TU WIEN Informatics

Optimierung der Entwicklung und Regelbalance von Brettspielen mittels Reinforcement Learning

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Data Science

eingereicht von

Maximilian Lackner, BSc.
Matrikelnummer 11914689

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Associate Prof. Dipl.-Ing. Dr.techn. Nysret Musliu

Wien, 29. April 2025

Maximilian Lackner	Nysret Musliu

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TU WIEN Informatics

Optimizing Board Game Design and Rule Balancing through Reinforcement Learning

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Data Science

by

Maximilian Lackner, BSc.
Registration Number 11914689

to the Faculty of Informatics

at the TU Wien

Advisor: Associate Prof. Dipl.-Ing. Dr.techn. Nysret Musliu

Vienna, April 29, 2025

Maximilian Lackner	Nysret Musliu

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Maximilian Lackner, BSc.

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Overview of Generative AI Tools Used“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, habe ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT-Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 29. April 2025

Maximilian Lackner

Danksagung

An dieser Stelle möchte ich mich bei allen bedanken, die mich während der Anfertigung dieser Arbeit unterstützt haben.

Mein besonderer Dank gilt meinem Betreuer, Assoc. Prof. Dipl.-Ing. Dr.techn. Nysret Musliu, für seine wertvolle Unterstützung, die konstruktiven Rückmeldungen und die Gelassenheit, die er während des gesamten Prozesses gezeigt hat.

Ein weiterer herzlicher Dank geht an meine Frau, deren Unterstützung, Geduld und Motivation mich während der gesamten Diplomarbeit getragen haben. Ohne ihre unermüdliche Hilfe und ihr Verständnis in vielen schwierigen Momenten wäre diese Arbeit nicht möglich gewesen.

Ich möchte mich auch bei den zahlreichen Spieltestern und Spieltesterinnen bedanken, die bereit waren, ihre Zeit und Energie zu investieren, um gegen mehrere unterschiedliche Modelle zu spielen. Ihre Rückmeldungen und ihr Engagement waren entscheidend für die Weiterentwicklung des Projekts und haben maßgeblich zur Qualität der Arbeit beigetragen.

Abschließend danke ich allen, die mich auf irgendeine Weise unterstützt oder ermutigt haben – sei es durch Gespräche, Ratschläge oder andere Formen der Hilfe. Eure Unterstützung bedeutet mir viel und hat mir geholfen, diese Herausforderung erfolgreich zu meistern.

Acknowledgements

I would like to take this opportunity to express my gratitude to all those who supported me during the preparation of this thesis.

My special thanks go to my supervisor, Assoc. Prof. Dipl.-Ing. Dr.techn. Nysret Musliu, for his valuable support, constructive feedback, and the calmness he demonstrated throughout the entire process.

I would also like to extend my heartfelt thanks to my wife, whose support, patience, and motivation carried me through the entire thesis process. Without her tireless help and understanding during many challenging moments, this work would not have been possible.

I would like to thank the numerous playtesters who were willing to invest their time and energy to play against various different models. Their feedback and commitment were crucial for the further development of the project and have significantly contributed to the quality of this work.

Finally, I would like to thank everyone who has supported or encouraged me in any way – whether through discussions, advice, or other forms of help. Your support means a lot to me and has helped me successfully complete this challenge.

Kurzfassung

In den letzten Jahren hat sich Reinforcement Learning als ein leistungsstarkes Werkzeug zur Optimierung von Strategien in Brettspielen etabliert, das es Künstlichen Intelligenzsystemen ermöglicht, selbst die besten menschlichen Spieler herauszufordern. Diese Arbeit untersucht die Anwendung von Reinforcement Learning zur Unterstützung der Brettspielentwicklung, mit besonderem Fokus auf die Spiele *„Noch mal!“* und dessen Erweiterung *„Noch mal so gut!“* sowie das eigens entwickelte Spiel „Simala“. Durch das Training von Reinforcement Learning Agenten, die diese Spiele spielen, wird untersucht, inwieweit Künstliche Intelligenz zur Regelbalancierung beitragen, Schwächen aufdecken und das Spielverhalten optimieren kann.

Durch die Analyse von Spielabläufen, die von den Reinforcement-Learning-Agenten generiert werden, bewertet die Arbeit, ob bestimmte Regeländerungen in der *„Noch mal!“*-Reihe gerechtfertigt werden können, und untersucht die strategischen Auswirkungen neuer Mechaniken, die in *„Noch mal so gut!“* eingeführt wurden. Darüber hinaus wird erforscht, wie Reinforcement Learning den Test- und Entwicklungsprozess eines neuen Spiels wie „Simala“ beschleunigen kann, indem es optimale Regelkonfigurationen identifiziert und die Dominanz einer einzelnen Strategie verhindert.

Die Ergebnisse zeigen, dass Reinforcement-Learning-Agenten effektiv gegen menschliche Spieler konkurrieren können und dabei strategische Unterschiede und Schwächen in den untersuchten Spielen aufdecken. Die Ergebnisse deuten darauf hin, dass Reinforcement Learning nicht nur das Spieldesign verbessert, sondern auch einen datengestützten Ansatz für iteratives Spieletesten bietet, der einen effizienteren und ausgewogeneren Entwicklungsprozess ermöglicht. Insgesamt zeigt diese Arbeit das Potenzial von Reinforcement Learning im Bereich des Künstlichen Intelligenz-unterstützten Spieldesigns und verdeutlicht dessen Nutzen für die Entwicklung, das Testen und die Optimierung von Brettspielen.

Abstract

In recent years, Reinforcement Learning has emerged as a powerful tool for optimizing strategies in board games, enabling Artificial Intelligence systems to challenge even the most skilled human players. This thesis explores the application of Reinforcement Learning to enhance board game development, with a particular focus on the games *"Noch mal!"* and its extension *"Noch mal so gut!"*, as well as the custom game "Simala". By training Reinforcement Learning Agents to play these games, the thesis investigates the potential of Artificial Intelligence to assist in rule balancing, identify weaknesses, and optimize gameplay mechanics.

Through the analysis of gameplay data generated by the Reinforcement Learning Agents, the study evaluates whether certain rule changes in the *"Noch mal!"* series can be justified and assesses the strategic impact of new mechanics introduced in *"Noch mal so gut!"*. Additionally, the thesis explores how Reinforcement Learning can accelerate the testing and development of a new game like "Simala" by identifying optimal rule configurations and preventing the dominance of any single strategy.

The results show that Reinforcement Learning Agents can compete effectively against human players, uncovering strategic differences and weaknesses in the games analyzed. The findings suggest that Reinforcement Learning not only enhances gameplay design but also provides a data-driven approach to iterative game testing, allowing for a more efficient and balanced game development process. Overall this thesis shows the potential of Reinforcement Learning in the field of Artificial Intelligence-supported game design and shows its benefits for the development, testing, and optimization of board games.

Contents

CHAPTER 1

Introduction

In recent years, methods for solving board games have improved considerably thanks to advances in hardware and new algorithms. While early computer systems were primarily designed to correctly execute a game to the end, today's goal has increasingly become to outperform even human master players [Pla20]. A classic example of the use of Reinforcement Learning (RL) in board games is the training of machines to play *Chess*, *Shogi* (Japanese form of chess), or *Go* [SHS+18]. Machine Learning (ML), in particular RL, is used specifically to develop optimal strategies for known games. As early as 1959, Arthur Samuel formulated the idea that a machine "[...] can learn a better game of checkers than the programmer himself can master" [Sam59]. This vision is still being pursued in research today, with the aim of developing powerful game algorithms beyond these classic titles, showing that RL is not limited to traditional abstract games but is increasingly relevant for modern board games with richer rule sets and more varied decision spaces [YBT+23, XCA19].

However, RL also opens up new possibilities beyond the mere optimization of playing strength: It can be used not only to develop particularly strong players, but also to analyze playing styles or even to support the development of new board games. One notable example is the work of Paolini et al. [PMVI24], who not only trained a strong agent for *7 Wonders Duel* but also identified a significant first-player advantage, which led to rule modifications that improved balance. Similarly, Hom and Marks [MH07] investigated how rule variations affect balance in games like *Tic-Tac-Toe* and *Reversi*, applying Genetic Algorithms to explore the design space. These approaches emphasize that the goal of using Artificial Intelligence (AI) in games is not always to win, but to understand and improve the game itself.

Nowadays, creating a board game is a challenging process. Initially, the game designer must conceptualize the core idea of the game, followed by the development of its rules and mechanics. Additionally, considerations regarding graphical representation and player interaction must be addressed. However, the most challenging phase begins after this

initial development. At this stage, the game designer must ensure that all potential edge cases are properly addressed, that the game is well-balanced, and that it remains engaging and enjoyable for players. Achieving these goals requires rigorous playtesting. This process is very time consuming, because multiple games have to be played and feedback must be collected. After the first iteration of playing, the rules have to be changed based on the feedback of the testers and the testing must start from the beginning. All this has to been done till the developer considers the game to be good enough, which is hugely influenced by their bias.
In the above example, the game developer has to rely on iterative testing and player feedback which can be biased as well. Most likely, they have about 30 contestants play the game over and over again while also collecting their opinions on which rules are not yet balanced or which strategy should be weakened. Afterwards, they try to incorporate all the feedback while also assessing which of the feedback is just, simply due to a particular game situation and which is a general flaw in the game.

In the age of ML and AI, a natural question arises: Can this process be improved with the help of a computer? What if a computer could test the game for the developer, reducing their role to merely evaluating whether they like a particular outcome or not? If the results are unsatisfactory, the developer could simply adjust the rules and let the computer test the game again under the new conditions. This approach would not only be faster but also more data-driven, as rule adjustments could be based on a much larger set of gameplay data compared to traditional playtesting with human testers. Additionally, by running a significantly higher number of simulations, the likelihood of identifying unresolved edge cases increases.

However, the main challenge of this approach lies in ensuring that the computer understands and plays the game well. If the AIs were to make only random moves, the generated data would be of little value to the developer, as no meaningful strategies could be analyzed. Attempting to balance game rules based on such flawed data would be both ineffective and frustrating. Furthermore, modifying the game based on the decisions of a poorly performing AI could lead to unnecessary changes, potentially disrupting a well-designed game simply because the AI failed to discover optimal strategies. To explore this challenge further, we shall examine a concrete example.

Example 1.0.1 *Suppose there is a game that you play on a 4×4 grid where you have to try to get from the top left to the bottom right corner with as few moves as possible. The rules are simple: you can only move one step at a time, either horizontally or vertically, and you cannot move diagonally or step outside the boundaries of the grid. The player with the fewest moves wins.*

Imagine we only have a bad playing agent and we ourself do not know the best path, so that we are forced to rely on the strategy of the computer. For example, the agent only finds the red path in Figure 1.1 as the best path, the developer would maybe change the game rules by adding an obstacle in the right upper corner, to make the game more

difficult. Then, the computer has to adjust and maybe find another path like the blue one, which is, again, clearly not the best one. If the developer still considers the game not good enough, they would again adjust the rules based on the current path. This process could be repeated until the game appears to meet the desired expectation.

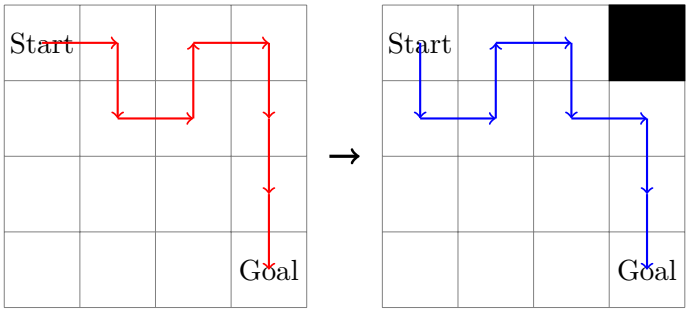


Figure 1.1: An example of a rule change based on a non-optimal path.

As is apparent in this simple example, the new rule setting with the obstacle would not have changed anything for a good player as can be seen in Figure 1.2. Since one of the best paths would have been the green one, among others, the changes would not have affected the gameplay.

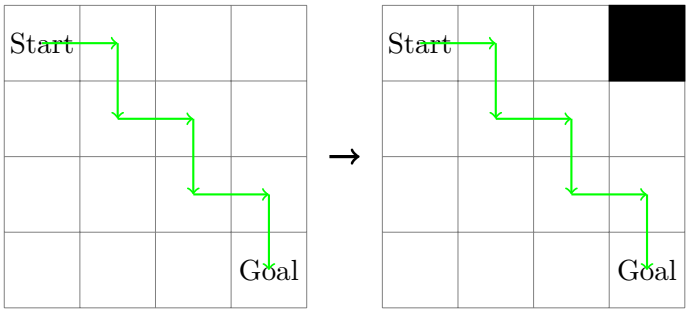


Figure 1.2: An example of an optimal path, demonstrating that the rule change had no impact on it.

In traditional board games, such examples are more difficult to identify. However, it is clear that a computer must be a decent player in order to serve as a reliable game tester for rule balancing. If a computer discovers the optimal strategy immediately, the game developer might need to reconsider the rules entirely. This could even lead to a fundamental change in the game's objective, such as redefining the goal to: *you have to try to get from the top left to the bottom right corner with as* ***many*** *moves as possible.*

1.1 Goal and Research Questions

The main aim of this thesis is to analyze the existing board games *"Noch mal!"* and its extension *"Noch mal so gut!"*, designed by Inka and Markus Brand, from a game

development perspective. Specifically, we will investigate whether certain rule changes introduced in the extension can be justified based on data generated by our Reinforcement Learning Agent (RLA). Furthermore, we examine whether some of the new options in *"Noch mal so gut!"* are inherently weaker than others by analyzing how frequently the agent chooses them in competitive gameplay.

Additionally, an objective of this thesis is to explore how RL can accelerate the testing process for a newly developed board game called "Simala". Since "Simala" is still in development, we will leverage insights from our RLA to refine the game's rules, ensuring a balanced win rate and preventing any single strategy from becoming dominant.

Research questions:

- How effectively can our RLAs compete against experienced human players in terms of win rate in the games of *"Noch mal!"* and *"Noch mal so gut!"*?
- Which of the new options introduced in the extension *"Noch mal so gut!"* are used more frequently by the agent and how does the strategy differ from the original game *"Noch mal!"*?
- How does the win rate of the starting player and the diversity of the strategy of the agent in the game "Simala" change if we modify the rules based on the data of the trained RLA?

1.2 Challenges and Contributions

One of the key challenges is ensuring that the RLA is sufficiently skilled in order for its data to be reliable during game analysis. From a developer's perspective, a new game should not exhibit a clearly dominant strategy early in its design process. To assess the agent's competence in existing games, we let it compete against various experienced human players. For the undeveloped game "Simala", we conduct direct playtesting against the agent ourselves, mirroring the way a game developer evaluates a game by competing against human testers. Ultimately, our observations confirm that suitably trained agents perform well enough for their gameplay data to be used in analyzing and refining the game.

Our contributions therefore are:

- We build and train RLAs for the games *"Noch mal!"* and *"Noch mal so gut!"* that win more than 50% of their games against human players.
- We identify balance weaknesses in the games of *"Noch mal!"* and *"Noch mal so gut!"*.
- We analyze the strategic differences between *"Noch mal!"* and its extension *"Noch mal so gut!"* by examining the choices made by our RLAs.

- We determine which of the new mechanics in *"Noch mal so gut!"* are actually beneficial for winning and which are less impactful.
- We develop and train RLAs to test different rule sets for our custom board game "Simala", demonstrating how RL can be used as a tool in board game design.
- We use RL-driven insights to iteratively balance "Simala", ensuring fair win rates and diverse strategies.
- We provide a playable prototype of "Simala" that integrates findings from RL experiments, serving as a case study for AI-assisted game design.

1.3 Structure of the Thesis

In Chapter 2 we explain the basics of ML and RL and also dive deeper into existing research related to board games and rule balancing. After that, Chapter 3, is dedicated to the games *"Noch mal!"* and *"Noch mal so gut!"* where we describe the models that were built and also evaluate them. In Chapter 4, the game "Simala" is analyzed and all the development steps taken with the help of RL are described. Chapter 5 concludes the thesis by summarizing the key findings and proposing directions for future research.

CHAPTER 2

Preliminaries

In this chapter, we explore the fundamentals of RL and its connection to games. We begin by defining key terms, introducing relevant research areas, and outlining the core concepts essential for understanding the chapters that follow. Additionally, we present an overview of various RL methods and discuss the reasons for choosing Q-Networks and Deep Q-Networks (DQNs) as the focus of this thesis.

2.1 What is Reinforcement Learning?

The simple answer to this question is: the "normal" way of learning, by interacting with the environment. If we think back to when we were toddlers, we can ask ourselves the question: How did we learn to walk? It was not by reading tons of literature or asking others how they do that, it was just "learning by doing". A baby gets up at some point and falls over. The next time it gets up, it will adjust a few things because it realizes that it fell over. It will do this again and again until it can stand. Then it will take its first steps and it will fall again, but each time it will learn to adapt to its surroundings and find more balance for the next step until it can eventually walk. In a simplified way this is RL. It is a way of learning by interacting. And we encounter this kind of learning quite often in our lives. As [SB18] explains, we are very aware of how our environment reacts to our actions and we try to influence what happens through our behavior, whether we learn to drive a car or hold a conversation.

But to gain a deeper understanding of what RL truly entails, it can be helpful to first explore ML, of which it is a subfield. Nowadays, ML can be divided into three main areas based on [SB18]:

- *supervised learning*: A method of learning from a training set of labeled examples, where each example is associated with a result provided by an informed external

supervisor. The idea is to take specific situations along with their corresponding labels, and the system seeks to identify patterns in the data. This allows the system to generalize its responses and determine the appropriate action for situations that were not included in the presented training set. This is a very important way of learning, but for our purposes not useful, because for games, which are not developed yet, there is no given data.

- *unsupervised leaning*: A method of learning from a training set of unlabeled examples. The main task for the system is to find structure or clusters in the collections of unlabeled data to classify new data correctly. Again, this approach is not useful for our purposes, because no data is given.

- *reinforcement learning*: A method of learning from interacting with the environment. Based on a reward function the system learns which action is the best and tries to find the actions which yield the most reward by trying them out. This is ideal for not developed games, because the system can try these actions by playing the game again and again to find the actions with the most reward in every situation.

The Figure 2.1 represents the learning loop of a RLA. As [Sew19] describes, the agent gets a state S_t from the environment and takes the best possible action A_t at step t. This action changes the state S_t to S_{t+1} as illustrated in the figure. It also generates a reward R_t, which is given to the agent. Then the agent again takes the action it deems best for the given state S_{t+1} and receives the reward R_{t+1}. This loop is repeated over and over again and the agent tries to learn which action would be the best in the given situations, factoring in both immediate and future rewards..

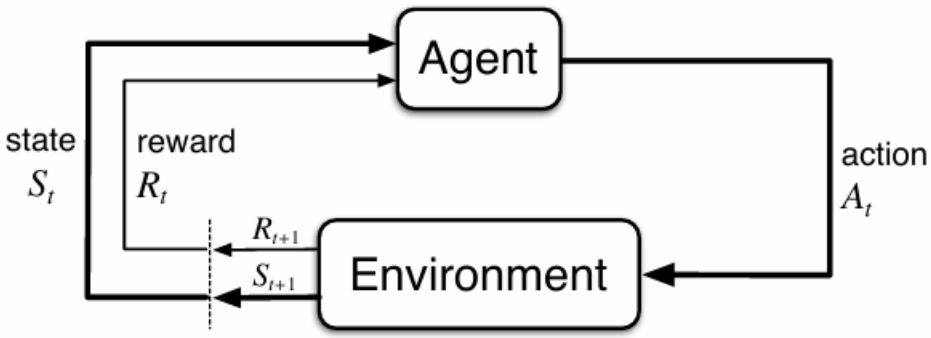


Figure 2.1: The agent environment interaction in RL by [SB18].

The task of the environment is to map different situation as well as possible for the RLA and to give a reward or penalty based on the outcome of the action. It is important to note that the reward depends on the action and the state, so that the situation is relevant for the action that was taken. In the end, the agent will learn from tuples of the form (S_t, A_t, R_t, S_{t+1}), known as *state-action-reward-state* tuples.

To provide a better understanding of these key terms, the following sections will explain each of them in detail.

2.1.1 Reward

The reward is the crucial part of how the RLA learns to separate good actions from bad actions. Since the agent always tries to make the action with the most expected reward, the reward function must be well-balanced to teach it what to strive for. However, as stated in [Sew19], the search for such a function is not trivial, as it requires considering numerous factors that influence the agent's learning and decision-making process.
The first problem to consider is that the reward may not always be realized immediately in the situation. Some strategies need several specific actions in a row to get a big reward. For games with points it is pretty simple to see that not every action which yields the most points in a given turn also yields the most points in the long run. Maybe in combination with the following three turns the total reward could be much higher. To address this problem, we will later introduce the discount factor, which weights the importance of future rewards.
Additional complexity for the reward function comes with probabilistic and uncertain rewards. If the reward only comes with a certain probability, e.g., depending on a lucky dice roll, it will be difficult to capture it in a function.
Another question is also how much actions in the past are responsible for a given reward and how much this should be taken into account. If we can only get low rewards because our actions in the past have put us in such a situation, how much should the current actions be penalized?

