Master Thesis

in the field of

Computer Science

Application of Modern Reinforcement Learning Algorithms to the Card Game Cego

supervised by:	Prof. Dr. Maja Temerinac-Ott
cosupervised by:	Prof. Jirka Dell´Oro-Friedl
submitted on:	31.08.2022
submitted by:	Philipp Oeschger
	matriculation number: 268388
	Bregstraße 90
	78120 Furtwangen im Schwarzwald
	philipp.oeschger@hs-furtwangen.de

Abstract

Cego is a German card game that hasn't been addressed in research on reinforcement learning. The purpose of this thesis is to train an AI using artificial neural network-supported reinforcement learning algorithms to learn the card game Cego. Eventually, this AI will be implemented in the redevelopment of Cego-Online, the web-based game. The training environment of the game is implemented within the RLCard framework and divided into sub-games. Suitable action and game state encoding schemes are presented. The game environment is analyzed by simulating specific game aspects and defining benchmarks. Models for a single sub-problem of the card game are trained using the algorithms DQN, NFSP, and DMC. Comparing the results, the DMC model achieves the highest level of play. Additional models are trained on the Cego sub-problems with DMC. All models surpass their benchmarks. To ensure the accessibility of the trained models, an application programming interface is designed.

Cego ist ein deutsches Kartenspiel, das in der Forschung des bestärkenden Lernens bisher unerforscht ist. Das Ziel dieser Arbeit ist es, eine KI zu trainieren, die mithilfe von Algorithmen, des bestärkenden Lernens und künstlicher neuronaler Netze das Kartenspiel Cego erlernt. Diese KI soll schließlich in der Neuentwicklung des webbasierten Spiels Cego-Online eingesetzt werden. Die Trainingsumgebung für das Spiel ist implementiert im Framework RLCard und separiert in Subspiele. Angemessene Aktions- und Spielzustands-Kodierungsschemata werden vorgestellt. Die Spielumgebung wird analysiert, indem bestimmte Spielaspekte simuliert und Richtwerte definiert werden. Modelle werden, für ein Subprobleme des Kartenspiels, mit den Algorithmen DQN, NFSP und DMC trainiert. Die Ergebnisse werden anschließend miteinander verglichen. Im Vergleich erzielt das Modell, trainiert mit DMC, das höchste Spielniveau. Zusätzliche Modelle werden für die Cego-Subprobleme mit DMC trainiert. Alle Modelle übertreffen ihre Richtwerte. Schlussendlich wird eine Schnittstelle konzipiert, welche den Zugang zu den trainierten Modellen gewährleistet.

Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

AI	artificial intelligence
ANN	artificial neural network
API	application programming interface
APPG	average points per game
CFR	counterfactual regret minimization
CRUD	create, read, update, and delete
DMC	deep Monte Carlo
DNN	deep neural network
DP	dynamic programming
DQN	deep Q-network
FSP	fictitious self play
HPO	hyperparameter pptimization
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
MC	Monte Carlo
MDP	Markov decision process
ML	machine learning
MLP	multilayer perceptron
MRL	multiagent reinforcement learning
MSE	mean squared error
MVT	Model View Template
NFSP	neural fictitious self-play
ONNX	Open Neural Network Exchange
REST	representational state transfer
RL	reinforcement learning
SL	supervised learning
TD	temporal-difference
TIOT	training improvement over time
TLU	threshold logic unit

URI	Uniform Resource Identifier
URL	Uniform Resource Locator
WP	winning percentage
XFP	extensive form fictitious play

1 Introduction

1.1 Topic

If the question were asked which game embodies the mentality of the German region *Baden* the most, one might argue for the game *Cego* [1]. The history of the game is long and varied. In many ways, the game is unique compared to other games that may be played in Germany, such as *Skat*. One specific characteristic that personifies the game and the *badner* culture is a certain laid-back approach towards the enforcement of rules. In the native dialect, *Hinterfotzigkeit* is sometimes used as a description of an underhanded player who, for example, likes to play mind games and Cego offers game mechanics to these players. Cego also lacks a definitive rule set because of regional differences, which further complicates the generalization of the game.

Many of the named attributes make it difficult to transfer the game experience into digital form. Nevertheless, in 2013, *Cego Online* [2] was released and has since then grown into a small but loyal community of players.

A relaunch is planned due to many reasons, including the end of support for *Adobe Flash Player*[3] that Cego Online relied on. *Cego Online 2.0* [4] is the name of a project that tries to improve on the shortcomings of the first version. One of the improving points is artificial intelligence (AI). The idea is to improve on the AI utilizing modern algorithms and findings in reinforcement learning (RL). In addition, it has yet to be determined whether an AI based on RL algorithms can learn Cego and what level of play it can achieve in this domain.

1.2 Problem

This work aims to create an environment that allows the training of the card game Cego with RL algorithms. It also aims to train one or more models that maximize the level of play. Eventually, the models should be able to learn sub-problems of the card game. Moreover, a system should be developed that makes these models available for use in the game environment of Cego Online 2.0.

1.3 Methodology

A framework is selected for the purpose of ensuring a problem-free and goal-oriented training process. Sub-problems are identified and implemented within the framework to create a base for the training of AI models. With the intention of defining reference values that can be used as an orientation for the evaluation of RL models, the game environment is analyzed beforehand to identify benchmarks.

