More processing power

It would be interesting to see how strong the DMC agent becomes after more training. More training with more processing power, in the cloud or not, is expected to be beneficial. The experiments with the NN size and state representation show that improvements are achievable, but adding more computing power or training time seems the most beneficial way to improve the playing strength.

The use of RL techniques for board-card games depends on more than just the playing strength. RL has advantages and disadvantages, depending on the game and the deployment environment:

- + The agents have an interesting, original playing style.
- + There is no need to manually program the behavior of agents.
- + It is possible to let agents learn while playing.
- - It can take a long time to train a model.

- - RL requires more hardware resources (challenging in mobile environments).

The different characteristics make RL agents an interesting technique in addition to or even as a replacement of rule-based agents. The playing style may be the most important feature, as in general, a learning process of RL does not result in fully predictable, rule-based behavior. The required hardware resources and training time are disadvantages that seem manageable as the experiments were not run on high end hardware (see Section 5.5).

7.4. VALIDITY, VERIFIABILITY AND RELIABILITY

The research of DMC is validated by the results of a significant number of matches between various agents and manual analysis. The match statistics show which RL techniques achieve the strongest gameplay and how it performs against a rule-based agent. Manual analysis reveals the good and bad moves and the playing style.

It seems that the measurements, as described in Section 5.3, cover all aspects of the research. The three types of tournaments, against randomly playing agents, against other versions of DMC agents, and against rule-based agents, measure the performance in many different circumstances. The thousands and thousands of matches guarantee a wide range of game situations are evaluated.

The setup of the experiments with DMC delivers reliable results. The research is repeatable with the available software and the programs run on an average computer. The learning process starts from scratch with a configured seed of 42, which helps to reproduce the results. Tests with tournaments of 1000 or 4000 matches deliver consistent results, with and without a seed. The tournament results and the manual analysis form a cohesive picture. All measurement methods seem to correlate and reflect a step-by-step learning process.

8

CONCLUSIONS AND FUTURE WORK

This chapter answers the research questions based on the results and the discussions from Chapters 6 and 7. Future work is suggested and a reflection on the process of the research project is described thereafter.

8.1. ANSWERS TO RESEARCH QUESTIONS

The main research question was how an RL based agent can achieve the strongest gameplay in board-card games. We experimented with DQN and DMC. The results of the case study with Keezen presented in Section 7.2 show that DMC leads to much better results than DQN. DMC agents trained from scratch scored 99% win rate against random agents and 49% against rule-based agents. Extra experiments were conducted to optimize the results. A smaller NN, a more aggressive reward function and changes to the observation model improve the processing speed and the win rate. As argued in Section 7.3 better results are expected with more training and suggested improvements, especially with more processing power and more training.

Chapter 2 described RL research that is done into incomplete information and cooperative games. This answers research question (i). The previously conducted research on multi-player imperfect information games formed the base of this research and the experiments with DQN and DMC. The research on the Chinese card game Dou Dizhu proved useful for this research.

Chapter 4 described the method to train agents and examine the generated models. All implementations are available for further research. It looks like the Keezen implementation will be included in RLCard. The case study with Keezen could be used as a template for the other board-card games. The analysis of the characteristics of board-card games in Section 3.4 suggests that the results of the case study are valid for the other card-board games. This answers research question (ii).

The strength of the agents was measured by automatic tournament play, manual analysis, and processing time measurements. The measure methods to examine the playing strength and style were described in Section 5.3. This answers research question (iii). Manual analysis shows that the agent learned to look steps ahead and play as a team. The next steps are to keep improving and add even more anticipation to the game.

The answer to research question (iv) is that DMC leads to the best results. DQN and DMC were chosen from the related work as the most promising techniques. Based on the

study of related work, it is not expected and not ruled out totally, that other algorithms might perform better at board-card games. We could not compare with other research into board-card games and we experimented with a small subset of techniques. Hence, we cannot rule out that there might be better solutions than presented in this research.

We conclude that DMC is a great technique for board-card games. The playing strength is good enough and will be even higher. Other properties of RL techniques, such as playing style and no need to manually program the behavior, are interesting enough to be applied to board-card games. There are some disadvantages, such as training time and hardware requirements, that generally seem manageable.

8.2. FUTURE WORK

The results of this research suggest that future work on DMC is worthwhile. Suggestions for future work with DMC are:

- How good can DMC become at board-card games?
  The results with DMC are promising and the results against the rule-based agent could be even better. It would be interesting to investigate how good DMC can become. The conducted experiments with DMC suggest there is room for improvement.