One can handle all of these problems differently, but in the end [Sew19] comes up with one of the biggest questions "How lager should the magnitude of the reward be?" If the RLA takes a good action, should the reward be two or maybe ten times bigger than for other actions? All of these decisions are relevant for the outcome of the final agent, because by tweaking details, the behavior of the agent may change drastically.

In many games, reward functions typically fall into one of two main categories. The first is an immediate reward, given instantly after an action is performed, often aligning with the game's predefined scoring system. This type of reward helps guide short-term decision-making. The second is a delayed reward, which is only received at the end of the game and often depends on the overall outcome, such as winning or achieving specific objectives. This delayed reward provides a broader perspective on long-term strategy, encouraging the agent to make decisions that may not yield immediate benefits but contribute to ultimate success. Often they are combined as we see in the research of [YBT+23].

2.1.2 State

The state is the situation the RLA gets before choosing an action. The idea is to include everything that may affect the outcome of the scenario and could be measured. In a

simple way, it should be the best way of representing the current situation of the system. Although there are many different possibilities and there are also many difficulties for real world problems as described in [Sew19], it is usually straightforward for games. Most of the time, the state in games simply describes the current situation, i.e., all information that we as a player would have in a turn should be represented in the state. If there is information in a game that only one player can know (for example, cards in hand), this information is given to the other players only in abstract form (for example, the number of cards). For example in the well known game *Tic-Tac-Toe* the state would simply be the game board with the positions of all placed crosses and circles. For other games like *Ticket to Ride* it is a bit more difficult as can be read in the paper of Yang et al. [YBT+23], because much more information is taken into account here.

2.1.3 Action

The most important piece of the RLA is what it does and especially how it chooses a given option. As we mentioned earlier, the agent always tries to maximize its reward by taking an action based on the current state. Therefore, the term "policy" plays a crucial part. The policy π is a function that associates states with actions. That means it decides how the agents take actions. There are two kinds of policies:

- *deterministic policy*: It returns one action for every given state.

$$\pi : s \mapsto a$$

- *stochastic policy*: It returns probabilities of taking action a in state s.

$$\pi(a|s) = P(a|s)$$

The goal is to find the optimal policy π^* so that for every state s it gives an action $a = \pi^*(s)$ that should be chosen and that leads to a win [Pla20].

Another important function for calculating the best action is the "value function". There are again two ways how a value function could look like according to [Sew19] and [Pla20]. The first is the state-value function $V(s)$, which focuses on identifying which is the best next state to be in. That means it tries to predict the value of every state without seeing every state. This value is mostly a combination of all present but also all future rewards of this state that can be associated with this state.
Secondly, there is the action-value function also called Q-function $Q(s, a)$. It evaluates pairs of state and action and determines the best action by calculating the value based on the action and the state. It is important to notice the difference between that $V(s)$ is just a function of the state and $Q(s, a)$ is a function of both action and state. For this thesis we will use Q-Networks and DQNs, which got their name from this action-value function.

An important challenge when taking actions is the dilemma between "exploring" and "exploiting". Consider Example 1.0.1 from Chapter 1: The RLA has learned that the red

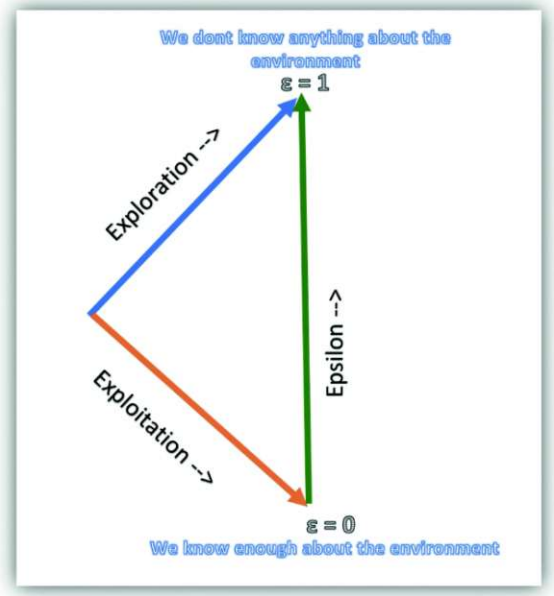


Figure 2.2: Selection of the optimal epsilon value for the problem [Sew19].

path is a good path because it consistently leads to the goal. If the agent never tries any other path, it will always choose this one, as it appears to be the best way to obtain the reward. This behavior is called "greedy" because it always selects the action with the highest known reward probability. However, as seen in the example, it may sometimes be beneficial to explore other options within the action space. By doing so, the agent might discover alternative paths that are equally as good – or even better.
This dilemma is well known in RL and there are various solutions to it, as described in [Pla20]. The most common one is the ϵ-greedy approach, where one mostly takes the best (known) action except that with a probability ϵ one explores the action space, as seen in Figure 2.2. If epsilon is 0.2, then in 80% of the cases the agent takes the best option (exploit), but in 20% of the cases it decides to take a random action (explore). In some cases it is also good to decrease ϵ over time, so that in the beginning of learning the agent explores as much as it can and then tries to figure out which of these options actually turns out to be the best one to use

2.2 Markov Decision Processes and Bellman Equation

Now that we know the basics of RL we can dive deeper into the Markov Decision Process (MDP). A Markov decision problem is a problem where the next state only depends on the current state and the action. This makes the calculations a lot easier, because we do not have to think about the entire past of the process. This means

$$P(s_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, \ldots, s_0, a_0) = P(s_{t+1}|s_t, a_t), \tag{2.1}$$

where t defines the time or turn, i.e., s_{t+1} defines the next state that follows after the action a_t in state s_t. As is apparent, the RL problems in the previous section can be modeled as MDP. In board games, the terms "board position" and "move" are used, whereas in the mathematical formalisms of MDP and RL, these concepts are referred to as "state" and "action", respectively, which we already discussed. According to [Pla20], a MDP consists of the following five elements:

- *Set of States* S: The environment or system can be in one of several states.
- *Set of Actions* A: In each state, an agent can choose from a set of possible actions.
- *Transition Probabilities* $P(s_{t+1}|s_t, a_t)$: The probabilities of transitioning from state s_t to state s_{t+1} when taking action a_t.
- *Reward Function* $R(s_t, a_t, s_{t+1})$: The rewards the agent receives for moving from state s_t to s_{t+1} when taking action a_t.
- *Discount Factor* γ: A value between 0 and 1 that determines how much less future rewards are valued compared to immediate rewards.

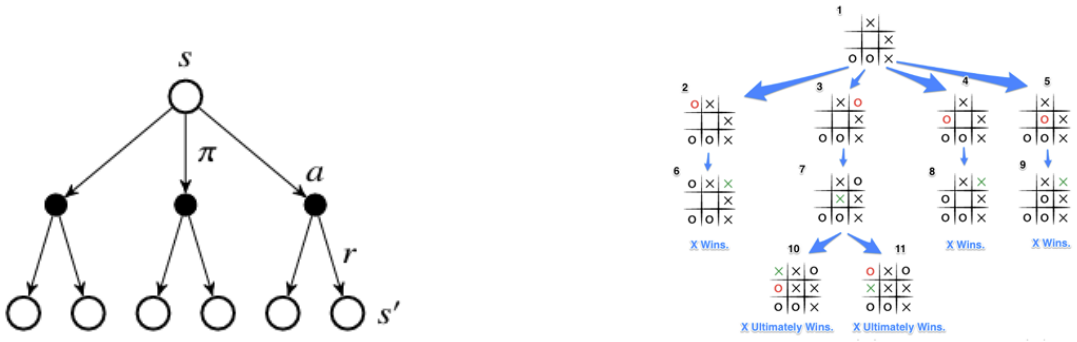
As we can observe, there are numerous parallels to the concepts we have explored thus far. Figure 2.3a illustrates the MDP tuple (S, A, P, R, γ) along with s' as s_{t+1} and π demonstrating how the value can be computed. The root node at the top denotes the state s_t as s, where the policy π enables the agent to choose from three possible actions a_t as a. Each action, based on the transition probability distribution p, leads to one of two possible successor states s_{t+1} as s', each associated with a reward r_t as r. The value of the root state is then determined through a backup procedure. Such a state space can also easily be shown as a directed graph, as can be seen in Figure 2.3b for the well known game *Tic-Tac-Toe*. In this game $P = 1$, because every action leads to a single successor state s_{t+1}.
The discount factor γ in RL determines the importance of future rewards versus immediate ones. A value close to 0 prioritizes short-term rewards, while a value close to 1 emphasizes long-term rewards. Adjusting γ controls the trade-off between short-term and long-term goals in an agent's learning.

The goal of such a problem is to maximize the total sum of discounted rewards. As stated by [Sew19], this involves finding a policy π that achieves this objective.
In other words, we want to find the optimal policy π^* that maximizes the sum:

$$\sum_{t=0}^{\infty} \gamma_t R(s_t, a_t, s_{t+1}), \tag{2.2}$$

where t represents the time, which corresponds to turns in a game.

There are a lot of methods, which try to solve such MDPs, but the most well known is the Bellman Recursion. The Bellman Equation provides a recursive decomposition



(a) A backup diagram with MDP tuple.

(b) A game tree for *Tic-Tac-Toe*.

Figure 2.3: Representations of MDPs from [SB18] and [Pla20].

of the optimal value function, which is a fundamental concept in solving MDPs. The value function, denoted as $V(s)$, represents the expected sum of discounted rewards when starting from state s and following an optimal policy π^*. It satisfies the Bellman optimality equation

$$V(s_t) = \max_{a_t \in A} \sum_{s_{t+1} \in S} P(s_{t+1}|s_t, a_t)[R(s_t, a_t, s_{t+1}) + \gamma V(s_{t+1})]. \tag{2.3}$$

This equation expresses the fact that the optimal value of a state is the maximum expected return that can be achieved by selecting the best action a_t, considering both the immediate reward and the discounted value of the next state s_{t+1}. Similarly, the Bellman equation (2.3) can also be expressed for the action-value function $Q(s_t, a_t)$, which represents the expected return of taking action a_t in state s_t and following the optimal policy:

$$Q(s_t, a_t) = \sum_{s_{t+1} \in S} P(s_{t+1}|s_t, a_t)[R(s_t, a_t, s_{t+1}) + \gamma \max_{a_{t+1} \in A} Q(s_{t+1}, a_{t+1})]. \tag{2.4}$$

[Sew19] concludes that by solving these equations, we can determine the optimal value function and derive the corresponding optimal policy π^*, which specifies the best action to take in each state.
Several techniques exist to solve MDPs by leveraging the Bellman equation, which are all explained in [Sew19], including:

- Value Iteration: This method updates the value function iteratively by applying the Bellman optimality equation until convergence.
- Policy Iteration: This method alternates between policy evaluation (computing the value function for a given policy) and policy improvement (updating the policy based on the computed values).

- Q-Learning: A model-free RL approach that estimates the action-value function $Q(s_t, a_t)$ and updates it using observed rewards from interactions with the environment.

While classical dynamic programming methods such as Value Iteration and Policy Iteration have been extensively studied in the context of MDPs, they come with significant limitations. Both of these methods require full knowledge of the environment's dynamics, including the transition probabilities $P(s_{t+1}|s_t, a_t)$ and the reward function $R(s_t, a_t, s_{t+1})$. In many real-world or complex simulated environments, such information is either not available or too cumbersome to model accurately. Also these methods rely on the explicit enumeration of the state space. For high-dimensional or continuous state spaces, the computational cost becomes prohibitive due to the so-called "curse of dimensionality" [Bel57]. In contrast, Q-learning uses function approximation, allowing them to generalize across similar states without the need for exhaustive state enumeration. Another benefit of model-free approaches like Q-Networks and DQNs is that they learn directly from sampled experiences, making them much more adaptable to large-scale and dynamic environments as described by [Sew19].
In this thesis, we will focus on Q-learning methods, including Q-Networks and DQNs, which will be discussed in the next section.

2.3 Q-Networks and Deep Q-Networks

A Q-Network is a neural network that serves as a function approximator for the Q-function:

$$Q(s, a; \theta) \approx Q^*(s, a), \tag{2.5}$$

where θ represents the parameters of the network. The network takes a representation of the state s and an action a as input and outputs a Q-value for each possible action. Training is typically performed by minimizing a suitable loss function (e.g., mean-squared error) so that the Q-values produced by the network approximate the optimal Q-values. As described in [SB18], the update is based on the Bellman error, which is defined as the difference between the current Q-value and a target value computed according to the Bellman equation:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\Big[R(s_t, a_t, s_{t+1}) + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\Big], \tag{2.6}$$

where α represents the learning rate, which will be discussed in more detail in further sections. In this case, the learned action-value function Q directly approximates Q^* regardless of the policy being followed. This significantly simplifies the analysis of the algorithm and facilitates early convergence proofs. However, the policy still influences which state-action pairs are visited and updated. As [SB18] notes, this method enables efficient learning even in environments where the full model of the system is not known.

The breakthrough in applying deep neural networks to RL was largely achieved with DQNs [Sew19]. DQN extends the basic idea of a Q-Network by incorporating two key mechanisms that improve training stability in high-dimensional environments:

Experience Replay

Rather than using each experience tuple (s_t, a_t, r_t, s_{t+1}) immediately for training, the RLA stores these tuples in a *replay buffer*. During training, mini-batches are randomly sampled from this buffer. This technique breaks the correlations between sequential samples, leading to more efficient and stable learning [Sew19].

Target Network

A significant issue when computing target values is that the parameters of the Q-network are continuously changing during training. To address this, DQN employs a separate target network with its own parameters θ^-. This target network is updated less frequently (for example, every fixed number of steps) by copying the parameters from the main network. This prevents the Q-values from shifting too rapidly, avoiding feedback loops that destabilize learning. As a result, the target values remain stable for several training iterations and the Equation (2.6) updates to

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\Big[R(s_t, a_t, s_{t+1}) + \gamma \max_{a_{t+1}} \mathbf{Q}(\mathbf{s_{t+1}}, \mathbf{a_{t+1}}; \theta^-) - Q(s_t, a_t)\Big]. \quad (2.7)$$

The decoupling of the network used for estimating Q-values and the one used for computing target values is a key innovation that greatly enhances training stability [Sew19].

2.3.1 Comparison

Both Q-Networks and DQNs use deep neural networks to approximate the Q-function. The crucial difference lies in how training stability is achieved:

- **Q-Networks:** These directly approximate the Q-function using a neural network. The network learns from observed state-action pairs and is updated continuously. However, because the target values and the estimated Q-values come from the same, constantly changing network, training can become unstable.
- **Deep Q-Networks:** In addition to using a deep neural network, DQN integrates both Experience Replay and a Target Network. Experience Replay helps to decorrelate the training data, and the Target Network ensures that target values remain unchanged over multiple updates. Together, these components are essential for achieving stable convergence in complex environments.

Although some implementations that use deep neural networks for Q-function approximation are loosely referred to as "Deep Q-networks", the term *Deep Q-Network* in the literature is typically reserved for methods that explicitly incorporate both Experience

Replay and a Target Network [Pla20].
For our work, we will decide individually for each game which of these two approaches or a combination is appropriate and suitable in terms of computing time and performance.

2.4 Applications of Reinforcement Learning and Related Work

RL is widely recognized for its ability to solve real-world problems by enabling agents to learn autonomously from their decision outcomes, reducing the need for constant expert supervision. This is particularly true for Deep Reinforcement Learning (DRL), which has demonstrated effectiveness in a range of sequential decision-making tasks [YBT+23]. The following fields are only a few of the sectors which have benefited significantly from RL and its advanced form:

- In robotics, RL is extensively used for learning physical-world tasks under practical constraints, such as manipulation [SBS+18], grasping [KIP+18], and legged locomotion [ITF+21].
- In recommendation systems, RL has driven advancements by optimizing long-term user engagement [ACF22].
- In transportation, it has been applied to areas like autonomous driving [KST+22], energy-efficient driving [FZAB21], traffic control [HY22], and vehicle routing [NOST18].
- In manufacturing, RL improves production scheduling [WPW21] and maintenance [LM23].

To get a deeper understanding of these applications and how RL plays a crucial role we refer to three practical examples of the book *Reinforcement Learning: An Introduction* [SB18], which we summarized here:

Example 2.4.1 *Bioreactor [SB18]: In this scenario, RL is used to optimize the temperatures and stirring rates in a bioreactor which produces useful chemicals from nutrients and bacteria. The agent's actions involve setting target temperatures and stirring rates, while the states consist of sensor readings and symbolic inputs representing the ingredients and target chemical. The reward is the rate at which the chemical is produced, guiding the agent to adjust parameters for optimal production.*

Example 2.4.2 *Pick-and-Place Robot [SB18]: Here, RL controls a robot arm in a repetitive task of picking up and placing objects. The agent's actions are the voltages applied to the motors at each joint, and the states are the joint angles and velocities. The reward system encourages success through positive rewards for picking and placing objects, with negative rewards for inefficient movements to promote smoother operations.*

Example 2.4.3 *Recycling Robot [SB18]: A mobile robot uses RL to decide when to search for cans, wait for cans, or return to recharge based on its battery level. The state is determined by the battery's charge, and the rewards are positive when the robot successfully collects cans, or negative if the battery runs out. The agent's decisions affect the robot's behavior in its environment, but it does not control the robot's full system, which includes navigation and gripper control.*

Each of these examples highlights the core strength of RL, its ability to learn from interactions and refine strategies over time, ultimately leading to more effective and intelligent automation across various fields.

There have also been many successes in the gaming domain. Nearly everyone has heard of the well known examples of a RLA playing *Chess*, *Go*, or *Shogi* (Japanese form of chess) etc. [SHS+18] on a high level and their impressive capabilities to discover nearly optimal strategies in complex games. Additionally, newer board games like *The Settlers Of Catan* [XCA19] or *Ticket to Ride* [YBT+23] were tackled. Most of theses papers have one thing in common: finding the best playing algorithm.
However, other studies go one step further, focusing on analyzing winning strategies or the win probabilities to balance the game. Paolini et al. [PMVI24] tried to find the best playing agent for the game *7 Wonders Duel*, which was published 2015, based on RL and also found out that the win probability of the starting player is a staggering 66.8%, by far exceeding the estimate of 55.7% based on games between human players. The rules were subsequently adjusted, yielding a starting-player win rate of 51.6%, which is considered a near-optimal balance in competitive scenarios. An alternative approach was taken by Hom and Marks in their paper "Automatic Design of Balanced Board Games" [MH07], applying Genetic Algorithms to examine game balance through rule variations for games like *Tic-Tac-Toe*, *Reversi*, and *Checkers*, analyzing different settings such as board types or victory conditions to find the most balanced configurations of these games. Extensive research has also focused on identifying the board size that offers the most balanced gameplay experience in *Go* [WKI20].

RL is also used in video games for gametesting, so that test coverage is increased and unintended game play mechanics, exploits, and bugs are discovered [BGTG20]. Specifically for "Match 3" a way was found to use RL to test levels and help the developers to measure the difficulty of levels through automated playtesting of different missions [SKJK20].

CHAPTER 3

Noch mal!/Noch mal so gut!

The purpose of the following chapter is to show how the games *"Noch mal!"* and its extension *"Noch mal so gut!"* by Inka and Markus Brand work and how we build, train, and evaluate RLAs which play the game on an experienced level. We also show if certain rule changes introduced in the extension can be justified based on data, and examine whether some of the new options in *"Noch mal so gut!"* are inherently weaker than others.

3.1 Game Concepts

"Noch mal!" and *"Noch mal so gut!"* are two games of dice, which were designed and released in 2016 and 2019, respectively, by Inka and Markus Brand. *"Noch mal!"* was also published in English under the name *"Encore!"*. Both games are recommended to be played by one to six players at the age of 8 and above. As each player has their own board in these games, for the sake of simplicity we will concentrate on the two-player game in this thesis. In both games the goal is to tick the right fields on the board to collect more points than the opponents. Since *"Noch mal so gut!"* is the extension of *"Noch mal!"* we will first explain the original game and then list the special rules and new options added in the advanced game.

3.1.1 Noch mal! – Game Instructions

Game Contents and Setup

For the original game of *"Noch mal!"* this content is needed [BB16]:

- Each player needs a game sheet, which features a single game board identical for every player where changes will be noted every round, and a pencil to tick the fields on the board, see Figure 3.1.

- Three color dice, where the colors *green, yellow, blue, red*, and *orange* as well as a black *"**X**"* for a joker option can be seen.
- Three number dice, where the numbers 1 to 5 as well as a *"**?**"* for the joker option can be seen.

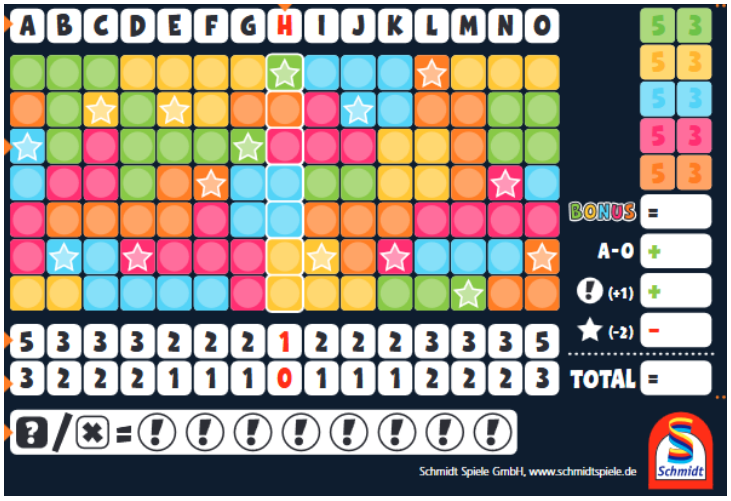


Figure 3.1: The game sheet for each player for the game *"Noch mal!"* [BB16].

Gameplay

Based on the game instructions [BB16], each turn consinst of the same phases:

1. All six dice are rolled.
2. The active player chooses a combination of a color die and a number die, takes them away and ticks fields on their game board according to certain placement rules detailed below.
3. All other players now choose a combination of a color die and a number die from the remaining four dice and also tick fields on their boards. In a game of more than two players, several players may use the same combination of color and number dice (from the four remaining dice).
4. At the end of each turn, every player checks if some colors or columns are finished.

There are two exceptions to these rules:

- In the first three rounds of the game, the dice chosen from the active player are not removed, so that all players can choose from the same set of dice and also are allowed to take the same combinations.

- It is always allowed to pass, so that the players do not choose dice. If the active players passes, the other players can choose from all six remaining dice in Phase 3.

If the players have chosen a combination of a color die and a number die, there are some rules for marking fields on their own board:

- In the first turn of the game, the players have to start in column *"H"*, i.e., the middle column.
- All crosses placed in any one turn must be placed contiguously in one color block of the selected color.
- Crosses may only be placed horizontally or vertically adjacent to at least one box that has already been marked or in column *"H"*. Boxes that only touch diagonally are not considered adjacent.
- It is not permitted to tick fewer or more boxes than the number the chosen number die indicates.
- A color block consisting of several boxes does not have to be completely filled in one turn.
- A player can never check more than five boxes in one turn, even with the number joker.
- In one turn, the number rolled may not be split in order to tick boxes in two different color blocks of the same color.
- If a player has chosen *"**X**"* or *"**?**"*, they can select any color or any number between 1 to 5, respectively. If so, the player has to cross out one of the joker-symbols on their game board, of which only eight are given at the start. It is also allowed to choose both the number and the color joker in one round so that the player can choose from all colors and numbers between 1 to 5, meaning, of course, that two joker-symbols have to be crossed out. If the player does not have enough joker-symbols left, the action is not permitted.

In Figure 3.2 we see on the left two examples of correct moves and on the right two examples of incorrect moves. The upper example is incorrect because the ticked fields are not contiguous. The bottom example is incorrect because the ticked fields are not adjacent to other marked fields.

Scoring and Game End

The last phase of each turn is to check if any player has finished any of the columns or colors, which will award them points. If so, the first player to finish one column announces the corresponding column out loud and circles the higher points right underneath the

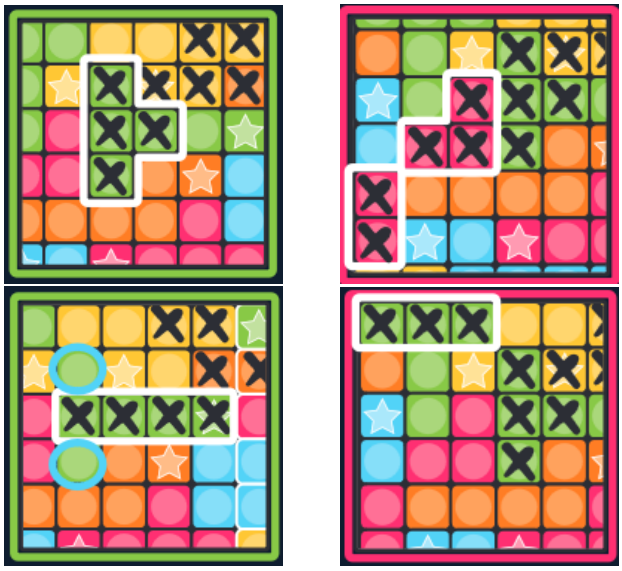


Figure 3.2: Examples of allowed and disallowed moves for crossing fields [BB16].

column (varies from 1 to 5, see Figure 3.1). All other players are required to cross it out, and from that point on, they can only score the lower number of points for this specific column (varies from 0 to 3). Again, if the game is played with more than two players, the lower amount of points for one column can then be earned by all remaining players, not just the second one, if they finish the column before the game ends. It is also possible to finish multiple columns in one turn. If several players are the first to fill the same column in the same turn, all these players receive the upper point value.

If the same happens with a whole color instead of columns, the first player to achieve this again gets the higher value of the color bonus field in the right upper corner of the game sheet (always 5), whereas the other players from now on can only get the lower point value (always 3) for this specific color.

The game ends immediately after the turn in which (at least) one player manages to circle their second color bonus field (the value does not matter), i.e., if a players manages to finish their second color. If this is the active player, all other players may form a combination of the remaining four dice for the last time and enter it.

The points of each individual player are then determined as follows:

- Add up all the circled points for all finished columns and colors.
- For every unused joker-symbol a player gets 1 extra point.
- For every not crossed star on the board a player loses 2 points.

The player with the highest score wins. In the event of a tie, the player who has more joker-symbols left wins. Otherwise there are several winners.

3.1.2 Noch mal so gut! – Game Instructions

For the extension of *"Noch mal so gut!"* the basic game remains the same. It is still the goal to get more points than the opponents by finishing colors and columns. But now there are a few other ways how one can earn points and also a few more options than taking just a color die and a number die. Again, all further details can be found in the game instructions [BB19b].

New Options

The main difference for the extension is the new special die. Instead of just rolling the six dice (three color and three number dice) from the basic game, we now roll a 7th die, which gives special options. Now every time someone picks dice in Phase 2 or 3, they can now take the special die if they have one unlocked, instead of taking a combination of color and number dice. If the active player chooses the special die in Phase 2, the others cannot pick it, but instead can choose from the remaining six instead of four dice. On the other hand, if the active player does not choose the special die, all other players have the option to choose it in Phase 3. Again, the exceptions of the original game mentioned above stay the same, i.e., in the first three turns everyone can pick the special die regardless of who else uses it.
The new options that the special die offers are:

- **Bomb:** The player detonates a bomb. To do this, they may mark up to four fields within a 2×2 grid (regardless of color) on the game block. These crosses do not have to be adjacent to existing crosses or in starting column *"H"*. This allows the player to mark fields away from column *"H"* at the start of the game. As the game progresses, further crosses may be placed adjacent to the fields that have been ticked by the bomb.

- **Heart:** With this option, the player circles a heart in their heart column, starting from the top. The hearts give the player an additional column bonus (from 1 to 5). The more hearts a player circles in the course of the game, the higher this bonus becomes. Whenever a player has completely ticked a column, they immediately receive their current heart bonus in addition to the point value they receive for the column. This is then entered directly in the heart below the corresponding column. If the player manages to circle additional hearts in the course of the game, they only increases the column bonus for columns completed at a later stage and not retroactively. Important to say is that this option is the only option which appears twice on the 6-sided special die.

- **Stars:** The player crosses up to two stars on their board sheet. However, these must be placed vertically or horizontally adjacent to at least one box that has already been crossed or in the starting column *"H"*.

- **Three in a row:** The player immediately crosses off up to three boxes in one row. These do not have to be adjacent to each other, but adjacent to boxes that have already been crossed.

- **Color block:** The player immediately crosses a complete color block. This block can be completely empty, but the player may also complete a color block that has already been started. The color block must be adjacent to at least one box that has already been crossed. This means that also a complete block of six can be crossed at once. This is the only way to achieve this in one turn.

New Board

Another difference of the extension is the game sheet, which you can see in Figure 3.3. Not only does it look different because the color blocks are sorted differently, there are also lots of new things to earn. In the extension there are only six normal jokers available (instead of eight in the basic game), but all players start with one special die symbol, which represents the available use of the special die, and can earn up to nine in the game. Some of the die symbols are distributed on the map and can be collected by ticking the corresponding field.
Other changes on the board include the hearts, which were described earlier, and the rewards for completing rows. In *"Noch mal so gut!"*, a key difference is that players can earn points and rewards by completing a full row. As before, finishing rows faster than the opponent is beneficial, as the first player to complete a specific row receives an extra bonus along with 5 points, while all other players only receive the 5 points. The extra bonuses can include additional die symbols, extra hearts, or the ability to immediately place a bomb on the board without using one of their special die symbols. If two players complete the same row simultaneously, both receive the extra bonus. Additionally, if a player completes both a row and a column in the same turn, the row bonus is awarded first.
A slight difference from the original board is that the first player to complete column *"H"* now earns 2 points. Additionally, regardless of order, every player who finishes *"H"* receives a special die symbol, represented by the die symbol behind the reward numbers on the game board.

New Scores

The points of each individual player are now determined like this:

- Add up all the circled points for all finished **rows**, columns, and colors.

- **Sum all bonus points earned from hearts by completing columns.**

- **For every unused but unlocked special die symbol, a player receives 2 extra points.**

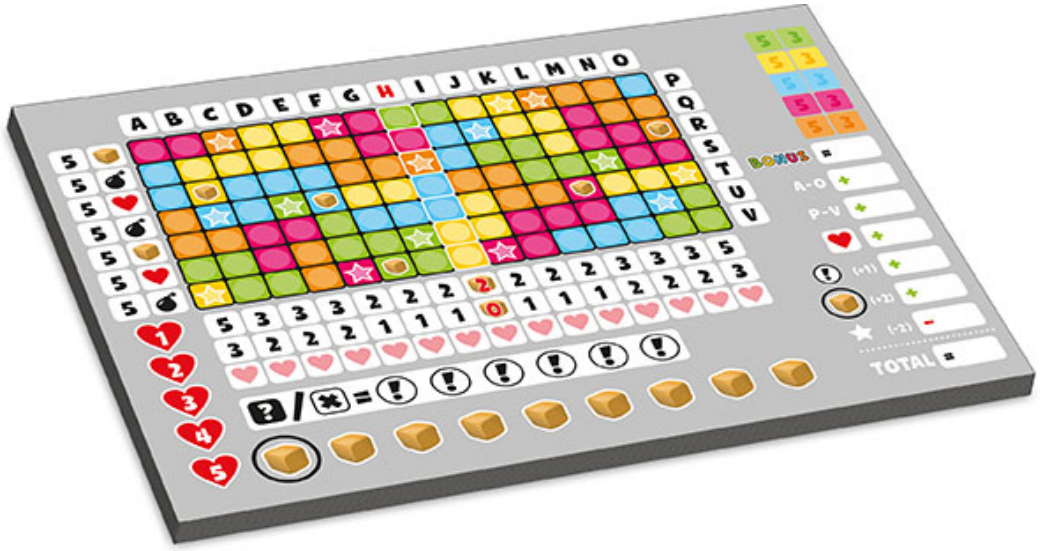


Figure 3.3: The game sheet for each player for the game *"Noch mal so gut!"* [BB19c].

- For every unused joker-symbol a player gets get 1 extra point.
- For every not crossed star on the board a player loses 2 points.

The player with the highest score wins. Again, in the event of a tie the player who has more joker-symbols left wins. Otherwise there are several winners.

3.2 Environment Architecture

This section will explain how we implemented the games *"Noch mal!"* and *"Noch mal so gut!"*, so that we could later train a RLA based on it. The game environments were both implemented in Python and both games can also be played by two human players in a terminal.
We again always start with the original game *"Noch mal!"* and then explain the differences to *"Noch mal so gut!"*.

3.2.1 Game

This class holds the methods for the majority of the game logic such as rolling the dice, ticking the fields on the players' boards, score calculations when stars, whole columns, or colors are ticked as well as calculating the legal actions for the current player. It also holds all the important information that we later need for the DQN like the number of features, the action size, the actual observation state of the current situation, or the possibility to make a random move to explore the action space. Since we are only looking at a two-player game in this thesis, all player-related information like the board, scored

points, jokers taken, etc., are saved two times, which can be adapted, but then also the input size for the DQN increases drastically for each additional player.
The game environment is largely the same for both versions. Naturally, the additional options are included in the class of *"Noch mal so gut!"*, but the core structure remains unchanged. The main difference is the need to track additional elements, such as the number of special die symbols available, hearts collected, and completed rows for the expansion.

3.2.2 Observation State

The state, as discussed in Section 2.1.2, is encoded as a list, allowing the RLA to utilize and interpret it effectively. The features included in the current state are:

- The game boards of both players, which contain $7 \cdot 15 = 105$ fields, whereby these are one-hot encoded, i.e., 0 means not crossed and 1 means crossed.
- The current score of each player, which are stored in two integers.
- The number of jokers that have already been used by each player, which are stored in two integers.
- The completed columns and colors for each player, which are stored in 40 integers, where again 0 means not finished and 1 means finished.
- The dice that are currently rolled, which we store in 12 integers, so that each digit is associated with one side of the dice. This means that the first digit stands for how often the color "red" was rolled, the second for "yellow" and so on.

Overall, the number of features is

$$\underbrace{210}_{\text{boards}} + \underbrace{2}_{\text{scores}} + \underbrace{2}_{\text{jokers}} + \underbrace{40}_{\text{columns and colors}} + \underbrace{12}_{\text{dice}} = 266.$$

For the game *"Noch mal so gut!"* we also add:

- The number of special die symbols that are available for each player, which are stored in two integers.
- The completed rows for each player, which are stored in 14 integers, again one-hot encoded.
- The new special die, which is also coded per digit, so that each option is represented by one digit, which add up five integers.
- The number of hearts collected, which are stored in two integers.

So overall we get

$$\underbrace{210}_{\text{boards}} + \underbrace{2}_{\text{scores}} + \underbrace{2}_{\text{jokers}} + \underbrace{2}_{\text{special die symbols}} + \underbrace{54}_{\text{columns, colors and rows}} + \underbrace{12}_{\text{dice}} + \underbrace{5}_{\text{special die}} + \underbrace{2}_{\text{hearts collected}} = 288.$$

3.2.3 Action Space

For the game *"Noch mal!"* we first write a script that saves all the possible actions in a JSON file. Since generalizing actions like "right", "left", "up", and "down" as in other games for example is not straightforward, we have to figure out all possible combinations of dice rolls and the fields that can be marked over the entire game. Therefore, we have to go through all the combinations of dice that can be chosen and all the possible fields that can be ticked with it. Each move is assigned an "ID", a "color", a "num_fields" (the number of fields that can be marked), and a "fields" array that lists the specific fields involved in the move as can be seen in Figure 3.4.
For example, with the combination "red" and "1" each of the 21 fields that are red, can be ticked. For the combination "red" and "2" only moves with two neighboring red fields are possible, which are 15.

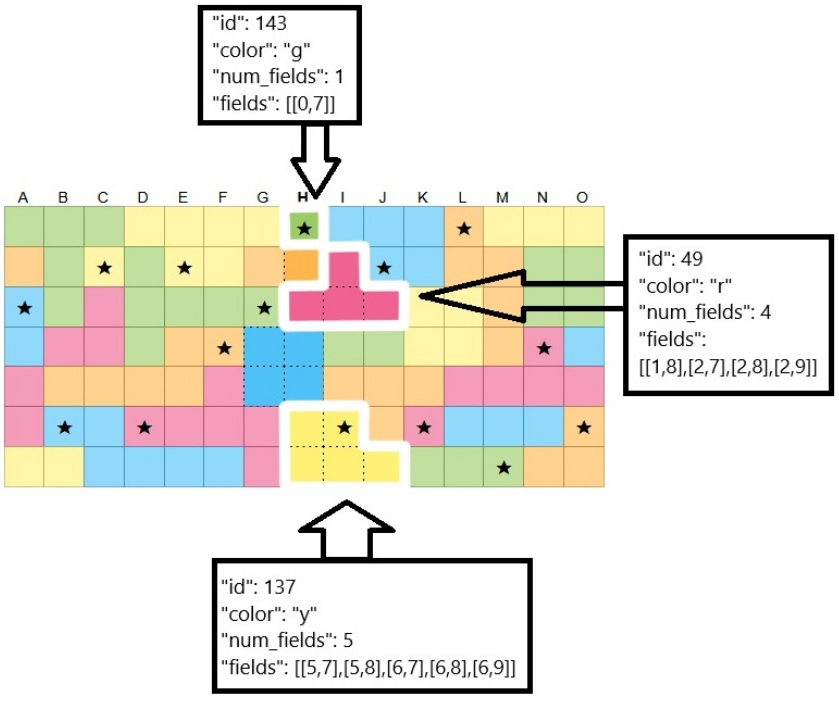


Figure 3.4: Visual examples illustrating the attributes "ID", "color", "num_fields", and "fields". This figure is a reconstructed representation of the game board [BB16].

We then also add the 10 combinations with jokers, like "**X**" and "1","**X**" and "2" etc., as well as "red" and "**?**", "yellow" and "**?**", etc. For these joker-options we choose not to save any fields, so that these options just open up the other already stated options and another input is needed to complete the turn (see Section 3.3.2). At least also the "pass" option has to be added. Because not all colors have the same number of options we get a total number of actions of

$$\underbrace{62}_{\text{red}} + \underbrace{76}_{\text{yellow}} + \underbrace{63}_{\text{green}} + \underbrace{64}_{\text{blue}} + \underbrace{56}_{\text{orange}} + \underbrace{11}_{\text{joker-options}} + \underbrace{1}_{\text{pass}} = 333$$

For the RLA, which will be explained later in Section 3.3, all of these action have to be individual output nodes. It must be said that this is the number of all possible actions over the entire game, but usually only about 20 of them are legal and therefore available in one turn, because of the given rules for ticking described in Section 3.1.1.

For the game *"Noch mal so gut!"*, one could believe that with the new special options there have to be more options than in the original game, but because the game sheet looks different there are actually less. For this game, we also count all the options for every dice combination and save them in a new JSON file. We add the joker-option and "pass" as before and also put in the five new special options which come from the new special die. For the option "bomb" and "three-in-a-row", we decided to use a different network afterwards so we do not add these options to the original file. But for the "color block" option, where a complete block of six can now be crossed off at once, we decided to put these five options also in the JSON file, because one always has these options in combination with the other options in this file. We end up with an action size of

$$\underbrace{55}_{\text{red}} + \underbrace{55}_{\text{yellow}} + \underbrace{64}_{\text{green}} + \underbrace{55}_{\text{blue}} + \underbrace{55}_{\text{orange}} + \underbrace{11}_{\text{joker-options}} + \underbrace{5}_{\text{special options}} + \underbrace{1}_{\text{pass}} + \underbrace{5}_{\text{6-block-option}} = 306.$$

For the two stated special options the amount of further options is pretty simple to compute. The "three-in-a-row" option brings up the choice of every row, which are seven, so the action size for this extra network is 7. After deciding the row, we can come back to the original network to choose some of the options restricted on the selected row.
For the "bomb" action we know that every possible 4×4 block on the field is an option. Since the board game is 7×15 fields large we have an action size of $6 \cdot 14 = 84$.

3.3 Agent Architecture

For the training phase we implement a self-playing program in which the RLA plays against itself and receives rewards based on the reward function. In this section, we will describe how we build the network and how we reward the agent in order to create a competitive player. As seen in Section 3.1, both of these games have no hidden information, allowing for a theoretically optimal strategy, similar to *Go* and *Chess*. However, unlike those games, *"Noch mal!"* and *"Noch mal so gut!"* incorporate elements of randomness, because of the dice rolling, which makes it even more difficult to produce a well playing agent.

3.3.1 Networks

The implementation of the RLA was inspired by the GitHub repository of the user "chiamp", which is publicly available at [chi20] and served as a foundation for key aspects of the architecture. While the original implementation served as a useful starting point, substantial modifications were necessary to adapt the architecture to the specific requirements and goals of this project and especially for our given games. The RLA is built using Keras [Ker15], which is a Python binding to the ML framework TensorFlow [Ten15]. The structure of the network is based on many hyper-parameters, for which the tuning will be discussed in detail in Section 3.3.3, but to give an overview we will briefly describe the structure of the network in this section:

- First, the neural network has an input layer with a number of neurons equal to the number of features, which was computed in Section 3.2.2. It uses the *ReLU* activation function, which was chosen due to its computational efficiency and its ability to soften the vanishing gradient problem in deep networks.
- Then we have a few hidden layers whose number is based on the hyper-parameter "hidden_layers", with 300 neurons based on [Kri21], which states: *"The number of hidden neurons should be between the size of the input layer and the size of the output layer"*. Again, the activation function is *ReLU*.
- At the end, we have the output layer which has the number of neurons based on the action size, which was calculated in Section 3.2.3. It uses the activation function *linear*, which is the default choice and was found to be sufficient for the purposes of this work.

The network structure varies depending on the game, as both state and action sizes differ between *"Noch mal!"* and *"Noch mal so gut!"*. All of these layers are dense layers, which means that all neurons are fully connected to all neurons of the next layer. Overall the network uses the *Mean Squared Error* as loss function and the *Adam* optimizer, which are both very common in ML tasks and also used by "chiamp" [chi20].