- Optimize DMC
  The current training generates four separate models. The generation of one shared model would be an improvement. The training could benefit from the symmetric nature of the game in two ways. First, sharing and synchronizing the model between agents brings efficiency to the training. Second, fewer resources are required if only 25% of the models are generated.

- Add a history to the training model
  The addition of history in the training model could provide more insights in the game. Processing of the most recent actions could be beneficial for the training process and the playing strength.

- Longer training
  The current status is that more training leads to better play. All experiments show that the agent is still improving.

- New or adapted algorithm
  The applied DMC implementation is relatively new and the developments in RL go fast. Expect the future to bring better solutions that can improve the playing strength of board-card agents.

As argued in Section 3.4, a classification system of games could be beneficial for RL research. Work has been done on the characteristics of games and this research project has looked at the characteristics of board-card games, but there seems to be a place for a structured classification system that shows the similarities and differences of games and their RL research status. This may also be valuable outside the context of games. Such a system would have helped this research. However, it is questionable if it is just beneficial for beginners and how much value it adds to the next project.

Section 3.5 contains a suggestion for further research into the relation of the characteristics of games and the role of skill and chance. Probably, one would be able to relate game characteristics and mechanics to the quantitative outcomes of relative performances.

Although the performance of DQN on board-card games is worse than DMC and expectations for improvements that compete with DMC are low, it could be interesting to see what happens when the techniques described in the rainbow paper [Hessel et al., 2018] are applied. The results could shine more light on the role of the action space in DQN. Possibly DQN is able to achieve good results at board-card games.

8.3. Reflection

Table 8.1 shows a coarse picture of the research process and timelines. This process deviates considerably from the initial planning. This is mainly caused by setbacks during the research process and the decision to keep on searching for a solution.

Table 8.1: Research project process with timeline.

Period	**Activities**
Jul 2019 - today	Self study AI and NN
Aug 2019	Start research preparations
Feb 2020	Finish research proposal.
Mar 2020	Purchase training hardware
Mar - May 2020	Develop game model and rule-based agent
Jun 2020	Proof of concept RLCard + DQN
Jul-Mar 2021	Attempts DQN
Apr-Jul 2021	Develop alternatives: A3C/PPO/DMC
Aug - Nov 2021	Experiments DMC
Nov-May 2022	Process results with DMC, additional experiments and finish project

The process involves a large amount of self-study. Initially, this was to obtain a required knowledge level, later it was necessary to follow the developments in the research area.

After finishing the research proposal, the research project started with the development of the game model, a rule-based agent, and a decision about the training resources. Hardware was purchased. The choice between running my own hardware or running in the cloud is mostly a personal preference with some expectations that a lot of experimenting was necessary. Looking at the amount of training labor during this research with an uncertain outcome, running self-owned hardware was a good decision. Especially if we consider technical difficulties, such as the encountered memory leaks. In the case of a stable solution, like at the end of this project, a cloud solution with more computing power is preferred. Probably, we should have anticipated to run the local project in the cloud later. It would have made it less time consuming to run experiments and evaluations in parallel.

Setting up the environment for the experiment with RLCard went reasonably smoothly, but the results were not as expected. Table 8.1 shows that experimenting with DQN alone took nine months and the development of alternatives, of which A3C and PPO failed, took another 4 months. DQN was expected to give decent results. This led to endlessly training DQN agents and experimenting with varying models, conditions, configurations, and parameters without much better results. Meanwhile we were trying to rule out errors and

questioning the assumptions.

We even considered to stop the project and report what did not work. Because it was fun and interesting we extended the project with the expectation of better results. In consultation with the supervisors, we decided to shift deadlines. Fortunately, better results came. After that we conducted extra experiments to try to optimize the results with DMC and did more work on the theory.

With today's knowledge, we should have stopped the experiments with DQN earlier. In hindsight, it took a long time to set the expectations aside while still doubting if errors in the setup were made. Besides this, we could have given the paper [Hessel et al., 2018] more attention, although it is more focused on Atari games. Based on this research, one would expect that the results improve slightly and not come near DMC because the characteristics of board-card games better match DMC. Of the rainbow techniques Double DQN [Van Hasselt et al., 2016] and Dueling DQN [Wang et al., 2016] seem most interesting for board-card games.

Part of the process was the identification of risks. All three risks were spot on:

Risk 1: Framework
Dependency on a framework introduces a risk. Things might not work out as promised upfront.

Risk 2: Training time
A well-known risk in RL is the amount of required training time to reach acceptable results. Buying the required hardware or applying a cloud solution are valid options.

Risk 3: Unexpected results
The RL techniques might not achieve the intended results. Despite a substantiated research design, the algorithms may fail in practice or need extensive tuning that runs out of schedule. The simple game implementation, which is part of the framework investigation in the first phase, will not mitigate this risk. This risk can be accepted (and reflected in the conclusions) or mitigated by investing more time. The path to follow will be discussed with the supervisor.

All three identified risk events were realized to a greater or lesser extent.

The first risk manifested itself on several occasions, although we did a proof-of-concept with RLCard early to mitigate this risk. The version of RLCard with DMC broke the code developed for DQN. On several occasions, memory leaks occurred. Some were hard to find and fix as they appeared in rare game positions or after hours of training. Later in the research project experimenting with another framework, Tf-agents with A3C and PPO, failed and took a lot of time.

The second risk manifested itself very clearly during the project. AI research may take a long time because of a loop of two time-consuming activities. Training a model takes time, and unexpected, interesting results take time to evaluate and explain or fix. To shorten the loop, investing in shortening the training time proved valuable. The training was made fail-fast by tweaking parameters and the size of the NN. Combined with training experience, the time to conclude whether changes worked or not was reduced by almost half. If possible, parallelization of training processes and powerful hardware will pay off.

The third risk came true in experimenting with DQN. We had not previously identified the characteristics of board-card games as problematic for DQN. It would have saved a lot of time.

BIBLIOGRAPHY

Majed Alhajry, Faisal Alvi, and Moataz Ahmed. TD (λ) and Q-learning based ludo players. In *2012 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 83–90. IEEE, 2012. 13

Karl J Astrom. Optimal control of Markov processes with incomplete state information. *Journal of mathematical analysis and applications*, 10(1):174–205, 1965. 6

Nolan Bard, Jakob N Foerster, Sarath Chandar, Neil Burch, Marc Lanctot, H Francis Song, Emilio Parisotto, Vincent Dumoulin, Subhodeep Moitra, Edward Hughes, et al. The hanabi challenge: A new frontier for AI research. *Artificial Intelligence*, 280:103216, 2020. 15

Marc G Bellemare, Will Dabney, and Rémi Munos. A distributional perspective on reinforcement learning. In *International Conference on Machine Learning*, pages 449–458. PMLR, 2017. 12

Richard Bellman. A Markovian decision process. *Indiana Univ. Math. J.*, 6:679–684, 1957. ISSN 0022-2518. 5

Hans Berliner. The b* tree search algorithm: A best-first proof procedure. In *Readings in Artificial Intelligence*, pages 79–87. Elsevier, 1981. 15

Peter Borm and Ben van der Genugten. On a relative measure of skill for games with chance elements. *Top*, 9(1):91–114, 2001. 23, 24

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016. 25

Noam Brown and Tuomas Sandholm. Superhuman AI for heads-up no-limit poker: Libratus beats top professionals. *Science*, 359(6374):418–424, 2018. 15

Chenghui Cai and Silvia Ferrari. A Q-learning approach to developing an automated neural computer player for the board game of Clue®. In *2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence)*, pages 2346–2352. IEEE, 2008. 14

Henry Charlesworth. Application of self-play reinforcement learning to a four-player game of imperfect information. *arXiv preprint arXiv:1808.10442*, 2018. 14

Peter Duersch, Marco Lambrecht, and Joerg Oechssler. Measuring skill and chance in games. *European Economic Review*, 127:103472, 2020. 24

George Skaff Elias, Richard Garfield, and K Robert Gutschera. *Characteristics of games*. MIT Press, 2012. 19, 20

Timothy Furtak and Michael Buro. Recursive Monte Carlo search for imperfect information games. In *2013 IEEE Conference on Computational Inteligence in Games (CIG)*, pages 1–8. IEEE, 2013. 15

Laura Graesser and Wah Loon Keng. *Foundations of deep reinforcement learning: theory and practice in Python*. Addison-Wesley Professional, 2019. 4, 11

Johannes Heinrich and David Silver. Deep reinforcement learning from self-play in imperfect-information games. *arXiv preprint arXiv:1603.01121*, 2016. 13, 49

Matteo Hessel, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018. 12, 69, 70

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997. 11

Carina Huchler. A MCTS agent for Ticket to ride. *Master thesis. Maastricht University*, 2015. 15

Leslie Pack Kaelbling, Michael L Littman, and Anthony R Cassandra. Planning and acting in partially observable stochastic domains. *Artificial intelligence*, 101(1-2):99–134, 1998. 6

Deepak Khemani and Shikha Singh. Contract bridge: Multi-agent adversarial planning in an uncertain environment. In *Poster Collection of the Sixth Annual Conference on Advances in Cognitive Systems. ACS (Online available at www. cogsys. org/papers/ACSvol6/posters/Khemani. pdf)*, 2018. 14