As mentioned earlier for the game *"Noch mal so gut!"*, we also build two further networks for the "bomb" and the "three-in-a-row" special-options. The idea behind that is to have hierarchical networks, which is also mentioned in [Pla20], where the first network is the high level network and the two others are only called if the first one decides to.
Both these networks have the same structure and functions used as the main one, but instead have the hyper-parameter "bomb_hidden_layers" or "row_hidden_layers", which give the number of hidden layers. The number of neurons was also reduced to 150 and 50, respectively, to match the considerably smaller output layer, as previously calculated in Section 3.2.3.

For these games we also argue that we do not need a target network, because in simpler, fully observable environments like these board games, there is less risk of instability

during learning, so a target network (used to prevent instability in complex environments) is not required. Moreover, these games have a clearly defined endpoint after a set number of moves, which reduces the accumulation of errors that can destabilize the learning process. Additionally, removing the target network reduces memory consumption since only one neural network needs to be stored, avoiding the duplication of weights and computations. This is quite similar to the approach of "chiamp" [chi20].

3.3.2 Training Loop

Algorithm 3.1 shows a simplified version of the main training loop that the RLA has to go through. The code will be discussed in detail in the following sections. As we can see in the overall context, the agent always plays 10 games against itself and then trains on a sample batch of actions that were taken in the last games. Every 500 games, the agent competes against the current best model, evaluating its win rate to track performance improvements. For the first opponent, we use a *RandomAgent* that always selects a random action from the permitted actions. An important fact to notice is that these agents only learn through self-play, which means no additional information about the rules, the points deduction for taking jokers or additional points for crossing fields with stars are given. The only guidance provided is that the agent cannot take illegal actions, as described in the next section.

Taking Action

Every turn the agent has to take one of the current legal actions. As mentioned before, in most of the cases around 10 to 20 options are actually allowed out of the huge action space. To simplify action learning and reduce computational complexity, we determine the set of allowed actions at each turn and mask all others. That means the DQN always computes the Q-values of all actions based on the given current state and set all illegal moves to $-\infty$. Algorithm 3.2 also shows the Epsilon-Greedy-strategy, which is commonly used to strike the trade-off between the "exploration" and "exploitation" phases, which we described in Section 2.1.3. The idea is that the RLA continuously explores the action space by a given amount of percentage, which decreases from game to game. If ϵ is greater than a random number between 0 and 1, then a random option out of the legal options is taken, if it is less, then the agent will choose an action based on the current network. A minimum value of $\epsilon = 0.1$ ensures that the agent continues to explore the action space at least slightly even in later stages of training. This prevents the agent from falling into a local optimum by always exploiting the current knowledge. It guarantees ongoing exploration, helping the agent refine its strategies and avoid premature convergence.

For joker- and special-actions, the agent must take an additional step, as these actions introduce additional decision-making levels. For example, if the agent chooses the option "red" and "**?**", we need to determine the second part of the action. At this point, the network is re-entered with an updated state, considering the already chosen die. The set of legal actions is then recalculated, restricting the choices to valid moves (based on Section 3.2) using "red" with any number from 1 to 5. A similar approach applies to

Algorithm 3.1: Main Training Loop

Require: Game environment G, number of episodes N

Initialize Q-network
Initialize replay memory $\mathcal{M} \leftarrow \emptyset$
for episode $e = 1$ to N **do**
  Reset game: $G.reset()$
  while game not done **do**
    Roll dice: $G.roll\_dice()$
    for player in {0, 1} **do**
      Get current state: $s_t \leftarrow G.get\_features(player)$
      Choose action a_t
      Execute action: $G.act(a_t)$
      Observe reward r_t, next state s_{t+1}
      Store (s_t, a_t, r_t, s_{t+1}) in $\mathcal{M}$
    end for
    if Game is over **then**
      End the game
    end if
  end while
  if $e \mod 10 == 0$ and memory $\mathcal{M}$ has enough samples **then**
    Sample batch from $\mathcal{M}$
    Train network on batch
  end if
  if $e \mod 500 == 0$ **then**
    Test performance against best version and save model
  end if
end for

Algorithm 3.2: Act

Get current state: $s_t \leftarrow G.get\_features(player)$
Get legal actions: $A_{legal} \leftarrow G.get\_legal\_actions()$
Choose action a_t using Epsilon-Greedy-strategy: ϵ = max(0.1, 1 - episode / 10000)

$$a_t \leftarrow \begin{cases} \text{random action from } A_{legal}, & \text{w.p. } \epsilon \\ \arg\max_{a'_t \in A_{legal}} Q(s_t, a'_t), & \text{w.p. } 1 - \epsilon \end{cases}$$

Execute action: $G.act(a_t)$

special actions, where the second choice is constrained based on the first.
In case of the "bomb" and the "three-in-a-row" action, we build the agent so that we enter the specific network with the current state and also use the Epsilon-Greedy-strategy to explore the action space of these networks.

Remembering

The remembering of the RLA is simple. Since the agent plays against itself, it saves all the actions made from the perspective of the current player. In this case, we choose to save the last 10,000 actions, as shown in detail in Algorithm 3.3. It always saves the action, the current state, the reward, and the next state. Because the dice are given in the current state, see Section 3.2.2, we decide to define the next state as the state of the next turn of the player. Additionally, we save the legal actions provided by the next situation to improve the accuracy of the training.

Algorithm 3.3: Remember

1: Store (s_t, a_t, r_t, s_{t+1}) in $\mathcal{M}$
2: $\mathcal{M} = \mathcal{M}[-10000:]$

For the low-level networks in the game of *"Noch mal so gut!"* we also save the action, the current state, the reward, but not the next state as in the main network. This is because the next state of these low-level networks cannot be defined as well. Simply treating the next turn as the new state would be inaccurate, because for example where the next "bomb" should be placed is not relevant, because very likely the agent does not choose this option in the next turn and we would not enter the low-level network. Another difference is that each action is used only once for training in the low-level networks. After each training phase, we clear the "bomb" and the "three-in-a-row" memories due to the limited amount of input data (only 10 games played). Retaining this small dataset would lead to the repeated use of the same data for training. By clearing the memory, we help prevent overfitting, ensuring the agent does not memorize infrequently used actions and instead focuses on generalizing better for future decisions.

Reward

The reward is an important factor in the training loop because it defines what good or bad options are. During the development of our approach, we experimented with various reward functions to assess their impact on the model's performance. The reward functions we ended up with, simply rewards winning and punishes losing, by setting the reward for all actions taken by one player in the game as follows:

- +1 for winning
- 0 for a draw

- -1 for losing

Additionally, we introduced two exceptions:

- -1 for passing
- -1 if the cluster marked in this turn could have been filled in that turn.

Both of these exceptions were taken to reduce the training time by anticipating bad moves manually. By imposing this penalty, we encourage the agent to make proactive decisions, rather than avoiding actions altogether. It also helps guide the agent to prioritize actions that have a more significant impact on the game's outcome. While there are rare situations where passing could be beneficial, we chose to treat it as generally undesirable.

After experimentation and analysis, we decided to use this reward function because it performed well and provided a more general understanding of the game. Due to its simplicity and generality, it can be reused in similar games, as it is not too specific to the game in particular. By keeping the reward structure minimal and avoiding complex, game-specific rules, the agent can learn more effectively without unnecessary bias. This flexibility also allows the reward function to be easily adapted to other environments with minimal adjustments as we will see in Chapter 4. A similar structure (without the exceptions) is also used in the paper [PMVI24] for the game *"7 Wonders Duel"*.

For joker and special actions that require additional action selections, we only reward the initial action. This is because the second action is simply a consequence of the agent's already learned strategy, based on the restrictions imposed by the first action. Rewarding the initial action reflects the agent's ability to make a good decision, considering the available options at the time, without needing to further evaluate the outcome of the second action. For the game *"Noch mal so gut!"*, we apply the same reward function to the two low-level networks as described for the main network.

Train on Batch

After every 10 games, we train the network using the memory buffer. As shown in Algorithm 3.4, we sample a small batch from the memory and update the weights of the network using the Bellman equation (2.4) from Section 2.2. We apply a similar approach for the low-level networks in *"Noch mal so gut!"*, as previously mentioned. However, instead of using the next state, we update the network based solely on the reward of the current move, treating every turn in the low-level networks as a terminal state.

Testing

For the last part, we implement an evaluation of the trained models. To this end, every 500 games the current model plays 200 games against the current best model and see if

Algorithm 3.4: Train on Batch

Require: Memory $\mathcal{M}$, batch_size, discount_factor γ

if e mod 10 == 0 and len($\mathcal{M}$) > 1000 **then**
  batch ←
  Get batch_size random samples from memory; sample_obj = (s_t, a_t, r_t, s_{t+1})
  create Q-values-table
  for each sample_obj in batch **do**
    if the game ended after this move (no next state available) **then**
      Update Q-values-table with r_t
    else
      Mask new state s_{t+1} with legal actions
      Update Q-values-table with $r_t + (\gamma \cdot \max_{a_{t+1} \in A_{legal}} Q(s_{t+1}, a_{t+1}))$
    end if
  end for
  Train network with Q-values-table
end if

it has improved over the last games. If so, we save the new model and use it as the new best model. It is clear that winning in a game is not transitive. This means that if A wins against B and B wins against C, A does not necessarily have to win against C, but because we have that many agents to test, we decide to use this as a presorting method. Algorithm 3.5 displays this process. We also play tournaments afterwards between the different best agents to see which one prevails against the other trained opponents, see Section 3.3.3. In the end, we also test some models against human players, as described in Section 3.4.

Algorithm 3.5: Test

Require: Current Network/Model Q_n, Best Network/Model Q

if e mod 500 == 0 **then**
  Test Q_n against Q in 200 games
  if wins > losses **then**
    Save Q_n
    $Q \leftarrow Q_n$
  end if
end if

3.3.3 Hyper-parameter tuning

For the entire RLA, there are many parameters to consider. We have already addressed some of these in the previous section and left some of them open for further testing. To provide a complete overview of all relevant hyper-parameters, we list them here:

- **hidden_layers**: The number of hidden layers in the network.
- **Number of neurons**: The number of neurons in each hidden layer. For the main network, we set this to 300, as we discussed earlier.
- **Epsilon (ϵ)**: The value that determines whether to explore or exploit the action space. We set this value to $\max(0.1,\ episode/10000)$ as mentioned earlier.
- **learning_rate**: The learning rate for the *Adam* optimizer.
- **Number of episodes**: The total number of games the agent plays during training.
- **discount_factor (γ)**: The value of γ in the Bellman equation (2.4).
- **memory_size**: The number of action-state-reward-next-state pairs stored in the memory buffer. We set this to 10,000 to learn from a broader range of past experiences.
- **batch_size**: The number of samples used in each training batch. We set this to 1,000 to provide enough variety to allow effective training without overwhelming the available resources.
- **training interval and testing interval**: The number of episodes after which we train/test the network. We set this to 10 and 500, respectively, to update the network frequently and provide a meaningful evaluation of the agents' performance.

Additionally, for the game *"Noch mal so gut!"*, there are a few more hyper-parameters specific to the low-level networks:

- **bomb_hidden_layers**: The number of hidden layers in the "bomb" network.
- **Number of neurons of the "bomb" network**: The number of neurons in each hidden layer of the bomb network. We set this to 150.
- **bomb_learning_rate**: The learning rate for the *Adam* optimizer in the bomb network.
- **row_hidden_layers**: The number of hidden layers in the "three-in-a-row" network.
- **Number of neurons of the "three-in-a-row" network**: The number of neurons in each hidden layer of the row network. We set this to 50.
- **row_learning_rate**: The learning rate for the *Adam* optimizer in the "three-in-a-row" network.

Since there are numerous hyper-parameters to consider, we choose to fix some and focus on an in-depth analysis of four key ones: the number of hidden layers, learning rate, number of episodes, and discount factor. These parameters were selected for their expected significant impact on model performance. To explore their impact, we conduct a grid search with specific settings. Table 3.1 presents the initial combinations we test.

Hyper-parameter	**Values Tested**
hidden_layers	[5, 8, 10]
learning_rate	[0.0001, 0.001]
discount_factor	[0.5, 0.6, 0.7, 0.8, 0.9, 0.95]

Table 3.1: The first set of tested hyper-parameters for the game *"Noch mal!"*.

The 36 combinations of these settings were initially tested with 50,000 episodes, so that every configuration runs through the training loop from Algorithm 3.1 of the game *"Noch mal!"*. Afterwards, the best model from each configuration is selected to compete against one another in a tournament, where each model plays 200 games against all other models. The tournament results are summarized in Table A.1 in the Appendix, and the top six models are displayed in Table 3.2.

Model	**hidden layers**	**learning rate**	**discount factor**	**Wins**
3	5	0.0001	0.8	**3823**
12	8	0.0001	0.5	**3907**
24	10	0.0001	0.5	**3817**
27	10	0.0001	0.8	**3854**
30	10	0.001	0.5	**3885**
31	10	0.001	0.6	**3938**

Table 3.2: The best six models after 50,000 episodes of Table A.1.

From this tournament, we observe that models with 10 hidden layers performed the best, and lower learning rates were more effective. We then conduct another round of training with 100,000 episodes, as detailed in Table 3.3, to further assess the models' performances. We retain both learning rates for this second round, hypothesizing that the networks with a lower learning rate would improve over time with more episodes.

Hyper-parameter	**Values Tested**
hidden_layers	[10]
learning_rate	[0.0001, 0.001]
discount_factor	[0.5, 0.6, 0.7, 0.8, 0.9]

Table 3.3: The second set of tested hyper-parameters for the game *"Noch mal!"*.

After this second training round, we conducted another tournament with these 10 networks, which are numbered in Table 3.4, and the detailed results are shown in Table 3.5.

Model	hidden layers	learning rate	discount factor
0	10	0.0001	0.5
1	10	0.0001	0.6
2	10	0.0001	0.7
3	10	0.0001	0.8
4	10	0.0001	0.9
5	10	0.001	0.5
6	10	0.001	0.6
7	10	0.001	0.7
8	10	0.001	0.8
9	10	0.001	0.9

Table 3.4: Numbering of the models after the second training round of 100,000 episodes of *"Noch mal!"*.

Model	0	1	2	3	4	5	6	7	8	9	Wins
0	X	93	101	120	127	116	97	107	125	121	**1007**
1	107	X	112	111	114	100	112	127	121	115	**1019**
2	98	87	X	118	117	97	96	110	109	108	**940**
3	73	86	82	X	106	96	85	101	107	98	**834**
4	73	86	80	93	X	97	100	105	103	100	**837**
5	84	97	101	102	100	X	90	121	105	104	**904**
6	103	86	103	114	100	109	X	96	106	110	**927**
7	91	69	90	99	92	77	104	X	88	97	**807**
8	73	76	90	92	95	93	94	111	X	104	**828**
9	78	82	92	100	100	94	88	103	95	X	**832**
Losses	780	762	851	949	951	879	866	981	959	957	

Table 3.5: The detailed tournament results of the 10 models listed in Table 3.4. Victories of the respective model are recorded in the corresponding row, while losses are shown in the respective column. Ties are not included in the table.

The table shows that the model 1 emerges as the overall winner, with 1,019 total wins. This model outperformed all others, winning at least 100 games in each duel. Notably, models with a learning rate of 0.0001 secured the top 3 positions. Based on these results, we decided to use model 1 for further analysis.
That means we finish the training phase of *"Noch mal!"* with the hyper-parameters you see in Table 3.6 and put them to the test against several human players, which is described in Section 3.4.

Hyper-Parameter	Value
Hidden layers	10
Number of neurons	300
Learning rate	0.0001
Discount factor (γ)	0.60
Epsilon (ϵ)	$\max(0.1, \text{episode}/10000)$
Number of episodes tested	100,000
Memory size	10,000
Batch size	1,000
Training interval	10
Testing interval	500

Table 3.6: The hyper-parameters of the final network of *"Noch mal!"*

For the game *"Noch mal so gut!"*, we leverage the insights gained from the hyper-parameter testing of *"Noch mal!"*, as both networks share a similar structure. Therefore, we adopt the hyper-parameters from Table 3.6 and incorporate the additional hyper-parameters for the low-level networks from Table 3.7. We also decide to test the three best discount factors from the previous tournament $(0.5, 0.6, 0.7)$ because the difference in performance was minimal. Given the computational cost of training three networks simultaneously with different configurations, we limit the first training round to testing only two numbers for the hidden layers and two numbers for the learning rates in the low-level networks. This results in 48 different models, which we train for 150,000 episodes/games, as we believe this game is more complex and requires more training, particularly for the low-level networks.

Hyper-parameter	Values Tested
hidden_layers	[10]
learning_rate	[0.0001]
discount_factor	[0.5, 0.6, 0.7]
bomb_hidden_layers	[3, 5]
bomb_learning_rate	[0.0001, 0.001]
row_hidden_layers	[3, 5]
row_learning_rate	[0.0001, 0.001]

Table 3.7: The set of tested hyper-parameters for the game *"Noch mal so gut!"*.

After training, we again conduct a tournament with all 48 models, where each model plays 200 games against every other model. The results of this tournament are shown in Table A.2 in the Appendix, and the best six models can be found in Table 3.8.

As we cannot identify any clear patterns in the top 6 models, we decide to extend the training to 200,000 episodes to further refine the models. The detailed results of the final tournament are presented in Table 3.9.

Model	hl	lr	df	bomb_hl	bomb_lr	row_hl	row_lr	Wins
6	10	0.0001	0.5	3	0.001	5	0.0001	**5015**
9	10	0.0001	0.5	5	0.0001	3	0.001	**5141**
25	10	0.0001	0.6	5	0.0001	3	0.001	**5227**
30	10	0.0001	0.6	5	0.001	5	0.0001	**4982**
38	10	0.0001	0.7	3	0.001	5	0.0001	**5054**
45	10	0.0001	0.7	5	0.001	3	0.001	**5289**

Table 3.8: The best six models after 150,000 episodes of Table A.2.

Model	hl	lr	df	bomb_hl	bomb_lr	row_hl	row_lr	Wins
0	10	0.0001	0.5	3	0.0001	3	0.0001	4281
1	10	0.0001	0.5	3	0.0001	3	0.001	4876
2	10	0.0001	0.5	3	0.0001	5	0.0001	4243
3	10	0.0001	0.5	3	0.0001	5	0.001	4178
4	10	0.0001	0.5	3	0.001	3	0.0001	3971
5	**10**	**0.0001**	**0.5**	**3**	**0.001**	**3**	**0.001**	**5099**
6	**10**	**0.0001**	**0.5**	**3**	**0.001**	**5**	**0.0001**	**5113**
7	10	0.0001	0.5	3	0.001	5	0.001	4888
8	10	0.0001	0.5	5	0.0001	3	0.0001	4715
9	**10**	**0.0001**	**0.5**	**5**	**0.0001**	**3**	**0.001**	**5011**
10	10	0.0001	0.5	5	0.0001	5	0.0001	4665
11	10	0.0001	0.5	5	0.0001	5	0.001	4797
12	10	0.0001	0.5	5	0.001	3	0.0001	4386
13	10	0.0001	0.5	5	0.001	3	0.001	4323
14	10	0.0001	0.5	5	0.001	5	0.0001	4623
15	10	0.0001	0.5	5	0.001	5	0.001	4999
16	10	0.0001	0.6	3	0.0001	3	0.0001	4496
17	10	0.0001	0.6	3	0.0001	3	0.001	4629
18	10	0.0001	0.6	3	0.0001	5	0.0001	4732
19	10	0.0001	0.6	3	0.0001	5	0.001	4493
20	**10**	**0.0001**	**0.6**	**3**	**0.001**	**3**	**0.0001**	**5097**
21	10	0.0001	0.6	3	0.001	3	0.001	4688
22	10	0.0001	0.6	3	0.001	5	0.0001	4589
23	10	0.0001	0.6	3	0.001	5	0.001	4509
24	10	0.0001	0.6	5	0.0001	3	0.0001	4928
25	10	0.0001	0.6	5	0.0001	3	0.001	4997
26	10	0.0001	0.6	5	0.0001	5	0.0001	4492
27	10	0.0001	0.6	5	0.0001	5	0.001	4491
28	10	0.0001	0.6	5	0.001	3	0.0001	4651
29	10	0.0001	0.6	5	0.001	3	0.001	4478

30	10	0.0001	0.6	5	0.001	5	0.0001	4562
31	10	0.0001	0.6	5	0.001	5	0.001	4501
32	**10**	**0.0001**	**0.7**	**3**	**0.0001**	**3**	**0.0001**	**5167**
33	10	0.0001	0.7	3	0.0001	3	0.001	4807
34	10	0.0001	0.7	3	0.0001	5	0.0001	4956
35	10	0.0001	0.7	3	0.0001	5	0.001	4446
36	10	0.0001	0.7	3	0.001	3	0.0001	4669
37	10	0.0001	0.7	3	0.001	3	0.001	4168
38	10	0.0001	0.7	3	0.001	5	0.0001	4892
39	10	0.0001	0.7	3	0.001	5	0.001	4731
40	10	0.0001	0.7	5	0.0001	3	0.0001	4788
41	10	0.0001	0.7	5	0.0001	3	0.001	4403
42	10	0.0001	0.7	5	0.0001	5	0.0001	4513
43	10	0.0001	0.7	5	0.0001	5	0.001	4734
44	**10**	**0.0001**	**0.7**	**5**	**0.001**	**3**	**0.0001**	**5108**
45	10	0.0001	0.7	5	0.001	3	0.001	4753
46	10	0.0001	0.7	5	0.001	5	0.0001	4909
47	10	0.0001	0.7	5	0.001	5	0.001	4591

Table 3.9: The results of the first training round of the game *"Noch mal so gut!"*. In the table, the best 6 models are highlighted in bold.