Vijay R Konda and John N Tsitsiklis. Actor-critic algorithms. In *Advances in neural information processing systems*, pages 1008–1014, 2000. 13

Xenou Konstantia, Georgios Chalkiadakis, and Stergos Afantenos. Deep reinforcement learning in strategic board game environments. EUMAS: European Workshop on Multi-Agent Systems, 2019. 15

Marc Lanctot, Edward Lockhart, Jean-Baptiste Lespiau, Vinicius Zambaldi, Satyaki Upadhyay, Julien Pérolat, Sriram Srinivasan, Finbarr Timbers, Karl Tuyls, Shayegan Omidshafiei, et al. Openspiel: A framework for reinforcement learning in games. *arXiv preprint arXiv:1908.09453*, 2019. 25

Glenn F Matthews and Khaled Rasheed. Temporal Difference learning for nondeterministic board games. In *IC-AI*, pages 800–806, 2008. 13

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013. 12

Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937. PMLR, 2016. 13

Matej Moravčík, Martin Schmid, Neil Burch, Viliam Lisỳ, 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, 2017. 15

Matthew Sadler and Natasha Regan. Game changer. *AlphaZero's Groundbreaking Chess Strategies and the Promise of AI. Alkmaar. The Netherlands. New in Chess*, 2019. 24

Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015. 12

Stephan Schiffel. Symmetry detection in general game playing. In *Twenty-fourth AAAI conference on artificial intelligence*, 2010. 21

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017. 13

Brian Sheppard. World-championship-caliber scrabble. *Artificial Intelligence*, 134(1-2): 241–275, 2002. 15

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484, 2016. 14

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017a. 15, 26

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017b. 15

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018. 1, 2, 15

Stefan Steinerberger. On the number of positions in chess without promotion. *International Journal of Game Theory*, 44(3):761–767, 2015. 22

Nathan Sturtevant. An analysis of uct in multi-player games. *ICGA Journal*, 31(4):195–208, 2008. 15

Richard S Sutton and Andrew G Barto. A Temporal-Difference model of classical conditioning. In *Proceedings of the ninth annual conference of the cognitive science society*, pages 355–378. Seattle, WA, 1987. 7

Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018. 4, 9, 11

Gerald Tesauro. TD-Gammon, a self-teaching backgammon program, achieves master-level play. *Neural computation*, 6(2):215–219, 1994. 14

Gerald Tesauro. Temporal Difference learning and TD-Gammon. *Communications of the ACM*, 38(3):58–68, 1995. 14

Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with Double Q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016. 12, 70

Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. Dueling network architectures for deep reinforcement learning. In *International conference on machine learning*, pages 1995–2003. PMLR, 2016. 12, 70

Christopher John Cornish Hellaby Watkins. Learning from delayed rewards. 1989. 9

Chih-Kuan Yeh, Cheng-Yu Hsieh, and Hsuan-Tien Lin. Automatic bridge bidding using deep reinforcement learning. *IEEE Transactions on Games*, 10(4):365–377, 2018. 14

Yang You, Liangwei Li, Baisong Guo, Weiming Wang, and Cewu Lu. Combinational Q-learning for dou di zhu. *arXiv preprint arXiv:1901.08925*, 2019. 14, 49

Daochen Zha, Kwei-Herng Lai, Yuanpu Cao, Songyi Huang, Ruzhe Wei, Junyu Guo, and Xia Hu. RLCard: A toolkit for reinforcement learning in card games. *arXiv preprint arXiv:1910.04376*, 2019. 25

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. URL http://proceedings.mlr.press/v139/zha21a.html. 12, 14, 30, 31, 49

Martin Zinkevich, Michael Johanson, Michael Bowling, and Carmelo Piccione. Regret minimization in games with incomplete information. *Advances in neural information processing systems*, 20:1729–1736, 2007. 13

ACRONYMS

A3C Asynchronous Advantage Actor-Critic.

AI Artificial Intelligence.

CFR Counterfactual Regret Minimization.

CNN Convolutional Neural Networks.

DMC Deep Monte Carlo.

DQN Deep Q-Networks.

LSTM Long Short-Term Memory.

MCM Monte Carlo Methods.

MCTS Monte Carlo Tree Search.

MDP Markov Decision Process.

ML Machine Learning.

MLP Multi Layer Perceptrons.

NFSP Neural Fictitious Self-Play.

NN Artificial Neural Network.

POMDP Partially Observable Markov Decision Process.

PPO Proximal Policy Optimization.

RL Reinforcement Learning.

RNN Recurrent Neural Networks.

TD Temporal Difference learning.