Model 32 emerges as the winner of the final tournament, with 5,167 wins out of 9,400 games played. While this model did not dominate all the others, it remained very competitive, winning almost half of the games in every match-up. We conclude the training phase with model 32 and proceed to evaluate its performance against human players, as described in Section 3.4. The specific hyper-parameters for the final agent are summarized in Table 3.10.

3.4 Evaluation

The purpose of this section is to show that the two agents from the previous section can prevail against human players and can be considered for analysis, especially with regard to game development. We will also analyze the different game statistics and play styles of our RLA and the human contestants. The first goal to be achieved is for the agents to play better than a player taking actions at random, which is achieved in every configuration after 500 training games, as the given action space is very large. In the end, the final agents won 100% of the games against the random agent.
The second goal was to get a minimum win rate of 50% against human players to confidently state that our agent is on par with human players, providing a solid foundation for further analysis of the game. To clarify further, the goal was not to create the best player for the given games, but to develop a competent player capable of analyzing the

Hyper-Parameter	**Value**
Hidden layers	10
Number of neurons	300
Learning rate	0.0001
Discount factor (γ)	0.70
Bomb hidden layers	3
Bomb number of neurons	150
Bomb learning rate	0.0001
Row hidden layers	3
Row number of neurons	50
Row learning rate	0.0001
Epsilon (ϵ)	$\max(0.1, \text{episode}/10000)$
Number of episodes tested	200,000
Memory size	10,000
Batch size	1,000
Training interval	10
Testing interval	500

Table 3.10: The hyper-parameters of the final network of *"Noch mal so gut!"*

game from a game developer's perspective in order to suggest improvements. Therefore, we set up a graphical environment on a website to play the game *"Noch mal!"* outside of the terminal, making it easier for the contestants to compete against the agent. In Figure 3.5 one can see a screenshot of the website during a game.

As one can see, all important information for the game is displayed, such as the board of both players, the chosen and currently available dice, the finished columns and rows, etc. We then grouped all the contestants into four categories based on their experience with the game *"Noch mal!"*, in order to have a clear overview of the agent's performance later on. The groups were divided as follows:

- Have played the game less than 3 times.
- Have played the game between 3 and 10 times.
- Have played the game between 11 and 25 times.
- Have played the game more than 25 times.

If the players have played against the AI very often, we have also upgraded them to the next higher level. In total, we had 18 contestant and the results of all the games in the specific categories can been seen in Table 3.11.

As one can see, the results are impressive, with the agent achieving a win rate of over 50% in every group. It must also be said that *"Noch mal!"* is a game that is also very

Figure 3.5: A screenshot of the graphical environment for testing the agent of the game *"Noch mal!"*. This figure is a reconstructed representation of the game [BB16].

Group (Games played beforehand)	< 3	$3 - 10$	$11 - 25$	> 25
Played Games	17	32	51	77
Won Games	9	24	26	40
Win rate	52.94%	75%	50.98%	51.95%

Table 3.11: Results of all *"Noch mal!"* games played against human players.

much based on luck, due to the rolling of dice, which is why it is remarkable to achieve a high win rate at all. This is also why the win rate does not decrease significantly with increased player experience.
In total, the RLA won 99 out of 177 games, resulting in a win rate of 55.93%. The agent averages 24.06 points and uses 6.32 jokers, whereas the players score 23.18 points and use 5.69 jokers in comparison. Figure 3.6 compares the color preferences of our RLA against the human players. And as we can see, the model in general finishes much more colors than the players. On average, the agent finished 1.66 colors per game and the players only 1.40. The diagram also shows a strong preference for the color green in both play styles, with the notable exception that the agent almost never finishes the color blue. For these graphics, one also has to say that the reason for such big differences in the numbers of the RLA completing something in comparison to the human players is that the human players are several people with different strategies, whereas the agent always has the same plan "in mind".

If we look at the columns, we notice that the agent on average finished 7.27 columns and

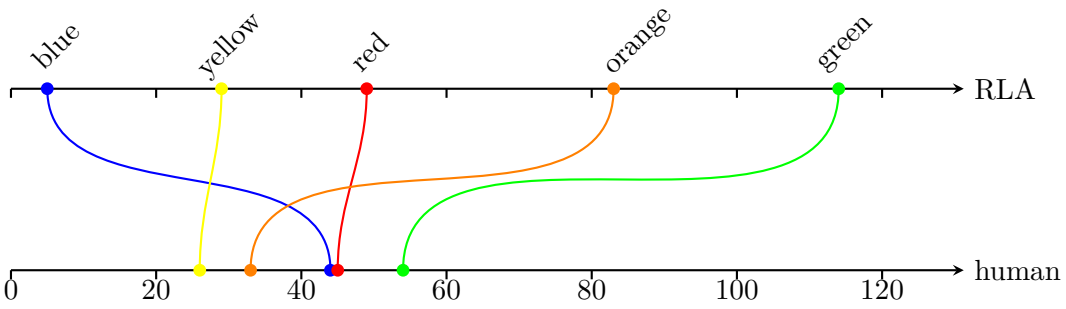


Figure 3.6: Color preferences of our RLA (top) and human players (bottom) across 177 games of *"Noch mal!"*.

the players 8.06 per game. The preferences in Figure 3.7 show that the columns *"I"* and *"J"* are very important for both play styles. Interestingly, the column *"H"*, which is the starting column, is not often finished by the agent, perhaps because it earns 0 points when the opponent already finished it. Also interesting to see is that the columns *"A"* and *"B"* are clearly less finished by human players than *"N"* and *"O"*, which could be due to the fact that all 6-field color blocks are on the left side of the game board and can therefore not be crossed in one turn.

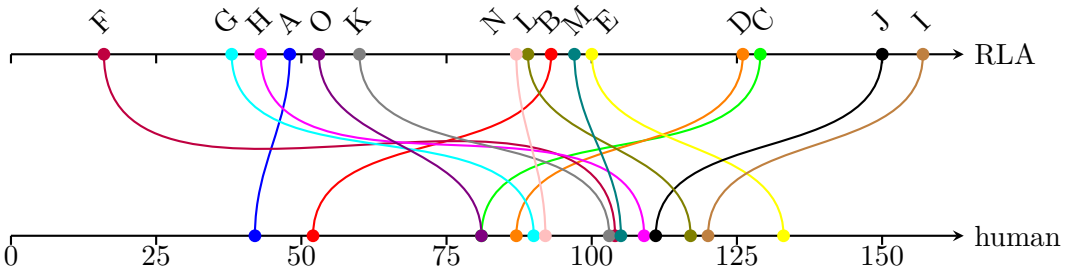


Figure 3.7: Column preferences of our RLA (top) and human players (bottom) across 177 games of *"Noch mal!"*.

Overall, our RLA proves to be a robust player, making it a valuable tool for game analysis – a topic we will further explore from a developer's perspective in Section 3.5.

For the game *"Noch mal so gut!"*, we also developed a graphical environment (Figure 3.8) that allows for easy interaction and visualization of the game. This environment features the new elements of the extension, including bonuses for rows, a special die above the board, hearts, and special die symbols at the bottom of the figure, which reflect the new rules and mechanics introduced.

We once again categorized our contestants into the four categories defined earlier and start playing against the RLA of *"Noch mal so gut!"*. As seen in Table 3.12, our agent won 146 out of the 278 games played, resulting in a win rate of 52.52%. It is clear that the group of participants who played more than 25 games, i.e., those who were more

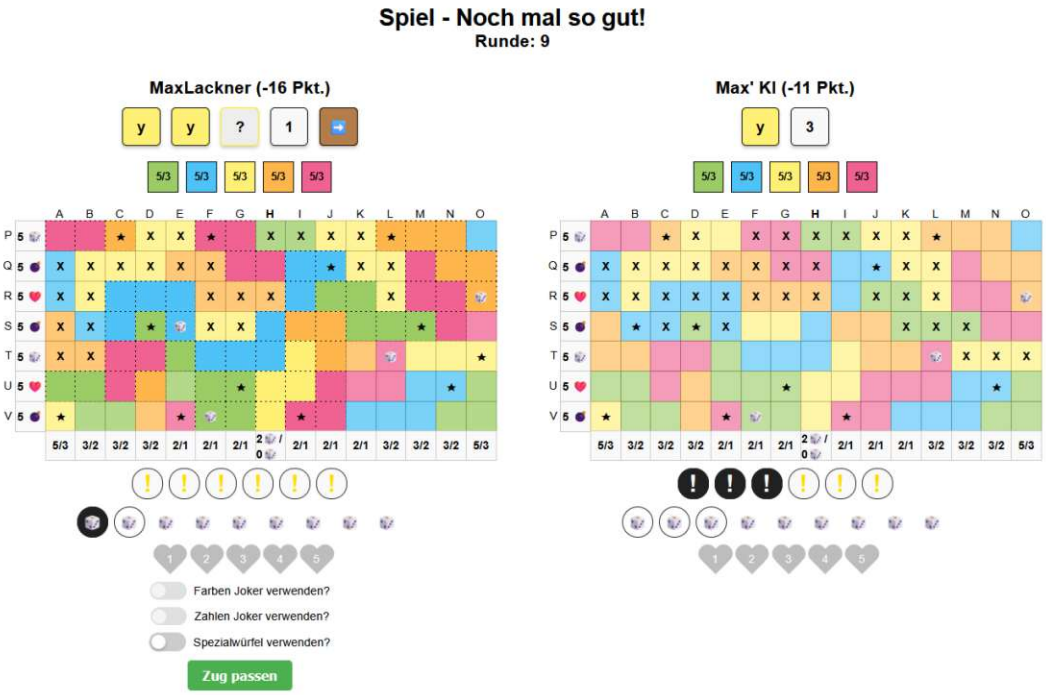


Figure 3.8: A screenshot of the graphical environment for testing the agent of the game *"Noch mal so gut!"*. This figure is a reconstructed representation of the game [BB19b].

familiar with the game, dominated the statistics, playing significantly more games than the other groups. Despite this, the agent still maintained a competitive win rate against this group, although it is below the expected 50%. This time we also see that the agent dominated the group of players which played the game less than 3 times, which shows that the game is more complicated and takes several games to understand.

Group (Games played beforehand)	< 3	$3 - 10$	$11 - 25$	> 25
Played Games	25	34	38	180
Won Games	18	20	20	88
Win rate	72%	58.82%	52.63%	48.62%

Table 3.12: Results of all *"Noch mal so gut!"* games against human players.

When comparing the RLA and human players (Table 3.13), several key differences stand out. For example, the agent uses more special die symbols on average, which is probably due to the fact that it has more specials available throughout the game. Additionally, while both the agent and human players found the "color block" action most useful, the agent more frequently used the "three-in-a-row" action, showing a preference for completing rows over other strategies.

Figures 3.9, 3.10, and 3.11 show the color, column, and row preferences. Notably, blue

Player	RLA	Human
Points	50.86	49.25
Jokers used	5.44	4.20
Special actions used	6.18	4.89
Special Die Symbols available at the end	1.29	2.05
Hearts collected	1.62	1.30
Colors finished	1.56	1.59
Columns finished	7.96	8.02
Rows finished	3.28	3.08
Received Extra Bombs	1.27	0.95
“color block” action used	1.84	1.65
“bomb” action used	1.24	1.11
“three-in-a-row” action used	1.31	0.83
“heart” action used	0.86	0.68
“stars” action used	0.93	0.61

Table 3.13: The average statistics across 278 games of *“Noch mal so gut!”*, comparing the performance of the RLA and human players.

seems to play a critical role in the agent's strategy, which is rarely completed by human players. This is because the human players tend not to check the blue box at the top right of the board, a key element in the agent's strategic decisions. There is also an outlier in the columns: while human players often complete column *“G”*, the agent rarely does so, but both play styles share a preference for completing columns *”H”* and *“F”*, as well as rows *”R”*, *“Q”*, and *”S”*.
Examining the row and column distributions, it is evident that the overall strategic approaches of human players and the agent do not differ significantly. However, the strong contrast in color preferences is particularly noticeable.

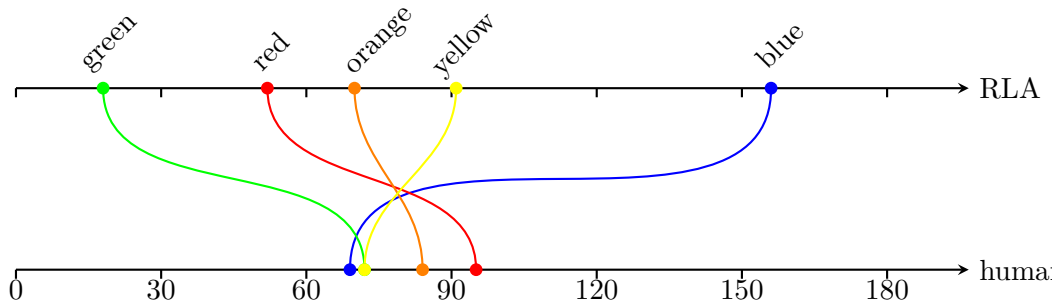


Figure 3.9: Color preferences of our RLA (top) and human players (bottom) across 278 games of *“Noch mal so gut!”*.

At the end of our evaluation, we consider both our agents for the games *“Noch mal!”* and *“Noch mal so gut!”*, respectively, suitable for game testing and move on to the findings

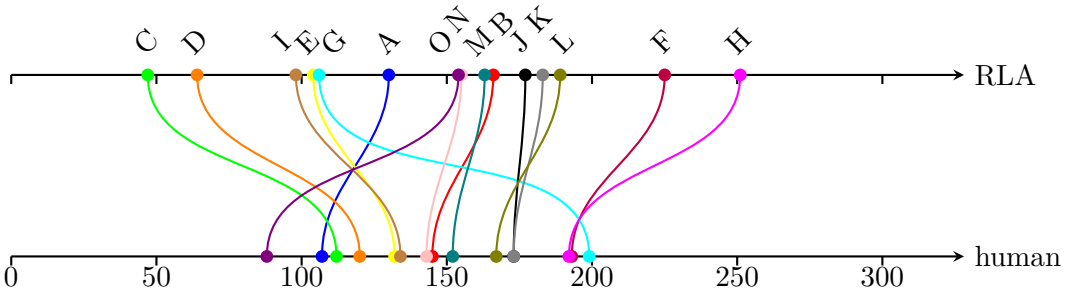


Figure 3.10: Column preferences of our RLA (top) and human players (bottom) across 278 games of *"Noch mal so gut!"*.

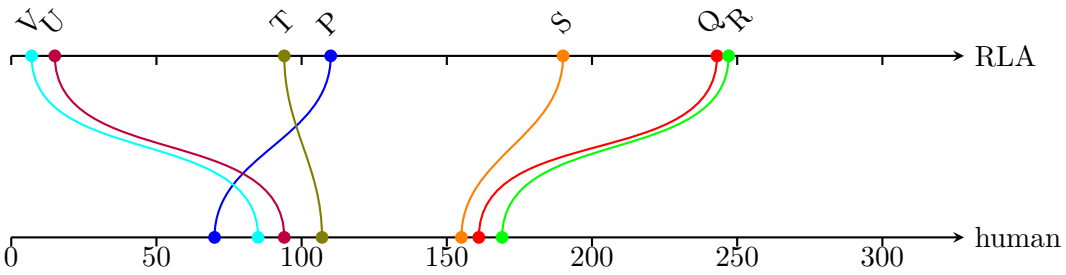


Figure 3.11: Row preferences of our RLA (top) and human players (bottom) across 278 games of *"Noch mal so gut!"*.

from a developer's perspective, which can help to improve the game.

3.5 Findings and Rule Changing Options

The purpose of this section is to show the insights that we get from the agents. In the previous section, we analyzed the different play styles of humans in comparison to our models, but in this section, we want to analyze it from a developer's perspective. As a developer we would not have the statistics of the games against humans, because for a game that is currently being developed there are no experienced testers. Therefore, we want to see what we could find out only with the knowledge of the trained models. For this reason we first generated 1,000 self-play games of *"Noch mal!"* to extract statistics and insights as was done in [PMVI24].

The distribution of wins was evenly matched between the starting and non-starting player, which is usually a good sign of a balanced game. Out of the 1,000 self-play games, the second player won 511 and lost 481, whereas 8 games end up in ties. This slight edge could be attributed to the fact that they are the first to select dice in round four, limiting the opponent's choices. However, overall, there appears to be no significant advantage associated with going first.

On average, a game took 31.33 turns, which is useful for the developer when estimating real-time play length. In 80.8% of the games, the winner finished with two completed colors, while that was only the case in 24.4% of losses. The winner also completed more columns on average – 7.75 compared to 6.42 – showing stronger performance in both main point-generating mechanics. Similarly, the player who won typically used fewer jokers than the loser, with 5.96 on average compared to 6.25, meaning extra points also came from avoiding joker use. The mean score of the winner was 27.28 points, while the loser had 18.25. In general, these numbers give developers a solid benchmark to compare against expectations. If 31 turns per game feels too long, the win condition could be adjusted – maybe requiring only one completed color instead of two. The breakdown of points also helps give a sense of whether the scoring system works as intended or if changes are needed, like adjusting how many points specific columns or colors are worth.

Another important insight of the self-play games that we can find is the heatmap of all crossed fields over all games. As illustrated in Figure 3.12, some fields are much less crossed than others. In the previous section we figured out the color green and orange are very popular in the strategy of our RLA, which was also the case in the self-play games, with green completed 743 times and orange completed 463 times of the winner. On the other hand, the color blue was nearly never finished with only 38 completions. A closer examination of the heatmap confirms that the fields that are not used much are mostly blue fields, like the one on the bottom in row 7 and column *"F"* or the field in row 4 and column *"O"*. Also other fields in colors of red and yellow were only slightly crossed, such as the yellow field in row 2 and column *"F"* or the fields in row 5 and 6 and column *"A"*, which are red.

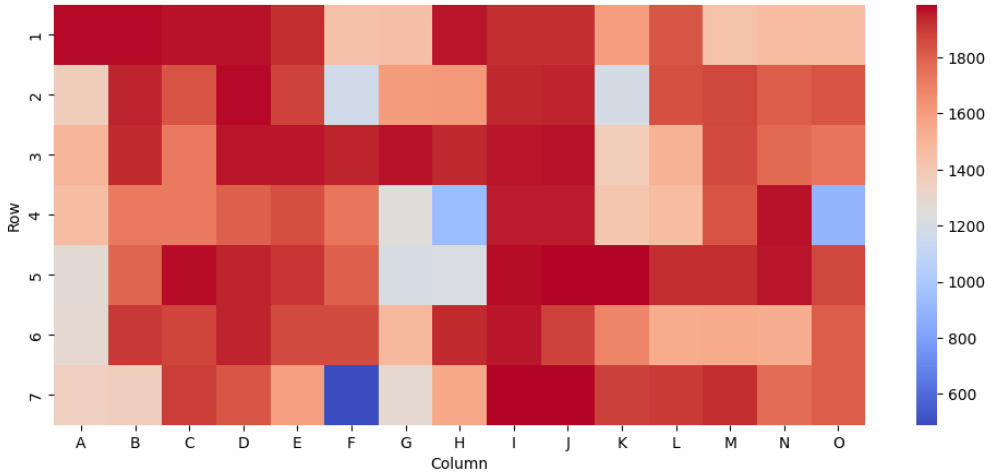


Figure 3.12: A heatmap of all crossed fields of both players across 1,000 self-play games of *"Noch mal!"* by our RLA.

This could be due to the fact that the large green color blocks are mostly positioned on the outer edges of the playing field. Since outer color blocks are selected less frequently –

simply because the opportunity does not arise as often – green becomes a more viable option due to its larger block sizes (consisting of groups of five and four) in the outer columns. Given the choice, players are more likely to complete larger color blocks, as this allows them to mark more spaces and expand their reach on the board. Additionally, the size of the green blocks makes them easier to access, which further reinforces their strategic advantage. As we evaluated in Section 3.4, this is not uncommon for human players either.
Again, for a game developer this information is useful, because now they can think of balancing the board by rearranging the colors. This also was done for the game of *"Noch mal!"*, where there are now several different game boards to choose from [BB19a]. In combination with the other insights, one could also think about whether to make the game board smaller or larger in order to have more or less options as a player.

Concerning the columns, it is interesting that columns *"I"* and *"J"* are finished the most by our RLA. In Figure 3.13 you can see the distribution of the finished columns of both players combined. One might assume that the middle columns are completed most frequently because they are the easiest to achieve most of the time, but the game developers have already balanced the points in such a way that the outer columns are also completed, as they offer higher rewards. It is also interesting that column *"F"* is the one with the fewest completions and also one of the columns that consists entirely of fields in 6-field color blocks.

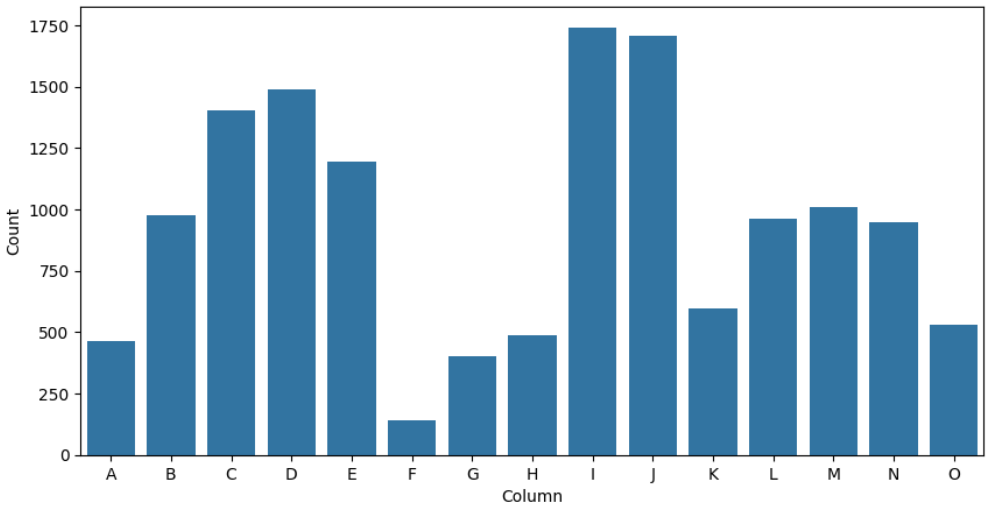


Figure 3.13: The distribution of completed columns of both players across 1,000 self-play games of *"Noch mal!"* by our RLA.

As mentioned before in Section 3, adjustments have been made by the developers and also new rules were added to the game to develop the extension *"Noch mal so gut!"*. Now we can look closely into the second agent that we also trained in the previous section and see if the changes have an impact on the gameplay. We also want to see if the special

actions were used equally from our RLA or if some of them are much more useful than others to it.

First, we examine the overall statistical changes before delving into the agent's new play style. To do this, we again play 1,000 self-play games of *"Noch mal so gut!"* and evaluate them. Of these 1,000 games, the starting player only won 445, which could again be due to the effect mentioned that the non-starting player is the first to take dice in round four that the opponent is not allowed to choose. This effect could be enhanced with the special die, as the bomb, for example, makes it easier to spread all over the board which is particularly relevant at the beginning, because a player instantly get more options in the game.
Another important fact is that the average number of turns taken in a game is 28.32, which is slightly fewer than in *"Noch mal!"* – likely due to the new options and of course bonuses available. The number of completed colors by the winner has decreased slightly. Now, in 75.9% of the games, the winner completes two colors, while in 26.5% of the games, the loser does. The number of completed columns increased on average to 8.83 for the winner and 7.19 for the loser, which could be because on average more boxes are ticked in comparison to the basic game as the agent is not looking to end the game quickly. Also, judging by the number of rows, the winner is far ahead by finishing 3.69 rows on average while the loser only complete 2.60. The use of the jokers is similar for both players because now the winner and the loser typically use 5.43 and 5.38 jokers respectively. For the usage of the special die symbols, the winner on average used 6.10 and the loser 5.85, while having 1.44 and 1.21 left over at the end of the game. This shows that the winner often has more special die symbols available over the whole game. Additionally, by the end of the game, the winner and the loser have an average of 1.70 and 1.25 hearts, respectively. It was clear that the mean scores will rise drastically, so that the average score for the winner is now 58.69 and for the loser 36.74, which is much higher than before. Again, the winner is better in all main point-generating mechanics. This is all valuable information for the developers to analyze their own game and reconsider the rule changes and their impact on the game. Now that we have the basic stats of the 1,000 games, we can dive deeper into the strategy.

The first thing which is clear is that the game behavior has changed completely. Because there are now many points and also extra bonuses for completing rows, the play style changes to completing rows as quickly as possible instead of only completing columns as before. In addition, the change in the game board has clearly resulted in a major change. These strategy changes can be seen drastically in the data. If we look at the heatmap of the game *"Noch mal so gut!"* in Figure 3.14, we can clearly see that the columns were not as important anymore. We can see that the rows *"Q"*, *"R"*, and *"S"* were used much more than the rows *"U"* and *"V"*.

Also interesting is that now the color blue is much more interesting being the most finished color for both players, 699 completions by the winner and 402 by the loser. In second place is yellow with 469 and 204 completions respectively. Amusingly, now green is the color which gets finished least by both players. These color shifts are also due to

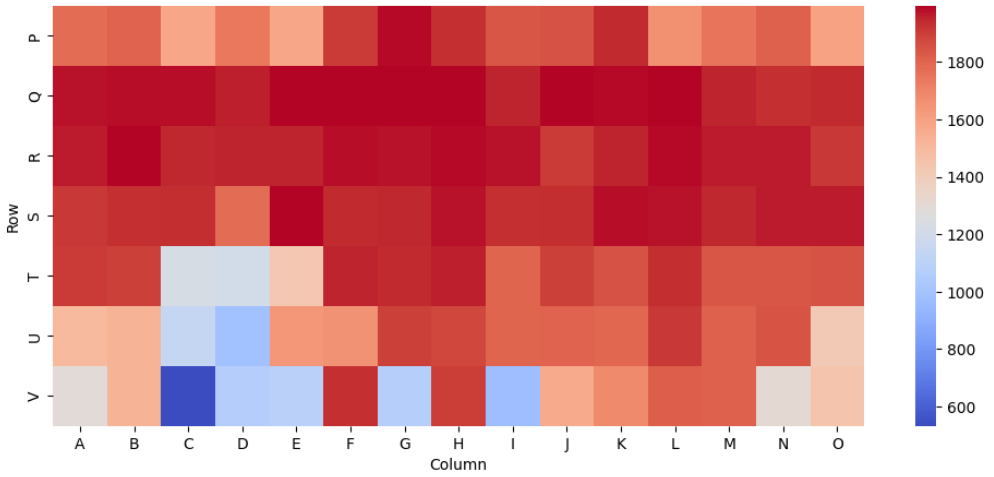


Figure 3.14: A heatmap of all crossed fields of both players across 1,000 self-play games of *"Noch mal so gut!"* by our RLA.

the fact that in the rows *"Q"*, *"R"*, and *"S"*, which are used most often by our RLA, more blue and yellow fields appear than green ones. Also the heatmap shows that the most unused fields are green, red, and orange ones.

We can also see that the developers made column *"H"* more appealing to players by offering more points and an extra special die symbol, which proved to be effective. Figure 3.15 presents the distribution of completed columns, highlighting a shift in strategic focus compared to the original game. Now, *"H"* is the most finished column by our RLA. Also the outer columns are completed more often, because the "bomb" action allows faster spreading on the board.

If we look at the distribution of the rows in Figure 3.16, we see the things which we expected from the heatmap. The rows *"Q"*, *"R"* and *"S"* are completed almost every time, while *"V"* is completed almost never. This could be because *"V"* is one of the rows that is on the very outside of the field, but also has fields of nine different color blocks, while for example *"Q"* or *"S"* only have fields of eight different color blocks in their row. Another reason could also be that for the first finisher of *"Q"* and *"S"*, which are really close to each other, one get an additional bomb to place, which could be a huge bonus in the game. It was clear to observe that in 587 games, the winner had more extra bombs – meaning the bonus bomb from a row completed first – available than the loser, and in 309 games, they had the same amount, which indicates that this bonus is particularly powerful.

Other important insights for game developers are, of course, the use of the new special actions that the game *"Noch mal so gut!"* brings. Therefore, we count how often the RLA uses the different options and display them in a bar plot, which we can see in Figure 3.17. We notice that the special action "color block" is used the most, followed by the "bomb" and the "three-in-a-row" action. Despite the higher availability of the

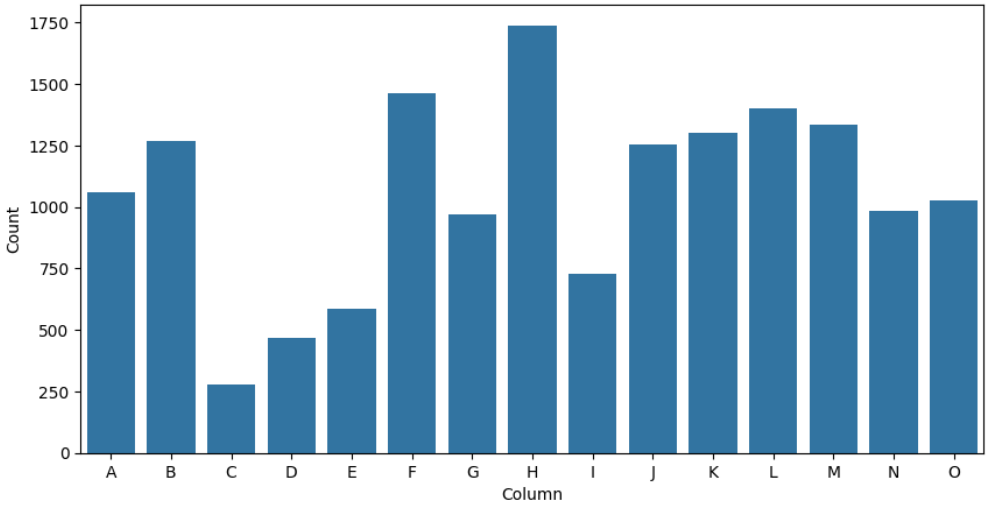


Figure 3.15: The distribution of completed columns of both players across 1,000 self-play games of *"Noch mal so gut!"* by our RLA.

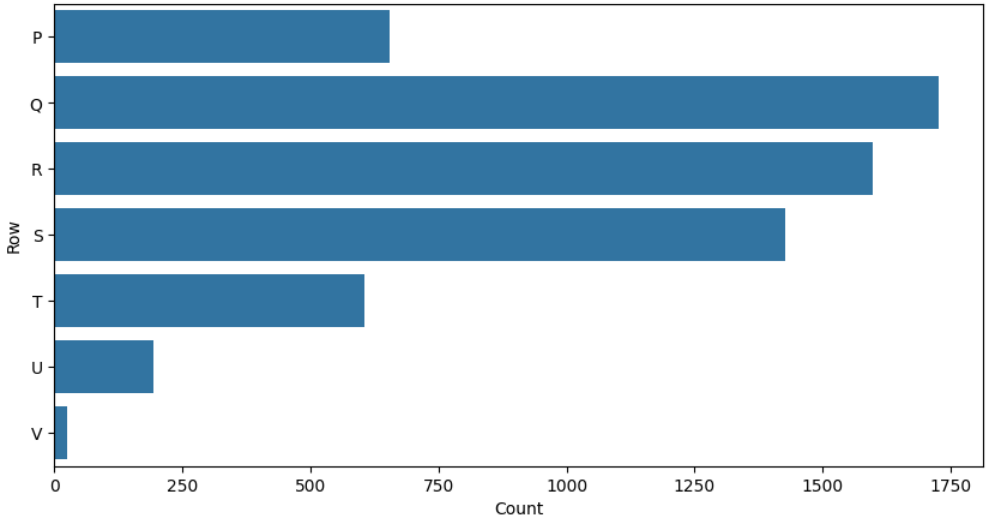


Figure 3.16: The distribution of completed rows of both players across 1,000 self-play games of *"Noch mal so gut!"* by our RLA.

"heart" action (with two hearts on the 6-sided die), both "heart" and "stars" are used less frequently. This is also an indicator that these two options are less useful than the others.

To get more specific, we can analyze the actions the agent takes with the top 3 specials. First, we focus on the "color block" action, where the key question is how often the RLA uses it to complete a whole 6-field color block, which is only possible with this special

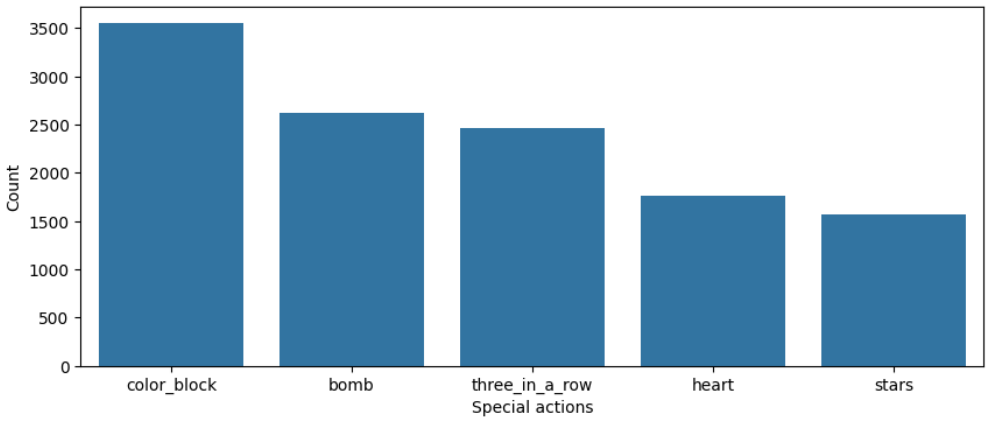


Figure 3.17: The frequency of special actions used by both players across 1,000 self-play games by our RLA.

action. Surprisingly, the agent fills a 6-field color block in only 13.92% of cases when using this action. The most used actions are shown in Table 3.14.

Fields	Num_Fields	Count
[[1, 12], [2, 12], [2, 13], [3, 13], [3, 14]]	5	381
[[0, 9], [0, 10], [1, 10], [1, 11], [2, 11]]	5	350
[[4, 11], [5, 9], [5, 10], [5, 11], [6, 8], [6, 9]]	6	256
[[1, 4], [1, 5], [2, 5], [2, 6], [2, 7]]	5	201
[[4, 12], [4, 13], [4, 14]]	3	198

Table 3.14: The most used actions with the "color block" action by both players across 1,000 self-play games by our RLA.

Next, we look at the "bombs" actions, where we notice that there are a few locations that the RLA found which might be better than others. The top 5 locations can be seen in Table 3.15. It is interesting that the top 3 locations are all in columns *"A"* and *"B"*. More importantly, they consistently complete a whole color block and sometimes even cross a field of a 6-field color block, which is a clever strategy.

Remarkably, the top three bomb placements identified by the RLA align with the top three choices made by human players in test games. This consistency suggests that the RLA has developed an effective strategy purely through self-play.

The results are also very interesting regarding the rows. As we can see in Table 3.16, the "three-in-a-row" action is mostly used to check boxes in the rows *"P"* and *"S"*, but almost never for *"Q"*, *"R"*, and *"U"*.

All in all, these insights are very important for game developers. Of course one could say that this is one strategy of many in this game, but with the knowledge that this strategy wins nearly 50% of the games against experienced players, developers might

Fields	Count
[[3, 0], [3, 1], [4, 0], [4, 1]]	1461
[[5, 0], [5, 1], [6, 0], [6, 1]]	1093
[[1, 0], [1, 1], [2, 0], [2, 1]]	751
[[5, 3], [5, 4], [6, 3], [6, 4]]	457
[[5, 5], [5, 6], [6, 5], [6, 6]]	141

Table 3.15: The most used bomb locations by both players across 1,000 self-play games by our RLA.

Row	Count
P	1559
S	692
V	121
T	78
Q	7
R	1
U	1

Table 3.16: The most used rows with the "three-in-a-row" action by both players across 1,000 self-play games by our RLA.

want to change something. Maybe the intention is not to have some bomb locations that are much more powerful than others, in which case a rearranging of the board would be the solution. Or since the data indicates that the option "stars" is used less frequently, one could think of changing the option to cross three stars instead of two. Maybe then the action would be as frequently used as the "color block" option. Also the basic information about the duration of the game and the distribution of points is helpful to further develop the game. Thus, the RLA provides valuable information about the game, such that a game developer can see how to improve the game balance. We also notice in the development that changes from the base game *"Noch mal!"* to the expansion *"Noch mal so gut!"* have changed several aspects, some of which with quite significant implications.

Now, the developer would have to decide, which changes are beneficial to the game or which rules should be adjusted. Then we could again let a RLA learn the new rule set and try to find weaknesses with its help. This iterative process could go on and on until the developer decides that the game is good and balanced enough to finish developing. For these two games, this process would be too extensive for this thesis. However, to provide a better understanding of the iterative process, Chapter 4 introduces our own simple board game and presents step-by-step instructions for creating it with the help of RL.

CHAPTER 4

Simala

This chapter is intended to show the iterative process of game development with a RLA. By introducing our own game "Simala", we want to show a practical example how the development process can be improved with the help of RL. First, we explain the game environment and the architecture of our agent and then proceed to change the game rules to balance emergent strategies.

4.1 Game Concept

"Simala" is intended to be a board game based on strategy and luck for two players. The idea behind the game is that both players play heroes who try to defeat a certain number of monsters or slay the final boss during the course of the game. Along the way, they can buy items, improve their own skills to be stronger in battle, and also engage in player versus player combat to prevent the other from winning. Since this game is not yet developed, all rules should be treated as a first draft of the game to have a framework in which we can balance the game.

4.1.1 Simala – Game Instructions

Game Contents and Setup

The game is played on a board with 24 fields arranged in a loop, which can be seen in Figure 4.1. Each field shows the actions that can be performed there, which are explained in detail later on. Both players receive a 6-sided die and a character sheet showing how much health points, maximum health points, attack points, gold, experience points, trophies, and equipment they currently have and which next reward they will get if they level up. Each player starts with the stats listed in the table of Figure 4.1. In addition, each player places a figure on the "Shop"-fields at the bottom right and top left corner, respectively, each representing one player. Two 6-sided dice are also provided, featuring

the numbers 1 to 5 and one ”boss” symbol. Lastly, the level 1 and level 2 monsters are shuffled and placed in a pile and the boss is placed next to the playing field.

Character stats

health points	5
maximum health points	5
attack points	1
gold	0
experience points	0
trophies	0
inventory	[ ]
next reward	attack

Figure 4.1: The game board and starting stats of each player in the game “Simala”.

Gameplay

Each turn consists of the same phases:

1. If the current player has above 0 health points, they roll both dice, if not they must skip this round and regenerates their health points.
2. Then they choose one of the dice to move as many fields clockwise on the game board as the chosen die indicates.
3. After that they choose one of the given actions on the field they landed on.
4. At the end of the turn, both players check whether they have moved up a level and whether they fulfill the victory conditions. This can only happen to the non-current player if the players fight against each other.

The actions that can be performed are:

- **Well (“We”)**: The player gets fully healed, i.e., their health points are set to their maximum health points.
- **Mine (“Mi”)**: The player rolls their 6-sided die and receives one gold if the number is 4 or above.
- **Train (“Tr”)**: The player receives one experience point.

- **Fight against a Monster of level 1 ("M1")**: A level 1 monster is drawn from the corresponding draw pile and the player must fight it (see "Fight" below). The level 1 monster has between 5 and 7 health points, between 1 and 3 attack points, and no inventory. For defeating it, one receive 2 experience points, one gold, and a trophy.

- **Fight against a Monster of level 2 ("M2")**: A level 2 monster is drawn from the corresponding draw pile and the player must fight it (see "Fight" below). The level 2 monster has between 7 and 9 health points, between 3 and 5 attack points, and a bow in its inventory. For defeating it, you receive 3 experience points, two gold, and a trophy.

- **Shop ("Shop")**: The player is allowed to buy one item for their inventory using their gold. If so, they place it on their character sheet, because every player can only own each item once. The costs and effects of the respective items are listed in Table 4.1.

Item	Cost	Effect
Sword	2	+1 attack point
Helmet	1	+1 health point and +1 maximum health point
Shield	2	Reduces the opponent's attack die by 1
Bow	2	Increases own attack die by 1

Table 4.1: Items available in the shop.

- **Fight against the other player**: If the opponent occupies the field you intend to move to, you must engage in a fight (see "Fight" below) and cannot take any action associated with that field. The victorious player earns 2 experience points and claims all of the opponent's gold. This is the only way the non-active player can level up in the opponent's turn.

- **Fight against the Boss**: If both dice show the "boss" symbol, the player is attacked by the boss and must fight against it (see "Fight" below). This also means the player cannot choose any option in Phase 2 and 3. The boss has 10 health points, 6 attack points, and a shield and bow in its inventory. For defeating it, you immediately win the game.

The **"Fight"** proceeds as follows:
The player whose turn it is attacks first and rolls their die. Any bonuses or penalties from their bow or the opponent's shield are then applied. If the total result is greater than 3, the opponent loses health points equal to the attacking player's attack points. If the opponent still has health points (i.e., above 0), they now attack by rolling their die in the same manner. If the opponent is a monster or the boss, the other player rolls the dice for it. Again, bonuses or penalties from the bow or shield are applied, and the result

is checked. If the total is greater than 3, the current player loses health points equal to the opponent's attack points. This process repeats until one of the fighters is reduced to 0 or fewer health points. The combatant who still has health points wins the fight.
If a player drops to 0 health points or below, they are placed back on their starting square and must sit out the next round to fully regenerate their health points. So every time a player loses a fight, they get placed back.
If a monster or the boss was involved in the fight, the associated card is placed back, and the health points of the creature is reset. After that, the respective monster deck (level 1 or level 2) is shuffled, i.e., no monster gets discarded, so that every time a player fights against a monster, it is randomly picked from all existing ones.

At the end of each turn, if a player has gathered 3 or more experience points they have to exchange 3 experience points for the next reward on their character sheet. The rewards alternate between "attack" and "health", meaning the player receives +1 attack point on one cause, and +1 health point together with +1 maximum health point on the next, and so on.

Game End

The game ends immediately if one of the players has collected five trophies or defeats the boss in a boss fight. That player is the winner of the game.

4.2 Environment Architecture

For the game environment, we also implemented the game "Simala" in Python, following the same approach as with the games described in previous Sections. Again, the game is designed to be tested in the terminal so that we can check its functionality and later test our RLA.

4.2.1 Game

This class holds all the methods that we explained in the game description such as rolling the dice, tacking actions, and fighting against monsters. Additionally, all the relevant information for the DQN is implemented, including the action space, the actual observation state of the current situation, and the option to perform random legal actions to encourage exploration. All relevant game data, such as the board state and character sheets, are also stored and can be easily adapted in the future, should we need to make modifications.

4.2.2 Observation State

As we learned, the observation state should only include information that the model can use to learn and make decisions within the game. Elements that remain constant throughout the game are not useful, as the model cannot learn from them. Therefore,

we choose not to include the exact game board in the observation state, but rather the players' positions on the playing field. Again, all features are put into a list, so that the RLA then can use and interpret the current state. The features for the game "Simala" are:

- The positions of both players are represented using one-hot encoding. 24 integers are used to represent each space on the game board. One set of 24 integers is used for the figure controlled by the model, while another 24 integers represent the position of the opponent's figure on the game board. A value of 1 indicates the position where a piece is located, while all other positions are marked with 0.
- All stats for both players are also stored in the observation state, by reserving 11 integers for every player for all stats represented in the table of Figure 4.1. The inventory is also one-hot encoded, where each digit of the 4-digit list represents one item.
- Lastly, we also put in the current dice like in the games of *"Noch mal!"* and *"Noch mal so gut!"*.

We end up with a feature vector of size

$$\underbrace{48}_{\text{player positions}} + \underbrace{22}_{\text{player stats}} + \underbrace{6}_{\text{dice}} = 76.$$

4.2.3 Action Space

For the action space, it is straightforward to see that each option in Phase 2 (choosing a die) combined with the action in Phase 3 (choosing an action) must correspond to one action in the DQN. Therefore, there are seven actions that can be taken, whereas the "Shop" action has to be split into five: buy a sword, a helmet, a shield, a bow, or choosing to pass, which are all distinct actions. We also decide to include the "fight boss" action in the action space, even though it is not optional. We considered the possibility that, in future rule sets, it could become optional. This brings the total number of actions in the action space to

$$\underbrace{5}_{\text{number die}} \cdot \underbrace{11}_{\text{actions}} + \underbrace{1}_{\text{fight boss}} = 56.$$

4.3 Agent Architecture

For the RLA, we use the same network structure as we used for the previous games, so that the self-playing program learns based on rewards that are given by a win or a loss. Because we took the same approach as in Section 3.3 we shall only list the differences here:

- **Network**: Naturally, we have to change the number of neurons for every hidden layer. Since now the number of features is 76 and the action size is 56, we decide to have fully connected layers with 60 neurons.
  Another difference is that we now use a target network for stable training. This target network, as explained in Section 2.3, is updated every hundredth game, i.e., after 100 games the weights of our main training network are copied and put into the target network.

- **Training loop**: The training loop is the same as in Algorithm 3.1. For our reward, we also took the last one we used for *"Noch mal!"* and *"Noch mal so gut!"*, i.e., +1 for winning and -1 for losing. In "Simala", no tie is possible, because the first player to fulfill any win condition wins. We also had no exceptions to take for the reward, because no option could be generally considered as bad.
  As mentioned, we have to add the target network update in the training loop which can be seen in Algorithm 4.1.

Algorithm 4.1: Update Target Network

Require: Target Network $\mathcal{T}$, Current Network Q_n

1: **if** e mod 100 == 0 **then**
2: $\quad\mathcal{T} \leftarrow Q_n$
3: **end if**

Similarly, for Algorithm 3.4, where the training of the network is carried out, we also have to add the Target Network in line 9:

Algorithm 4.2: Train on Batch with Target Network

Require: Memory $\mathcal{M}$, batch_size, discount_factor γ, Target Network $\mathcal{T}$

1: **if** e mod 10 == 0 and len($\mathcal{M}$) > 1000 **then**
2: $\quad$batch $\leftarrow$
   Get batch_size random samples from memory; sample_obj = (s_t, a_t, r_t, s_{t+1})
3: $\quad$create Q-values-table
4: $\quad$**for** each sample_obj in batch **do**
5: $\quad\quad$**if** the game ended after this move (no next state available) **then**
6: $\quad\quad\quad$Update Q-values-table with r_t
7: $\quad\quad$**else**
8: $\quad\quad\quad$Mask new state s_{t+1} with legal actions
9: $\quad\quad\quad$Update Q-values-table with $r_t + (\gamma \cdot \max_{a_{t+1} \in A_{legal}} \mathcal{T}(s_{t+1}, a_{t+1}))$
   $\quad\quad\quad$ {Adjustment: Target Network $\mathcal{T}$ used for next-state evaluation}
10: $\quad\quad$**end if**
11: $\quad$**end for**
12: $\quad$Train network with Q-values-table
13: **end if**

- **Hyper-parameter tuning**: For the hyper-parameter tuning, we use the same parameters as outlined in Section 3.3.3 for the main network. Most of these parameters are kept fixed, and we only test the ones listed in Table 4.2. As shown, we decided to test a lower number of hidden layers, as the game's complexity does not require a deep network for effective learning. The discount factors are set relatively high, as foresight plays a crucial role in this game, helping the agent plan and prepare for upcoming fights.

Hyper-parameter	**Values Tested**
hidden_layers	[3, 5, 8]
learning_rate	[0.0001, 0.001, 0.01]
discount_factor	[0.9, 0.95, 0.99]
Number of episodes tested	100,000

Table 4.2: The set of hyper-parameters tested in each iteration of the rule configuration process in "Simala".

We will use these parameters for each rule configuration. After each training phase, we will determine the best model through a tournament, as described in the previous chapter, and then play against it ourselves. Since we have already covered the evaluation phase in detail, we will only discuss the strategies of each model here and fully elaborate the evaluation process. Furthermore, for a game which is not yet fully developed, there are no experienced testers who could recognize which way of playing is good or bad. Therefore, we test the models against randomly playing agents and ourselves to see if the trained models play randomly or follow a strategy.

4.4 Iterative Development Process

Now that we have set up the DQN and fixed the rules, we can dive deeper to balance the game. The goal of this section is to go through an iterative process as follows:

- Let the RLA learn the game and determine the best one through a tournament.
- Let the best RLA play 1,000 games.
- Analyze the stats of the games, the strategy, and the behavior of the RLA.
- If the game does not meet our expectations, refine the rules based on the data.

This process aims to develop a well-balanced board game. For us, "balanced" means in this case that no player has an advantage going first or second, that no single strategy always wins, and that the actions taken are used about as often as intended. For example, it would not be good if every second turn the player has to go to the well because they have no health points or that you can win the game without buying items. Clear is also

that the action "M1" will be used more often than "M2", which is intended because level 2 monsters are supposed to be much stronger and thus should be more difficult to defeat. This is why they also give slightly higher rewards in the game. Additionally, the action to fight against the other player or the boss will be much less used, because these are actions that can only be taken in specific situations. For the first iteration, we do not expect everything to be perfectly balanced yet. It is likely that some actions will be used way more frequently because they are simply better than others. We also expect that we will have to adjust the stats of the monsters and possibly even rework actions completely because they are not yet fully thought through.
In the following sections, we will go through the analysis of every rule set and point out the weaknesses that should be improved in the next iteration. We also will show only the play styles of the best RLAs in each iteration, but we note that all the top agents have a similar strategy as the best one.

4.4.1 First Iteration

In the first iteration, we train our models based on the rule set described in Section 4.1.1. We then run a tournament between all models using the hyper-parameters from Table 4.2 and find that the best-performing model for these rules has three hidden layers, a learning rate of 0.001, and a discount factor of 0.95. While testing, we also realized that the game relies too heavily on luck and players often do not have enough meaningful choices, so that they often only have one single option to begin with. This occurs in situations where a double or the "boss" symbol is rolled, as only one number is available in Phase 2. Despite this, we start with the analysis of the game style and the game stats by looking at 1,000 self-played games.
The first thing we wanted to check was whether the starting player has an advantage and wins more often. According to our data, the starting player won 540 out of 1,000 games, which suggests that the game is fairly balanced in this regard, but should not deviate any further.
Looking at the win conditions, we notice that in only 26 of the games the player won by defeating the boss, i.e., the game was nearly always won by getting five trophies. This result deviates from our initial expectations and suggests an imbalance in the game's mechanics. Our analysis shows that there is a huge difference between the player values and the boss's attributes. While the players end the game with an average of 7.3 health points and 3.75 attack points, the boss has 10 health points, 6 attack points, and additional equipment such as a shield and a bow.
We also observed that players purchase only one item per game on average, despite the possibility of carrying up to four. This suggests that either the incentives for purchasing items are insufficient or that the cost-benefit ratio discourages players from making additional purchases.
Finally, we create a heatmap of the positions of the players during the game. In Figure 4.2 we can see very quickly that the players are mainly on their starting fields. These outliers occur because every time they lose a battle, they are placed back on their starting square.

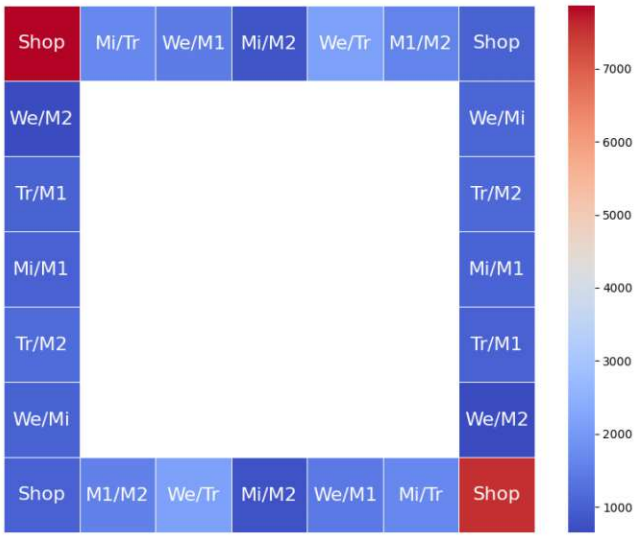


Figure 4.2: A heatmap of the players' positions across 1,000 self-play games of the first iteration by our RLA.

As we look at the actions distribution of our RLA in Figure 4.3, we also notice that one reason for that is that fighting against a monster is the most used action. That means our agent often tries to fight against a level 1 monster and if it loses it gets reset to its starting position. Another aspect we will aim to improve in future iterations is achieving a better balance between the number of fights against level 1 and level 2 monsters. Now, the "M1" action is more than nine times more used than the "M2" action and over the whole game only 0.17 fights against monster level 2 are won on average. This indicates that the strength of the monsters is very imbalanced and the incentive to get more rewards for stronger monsters is far too small.

In the graphic, we also see that the "mine" action in comparison to the "train" action is much less used, which could be the reason why not so many items were bought. Therefore, we decide to balance these options slightly.

Based on all of these insights and errors of the game, we come up with a few rule changes, which are listed below:

- A player will no longer be reset to the starting field if they lose a battle, so that the players have more possibilities to explore the map in one game.
- As there were too few options for the players, we decided that if a double is rolled, the player may also choose the adjacent fields. In concrete terms, this means that if a player rolls a double of 2, they can also choose between the numbers 1 and 3, and in the case of a double of 5 or 1, they can also choose the number 1 and 4 or 2 and 5, respectively.

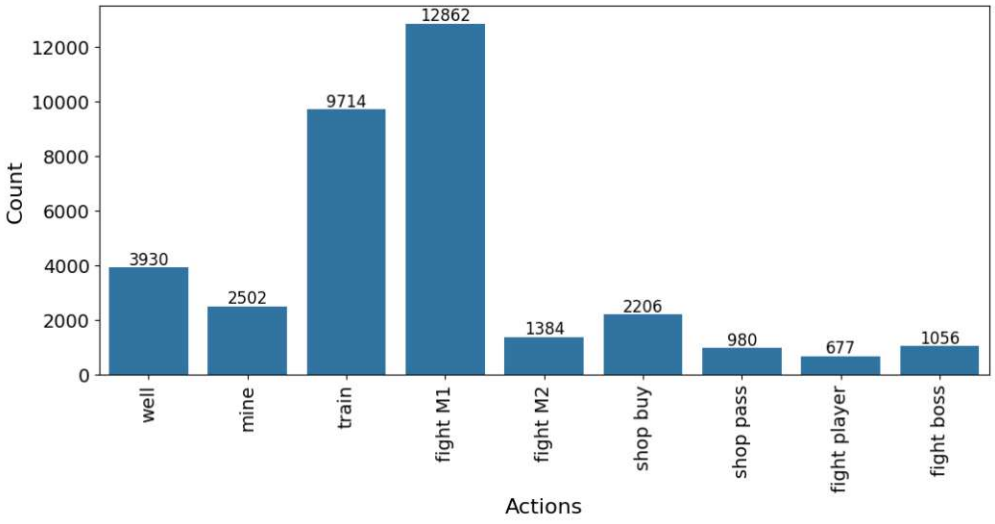


Figure 4.3: Action preferences of our RLA across 1,000 self-play games of the first iteration.

- To improve mining, we decide that a player now gets one gold every time they mine, but they still have to roll a die. If this shows 3 or less, the player loses one health point. This damage should also make using the well more attractive.
- For the training mechanism, we apply the same approach. Each time a player trains, they have to roll a die. Specifically, if the die roll results in a value of 3 or lower, the player loses one health point.
- Finally, we improve the level 1 monsters so that they are less likely to be defeated without any character improvement. Now they have health points between 6 and 8 and attack points of 2 or 3. We also increased the reward for level 2 monsters so that a player now receives two trophies for defeating them. This should give players more incentive to choose this option more often. We also lowered the attack points of the boss to 5 to make this win option easier.

4.4.2 Second Iteration

In the second iteration, we now edit the game class with our new rule settings and let the models train on the game. Again, after 100,000 episodes we play a tournament, in which this time the configuration with eight hidden layers, a learning rate of 0.001 and a discount factor of 0.9 came out on top. This time, there were sometimes more options available, because of the double rule, which makes the game more strategic.
From the analysis, we observe that 63 out of the 1,000 self-playing games resulted in a victory by defeating the boss. While this represents a notable improvement, the boss is still too strong to defeat, which influences the game strongly. This is especially evident in the early turns, where a player often wastes two turns – one due to defeat and another

for recovery. Also 532 of the games were won by the starting player, which indicates a slight improve of balance. The total items that are bought increased to 2.5, which is a good indicator that the new “mining” rule worked, so that it is now beneficial to buy more items. It is very likely that the stronger level 1 monsters also led to this reaction. By looking at the heatmap of the positions in Figure 4.4, we can also see that the players now can explore the map more. Some of these fields are used more often, which is also confirmed by the actions distribution in Figure 4.5.

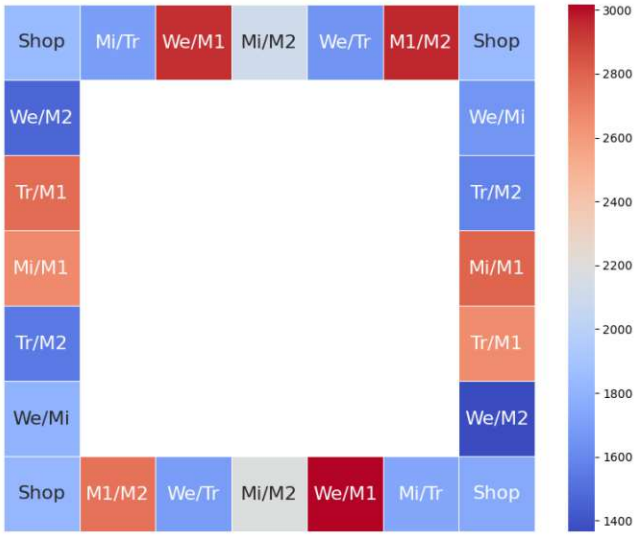


Figure 4.4: A heatmap of the players positions’ across 1,000 self-play games of the second iteration by our RLA.

Again, we can see that the difference between fighting against a level 1 monster and a level 2 monster is far too big and “M1” is used too often by our RLA. That means that the rule change of the first iteration was too weak to convince the player to use “M2”, which we should improve in the next iteration. The figure also shows that mining is now used much more frequently, while the use of training has decreased a bit. This shift is a direct response to the new rules we introduced.
We also realize that the new rule to give the player more options by rolling a double is too weak, because the problem of rolling a "boss" symbol still stands, which still leaves the player with just a single die to choose. One supporting indicator is the use of the “shop pass” action, which is never beneficial, but has to be taken sometimes with these rule configurations.

One observation is that the “well” action becomes a bit less appealing in comparison to mining and training. This shift in preference may be attributed to the fact that regenerating after death provides similar benefits to using the well for regeneration. Consequently, players may feel incentivized to take greater risks, as death may not be as detrimental. If a player’s risky strategy fails, they will receive a “free” regeneration in the following round, as with a well, resulting in making the well less critical in certain

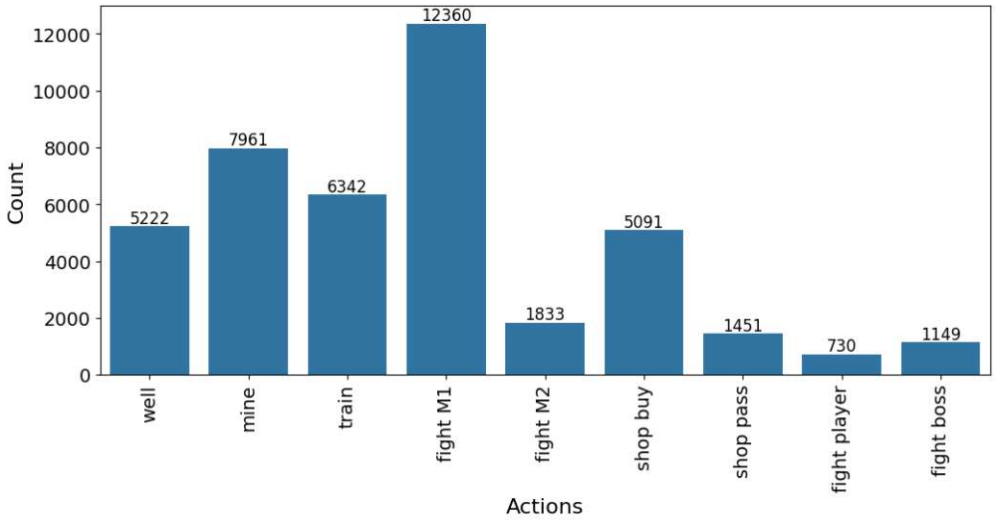


Figure 4.5: Action preferences of our RLA across 1,000 self-play games of the second iteration.

scenarios.
Interesting to see is also that the average turns a game takes rose from 42.5 in the first iteration to 50.3 in the second, which also has to be a result of buffing the level 1 monsters. Our analysis also shows that losing a fight against the boss has a strong impact on the outcome of the game, as the losing player tends to have more boss losses on average than the winner. As already mentioned, it is almost impossible to defeat the boss in the first few rounds, so we have decided to change this.
Another thing that we notice is that the maximal health points are far too low to win against level 2 monster or the boss, because on average the players again end up with 7.1 maximal health points, which becomes difficult compared to possibly 9 or 10 health points on the opponent's side.

Based on all these insights we again change the rules in the following:

- To weaken the level 2 monsters and the boss further, we decrease the attack points to 3 or 4 and 4 respectively. We also decide to reduce their inventory so that the level 2 monsters have no items and the boss only has a bow.
- To improve the "well" action, we decide to decrease the regeneration after a death. From now on, players will only regenerate three quarters of their maximum health points (rounded up) if they died last round. This means that the well is the only way to heal completely.
- To bring even more options into the game, the double rule of the last iteration is also extended to "boss" symbols. This means that if a player rolls a number and a "boss" symbol, they can also choose the adjacent fields of the number die this turn.

- To counter the problem with less health points, we buff the helmet by one health point.
- Lastly, we also restrict access to the boss for in the first 20 turns. This means that if a player rolls two "boss" symbols, they are not attacked by the boss, but instead are required to re-roll the dice until a different outcome is achieved. This should reduce the problem of early boss fights.

4.4.3 Third Iteration

In the third iteration, we again train our models and identify the best one, which features are five hidden layers, a learning rate of 0.001, and a discount factor of 0.95. Again, we get closer to the 50% win ratio, having 510 wins for the starting player, which is a good signal for a game developer. This time, only 54 games end by defeating the boss, which is due to the fact that much less boss attacks happen, because of the "20-turns-restriction" rule, which we set up after the last iteration. Auspiciously, the maximum health points on average increase to 8.1, which is a clear response to the helmet buff.

If we now examine the action distribution in Figure 4.6, it is clear that all our rules have had a significant impact. With more action choices, due to the extension of the "double" rule to include boss dice, we observe a decrease in the frequency of the "shop pass" action. Additionally, the well has become more important due to poorer regeneration after death, and training has become less attractive because of the higher rewards for defeating level 2 monsters, which are now being utilized. However, the most notable achievement is the balance between level 1 and level 2 monster fights, which are now in a much better ratio. Since level 1 monsters are easier to defeat, our intention is for the number of level 1 monster fights to exceed that of level 2 monster fights.

After this iteration, we decided to conclude the game, as continuing further would risk extending indefinitely, which goes beyond the scope of this thesis. Of course one could argue that the numbers of fighting against the other player or fighting against the boss should be higher, but we think that these are just actions supposed to add fun, but do not have to be used more frequently in our game. At the end of the day, these iterations should be an illustrative representation of how RL can be integrated into the development process as an assistance, showcasing the effective approach taken in this work to demonstrate its potential.

4.4.4 More Rules

There are several rule and design changes for further development, a few of which we would like to mention here but cannot analyze further in this thesis:

- One idea is to give the boss a fixed number of health points, which decrease each time the boss is hit by a player, but are not reset after the fight. This means that, in every subsequent fight, the boss could lose health points, and the player who delivers the final blow wins.

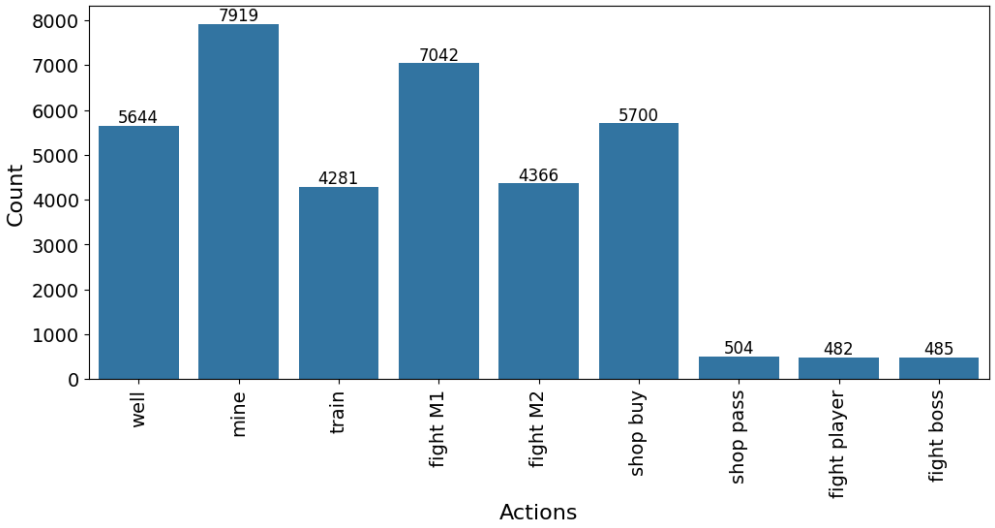


Figure 4.6: Action preferences of our RLA across 1,000 self-play games of the third iteration.

- Another idea is to introduce more items in the shop, each with significantly more effects and impact on the game.
- Additionally, one could consider a 3- or 4-player game, where player interaction is increased due to more fields being occupied by others, forcing players to engage in fights.
- Finally, one could consider changing the map entirely, so that players are not restricted to a fixed path but instead have the freedom to explore a larger, more open map on their own.

CHAPTER 5

Conclusion

This thesis explored the use of RL as a tool for analyzing and optimizing board game mechanics, particularly in the games *"Noch mal!"*, *"Noch mal so gut!"*, and the newly developed game "Simala". By implementing and training RLAs, we investigated their ability to identify strategic patterns, evaluate rule changes, and provide insights into game balance and mechanics.

Our results show that RL can be used as a powerful support for game development. In the games *"Noch mal!"* and the extension *"Noch mal so gut!"*, we first showed that we could develop a good RLA, by competing successfully against human players, achieving a win rate of approximately 50%. This indicates that the agents were capable of understanding and executing effective strategies, making them suitable for evaluating the game's design. We then used this for further analysis to find strategy differences between the base game and its extension. By studying the frequency of certain agent decisions, we provided insights into the relative strengths and weaknesses of different game elements, helping to determine whether the introduced rule modifications were justified from a game balance perspective.

Furthermore, our work on "Simala" showcased the potential of RL as a game design tool. Since "Simala" is still in development, the data from our RLAs provided an efficient method for refining the game's rules without relying exclusively on traditional, iterative human playtesting. By allowing the agent to explore various strategies under different rule conditions, we were able to assess the impact of rule changes on win rates and strategic diversity. This iterative approach ensured that "Simala" did not have a dominant strategy by mainly using one of the predetermined actions, and also tried to ensure that the vision of the game was actually realized in the game. The ability to rapidly test different rule sets through RL simulations highlights a major advantage: reducing subjective bias and speeding up the playtesting process.

Despite these promising results, our research has also uncovered important challenges in using RL for game development. One of the main concerns is to ensure that the RLAs are strong enough to provide meaningful insights. If an agent is unable to discover good strategies, the data it generates may lead to incorrect conclusions about game balance, potentially resulting in unnecessary or even harmful rule changes. This problem underlines the importance of appropriate training techniques, where the reward function can be crucial, but also do not emphasize certain strategies in order not to interfere too much with the open learning process. In our experiments, we tried to use objective rewards that interfere as little as possible with the style of play so as to not dictate a particular strategy. This approach guarantees that the RLA does not approximate a human strategy and can thus explore the game itself. However, it might also mean that good results are never achieved, as the RLA never becomes as good as humans.

Another challenge is the interpretability of RL-based decisions. Even if an agent favors certain moves or strategies, it is not always easy to understand why it makes these decisions. This lack of transparency can make it difficult for game developers to learn from the play style of AI, as small rule changes could shift the entire strategy. Future work could explore methods to improve explainability in RL-based game analysis, such as visualization tools or hybrid AI-human playtesting approaches.

For further research, this work contributes to the development of board games using AI. One possible direction is to extend the use of RL to a wider range of board games with varying complexity and mechanics. While our study focused on dice-based games with discrete decisions, RL could be applied to more complex strategic games that involve hidden information, multiplayer interactions, or more advanced game states.

Furthermore, the games analyzed in this work can also be used for research purposes, for example to investigate the effects of the proposed rule changes. Above all, however, it is important to understand how these changes are perceived by human players. After all, these efforts only make sense if they ultimately lead to games that are not only balanced, but most importantly fun for the players. In this context, further research into what makes a game truly engaging for people would be very valuable, as our work has often raised the question of whether a game with equally strong strategic options is also engaging for its players.

Additionally, future research could investigate how RL can be integrated with evolutionary algorithms or other AI techniques to develop entirely new board game concepts. Instead of just optimizing existing games, AI could be used as a creative partner in the game development process to propose new mechanisms and evaluate their playability, as was done with “Match 3” [SKJK20].

Another promising way is to combine RL with human feedback mechanisms. By having human testers guide or refine the training process, AI agents could develop strategies that better match human preferences, resulting in an engaging and entertaining gaming experience. This hybrid approach could bridge the gap between purely AI-driven optimization and traditional human playtesting.

Overview of Generative AI Tools Used

Generative AI tools were only used as assistance in the thesis. ChatGPT helped to increase the expressiveness and readability without changing my own arguments and for proofreading. DeepL was employed for translating individual words or phrases. ChatGPT supported debugging tasks and reduced the effort for simpler tasks by providing code snippets. Additionally, GitHub Copilot was used to assist with coding, helping to streamline development. The outputs generated by these tools were treated carefully and only served as starting points or checks for my own formulations and code. The ideas and arguments of this thesis were not generated by any AI Tool.

List of Figures

List of Tables

List of Algorithms

Acronyms

AI Artificial Intelligence. 1, 2, 5, 41, 70

DQN Deep Q-Network. 7, 10, 14, 15, 25, 26, 30, 58, 59, 61

DRL Deep Reinforcement Learning. 16

MDP Markov Decision Process. 11–14, 73

ML Machine Learning. 1, 2, 5, 7, 29

RL Reinforcement Learning. 1, 4, 5, 7, 8, 11, 12, 14–17, 53, 55, 67, 69, 70, 73

RLA Reinforcement Learning Agent. 4, 5, 8–10, 15, 17, 19, 25, 26, 28–30, 32, 34, 40, 42–53, 55, 58, 59, 61–66, 68–70, 73–75

Bibliography

[ACF22] M. M. Afsar, T. Crump, and B. Far. Reinforcement learning based recommender systems: A survey. *ACM Computing Surveys*, 55(7):1–38, 2022.

[BB16] I. Brand and M. Brand/Schmidt. Noch mal! DE. https://www.schmidtspiele.de/files/Produkte/4/49327%20-%20Noch%20mal!/49327_Noch_Mal_DE.pdf, 2016. [Online; accessed 31-March-2025].

[BB19a] I. Brand and M. Brand/Schmidt. Noch mal! - Zusatzblöcke. https://www.schmidtspiele-shop.de/noch-mal-zusatzbloecke-nr-iv-v-vi, 2019. [Online; accessed 31-March-2025].

[BB19b] I. Brand and M. Brand/Schmidt. Noch mal so gut! DE. https://www.schmidtspiele.de/files/Produkte/5/49365%20-%20Noch%20mal%20so%20gut!/49365_Noch_mal_so_gut_DE.pdf, 2019. [Online; accessed 31-March-2025].

[BB19c] I. Brand and M. Brand/Schmidt. Noch mal so gut! DE. https://www.schmidtspiele.de/details/produkt/noch-mal-so-gut-.html, 2019. [Online; accessed 31-March-2025].

[Bel57] R. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, USA, 1 edition, 1957.

[BGTG20] J. Bergdahl, C. Gordillo, K. Tollmar, and L. Gisslén. Augmenting automated game testing with deep reinforcement learning. In *2020 IEEE Conference on Games (CoG)*, pages 600–603. IEEE, 2020.

[chi20] chiamp. dqn-boardgames. https://github.com/chiamp/dqn-boardgames, 2020. GitHub, [Online; accessed 31-March-2025].

[FZAB21] N. P. Farazi, B. Zou, T. Ahamed, and L. Barua. Deep reinforcement learning in transportation research: A review. *Transportation Research Interdisciplinary Perspectives*, 11, 2021.

[HY22] A. Haydari and Y. Yilmaz. Deep reinforcement learning for intelligent transportation systems: A survey. *IEEE Transactions on Intelligent Transportation Systems*, 23(1):11–32, 2022.

[ITF+21] J. Ibarz, J. Tan, C. Finn, M. Kalakrishnan, P. Pastor, and S. Levine. How to train your robot with deep reinforcement learning: lessons we have learned. *The International Journal of Robotics Research*, 40(4–5):698–721, 2021.

[Ker15] Keras. `https://github.com/keras-team/keras`, 2015. GitHub, [Online; accessed 31-March-2025].

[KIP+18] D. Kalashnikov, A. Irpan, P. Pastor, J. Ibarz, A. Herzog, E. Jang, D. Quillen, E. Holly, M. Kalakrishnan, V. Vanhoucke, and S. Levine. Qt-opt: Scalable deep reinforcement learning for vision-based robotic manipulation. In *Proceedings of the Conference on Robot Learning (CoRL)*, pages 651–673, Zürich, Switzerland, 2018.

[Kri21] S. Krishnan. How to determine the number of layers and neurons in the hidden layer? `https://medium.com/geekculture/introduction-to-neural-network-2f8b8221fbd3`, 2021. Medium, accessed 31-March-2025.

[KST+22] B. R. Kiran, I. Sobh, V. Talpaert, P. Mannion, A. A. A. Sallab, S. Yogamani, and P. Pérez. Deep reinforcement learning for autonomous driving: A survey. *IEEE Transactions on Intelligent Transportation Systems*, 23(6):4909–4926, 2022.

[LM23] J. Lee and M. Mitici. Deep reinforcement learning for predictive aircraft maintenance using probabilistic remaining-useful-life prognostics. *Reliability Engineering & System Safety*, 230, February 2023.

[MH07] J. Marks and V. Hom. Automatic design of balanced board games. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, pages 25–30, 01 2007.

[NOST18] M. Nazari, A. Oroojlooy, L. Snyder, and M. Takác. Reinforcement learning for solving the vehicle routing problem. In *Proceedings of NeurIPS*, pages 1–15, Montréal, QC, Canada, 2018.

[Pla20] A. Plaat. *Learning to Play: Reinforcement Learning and Games*. Springer Nature, Cham, 1st ed. 2020. edition, 2020.

[PMVI24] G. Paolini, L. Moreschini, F. Veneziano, and A. Iraci. Learning to play 7 Wonders Duel without human supervision. In *2024 IEEE Conference on Games (CoG)*, pages 1–4, 2024.

[Sam59] A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):210–229, 1959.

[SB18] R. S. Sutton and A. G. Barto. *Reinforcement Learning, second edition: An Introduction*. Adaptive Computation and Machine Learning series. MIT Press, 2018.

[SBS+18] N. Sünderhauf, O. Brock, W. Scheirer, R. Hadsell, D. Fox, J. Leitner, B. Upcroft, P. Abbeel, W. Burgard, M. Milford, and P. Corke. The limits and potentials of deep learning for robotics. *International Journal of Robotics Research*, 37(4-5):405–420, 2018.

[Sew19] M. Sewak. *Deep Reinforcement Learning: Frontiers of Artificial Intelligence*. Springer Singapore, 2019.

[SHS+18] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis. A general reinforcement learning algorithm that masters Chess, Shogi, and Go through self-play. *Science*, 362(6419):1140–1144, 2018.

[SKJK20] Y. Shin, J. Kim, K. Jin, and Y. Kim. Playtesting in Match 3 game using strategic plays via reinforcement learning. *IEEE Access*, PP:1–1, 03 2020.

[Ten15] TensorFlow: Large-scale machine learning on heterogeneous systems. `https://www.tensorflow.org/`, 2015. GitHub, [Online; accessed 31-March-2025].

[WKI20] Y. Wu, M. N. A. Khalid, and H. Iida. Informatical analysis of go, part 1: Evolutionary changes of board size. In *2020 IEEE Conference on Games (CoG)*, pages 320–327, 2020.

[WPW21] L. Wang, Z. Pan, and J. Wang. A review of reinforcement learning based intelligent optimization for manufacturing scheduling. *Complex Systems Modeling and Simulation*, 1(4):257–270, December 2021.

[XCA19] K. Xenou, G. Chalkiadakis, and S. Afantenos. Deep reinforcement learning in strategic board game environments. In Marija Slavkovik, editor, *Multi-Agent Systems*, pages 233–248, Cham, 2019. Springer International Publishing.

[YBT+23] S. Yang, M. Barlow, T. Townsend, X. Liu, D. Samarasinghe, E. Lakshika, G. Moy, T. Lynar, and B. Turnbull. Reinforcement learning agents playing Ticket to Ride – a complex imperfect information board game with delayed rewards. *IEEE Access*, 11:60737–60757, 2023.

Appendix

Model	hidden layers	learning rate	discount factor	Wins
0	5	0.0001	0.5	3798
1	5	0.0001	0.6	3633
2	5	0.0001	0.7	3513
3	**5**	**0.0001**	**0.8**	**3823**
4	5	0.0001	0.9	3423
5	5	0.0001	0.95	3076
6	5	0.001	0.5	3762
7	5	0.001	0.6	3447
8	5	0.001	0.7	3511
9	5	0.001	0.8	3549
10	5	0.001	0.9	3370
11	5	0.001	0.95	2796
12	**8**	**0.0001**	**0.5**	**3907**
13	8	0.0001	0.6	3608
14	8	0.0001	0.7	3628
15	8	0.0001	0.8	3767
16	8	0.0001	0.9	3459
17	8	0.0001	0.95	3078
18	8	0.001	0.5	3560
19	8	0.001	0.6	3497
20	8	0.001	0.7	3748
21	8	0.001	0.8	3437
22	8	0.001	0.9	3340
23	8	0.001	0.95	2749
24	**10**	**0.0001**	**0.5**	**3817**
25	10	0.0001	0.6	3721
26	10	0.0001	0.7	3611
27	**10**	**0.0001**	**0.8**	**3854**
28	10	0.0001	0.9	3383
29	10	0.0001	0.95	3150

30	**10**	**0.001**	**0.5**	**3885**
31	**10**	**0.001**	**0.6**	**3938**
32	10	0.001	0.7	3395
33	10	0.001	0.8	3311
34	10	0.001	0.9	3391
35	10	0.001	0.95	2163

Table A.1: The results of the first training round of the game *"Noch mal!"* after 50,000 episodes. In the table, the best 6 models are highlighted in bold.

Model	hl	lr	df	bomb_hl	bomb_lr	row_hl	row_lr	Wins
0	10	0.0001	0.5	3	0.0001	3	0.0001	4435
1	10	0.0001	0.5	3	0.0001	3	0.001	4531
2	10	0.0001	0.5	3	0.0001	5	0.0001	4214
3	10	0.0001	0.5	3	0.0001	5	0.001	4272
4	10	0.0001	0.5	3	0.001	3	0.0001	4566
5	10	0.0001	0.5	3	0.001	3	0.001	4588
6	**10**	**0.0001**	**0.5**	**3**	**0.001**	**5**	**0.0001**	**5015**
7	10	0.0001	0.5	3	0.001	5	0.001	4366
8	10	0.0001	0.5	5	0.0001	3	0.0001	4885
9	**10**	**0.0001**	**0.5**	**5**	**0.0001**	**3**	**0.001**	**5141**
10	10	0.0001	0.5	5	0.0001	5	0.0001	4420
11	10	0.0001	0.5	5	0.0001	5	0.001	4908
12	10	0.0001	0.5	5	0.001	3	0.0001	4422
13	10	0.0001	0.5	5	0.001	3	0.001	4563
14	10	0.0001	0.5	5	0.001	5	0.0001	4679
15	10	0.0001	0.5	5	0.001	5	0.001	4943
16	10	0.0001	0.6	3	0.0001	3	0.0001	4671
17	10	0.0001	0.6	3	0.0001	3	0.001	4933
18	10	0.0001	0.6	3	0.0001	5	0.0001	4734
19	10	0.0001	0.6	3	0.0001	5	0.001	4727
20	10	0.0001	0.6	3	0.001	3	0.0001	4628
21	10	0.0001	0.6	3	0.001	3	0.001	4865
22	10	0.0001	0.6	3	0.001	5	0.0001	4118
23	10	0.0001	0.6	3	0.001	5	0.001	4624
24	10	0.0001	0.6	5	0.0001	3	0.0001	4888
25	**10**	**0.0001**	**0.6**	**5**	**0.0001**	**3**	**0.001**	**5227**
26	10	0.0001	0.6	5	0.0001	5	0.0001	4515
27	10	0.0001	0.6	5	0.0001	5	0.001	4466
28	10	0.0001	0.6	5	0.001	3	0.0001	4831
29	10	0.0001	0.6	5	0.001	3	0.001	4664

30	**10**	**0.0001**	**0.6**	**5**	**0.001**	**5**	**0.0001**	**4982**
31	10	0.0001	0.6	5	0.001	5	0.001	4514
32	10	0.0001	0.7	3	0.0001	3	0.0001	4638
33	10	0.0001	0.7	3	0.0001	3	0.001	4230
34	10	0.0001	0.7	3	0.0001	5	0.0001	4625
35	10	0.0001	0.7	3	0.0001	5	0.001	4766
36	10	0.0001	0.7	3	0.001	3	0.0001	4613
37	10	0.0001	0.7	3	0.001	3	0.001	4249
38	**10**	**0.0001**	**0.7**	**3**	**0.001**	**5**	**0.0001**	**5054**
39	10	0.0001	0.7	3	0.001	5	0.001	4834
40	10	0.0001	0.7	5	0.0001	3	0.0001	4526
41	10	0.0001	0.7	5	0.0001	3	0.001	4684
42	10	0.0001	0.7	5	0.0001	5	0.0001	4674
43	10	0.0001	0.7	5	0.0001	5	0.001	4705
44	10	0.0001	0.7	5	0.001	3	0.0001	4534
45	**10**	**0.0001**	**0.7**	**5**	**0.001**	**3**	**0.001**	**5289**
46	10	0.0001	0.7	5	0.001	5	0.0001	4523
47	10	0.0001	0.7	5	0.001	5	0.001	4880

Table A.2: The results of the first training round of the game *"Noch mal so gut!"* after 150,000 episodes. In the table, the best 6 models are highlighted in bold.