To find an algorithm that maximizes the level of play in Cego, multiple algorithms are compared. For this task, an evaluation process is conceptualized. To ensure the best possible results for the applied algorithms, the hyperparameters of the deployed algorithms are optimized in advance. After defining the hyperparameters, models are trained for the algorithms based on one specific sub-problem at first. Afterward, the evaluation is conducted to identify the algorithm and parameters that provide the best performing results.

Subsequently, models for the sub-problems that haven't been addressed yet are trained with the identified algorithm parameters. The models are evaluated by conducting experiments and comparing the results against the defined benchmarks. As a final step, a solution is designed and implemented that makes the trained sub-models available for the game engine of Cego Online 2.0.

1.4 Related Work

Many advances have been made in the learning of perfect information games. A long pursued milestone was achieved in 2016 by the AI *AlphaGo* [5]. This was first AI able to beat an acclaimed professional human player by a comfortable margin in the Chinese strategy game *Go*. This achievement was later outperformed by the follow-up project *AlphaGo Zero* [6] that, instead of relying on human player data, supervised learning (SL), and RL, achieved a sub-human level of play only using RL techniques and self-play. This approach has shown to be transferable to games like Chess and Shogi [7].

On the other side of perfect information games lie the imperfect information games, in which category many card games fall, where early research has been made in the game of *Bridge* [8]. Similar to Cego, in Bridge, players have to win card tricks. This approach is based on Monte Carlo (MC) and perfect information states. Milestones in other imperfect information games have only been achieved recently. The solving of *Heads-up limit Texas Hold'em poker* [9] could be the first cornerstone for other advances in imperfect information games. The implemented algorithm counterfactual regret minimization (CFR)$^+$ was able to solve the multiple orders of magnitude large game space of Texas Hold'em poker. Contrary to other games discussed in this section, poker does not have many similarities to Cego, and therefore, this milestone might provide only minor insight for this research.

Cego's cultural and historical background has many facets that were discussed in detail by Blümle [1]. He described many attributes and trades that personify the game to this day. Some aspects make it particularly challenging for an AI to learn and master the full domain of the problem. Specifically, the branching into multiple sub-games may have varying implications on the way players act, but certainly have a large impact on the complexity of the game. Solving this problem is something that hasn't been addressed for Cego in any earlier scientific work. This thesis provides the first example of an RL-based AI within the game Cego.

In some respects, Cego can be compared to the Chinese card game *Doudizuh* [10]. Similar to Cego, the game is played with more than two players and with two unevenly matched teams that play against each other where players might have to cooperate. In addition, both games have to address imperfect information and large game states. The algorithm deep Monte Carlo (DMC) has been proven to be successful in the game. In its implementation, *DouZero* outperformed any of the existing AI of the game. But DMC hasn't been tested on other domains. There are also a few differences between the two games, Cego and Doudizuh, that make it unclear whether the performance of DMC can be translated to Cego. The effect of translating DMC to this works game domain has to be observed.

Another game that may in comparison provide more similarities to Cego than Doudizuh is the card game *Skat*. Both games are, for example, trick play games and have a game component of blind cards laid down in the middle of the table. But contrary to Cego, Skat's strategies and mathematical backgrounds have been well researched [11]–[14].

In terms of AI development, Skat provides interesting work and findings. Kupferschmid et al. [15] applied MC simulation and alpha-beta search. Additional techniques such as move ordering, quasi-symmetry reduction, and adversarial heuristics were implemented to improve tree searching algorithms. The result is a program determining the game-theoretical value of a Skat hand in just about 10 milliseconds. In 2013 Furtak [16], introduced symmetry-based search extensions for solving perfect information game states for the implementation in an MC-based computer player. The resulting AI was able to achieve high performance. Both previously referenced works focused on perfect information game states. But from the perspective of a fair player that plays against human opponents, an approach based on imperfect information states may provide a better game experience.

Edelkamp [17] discussed an alternative approach to creating a Skat AI. His research used information from a large sample size of games, a combination of expert rules, the aggregation of winning probabilities and in addition fast tree exploration to accomplish a high level of play in various sub-problems of Skat. The resulting AI performed better than humans in some areas, but was disadvantaged in other disciplines of the game. Even though similar approaches might be feasible for Cego, previous findings [10] indicate that the performance of Deep RL techniques could have been underestimated in domains of card games with imperfect information.

Besides game-specific approaches, the development of frameworks that generalize RL has taken many steps in a direction that makes RL approachable for a wider mass of researchers [18]–[21]. Most of these frameworks are in some way compatible with *OpenAI Gym* [22] a library that allows for the comparison of algorithms. In addition, it standardizes environment and algorithm communication. In 2020, *OpenSpiel* [23] was released, and it combines many established as well as state-of-the-art RL algorithms for comparisons in various games. A framework that is strictly focused on the specific task of card games is *RLCard* [24]. Similar to OpenAI Gym, the library offers a possibility to compare models with each other. Frameworks, such as OpenSpiel or RLCard, may present an even playing field for testing and comparing algorithms for Cego.

1.5 Overview of the Structure

Before discussing the details of this work, Chapter ***2, Background***, examines the base of knowledge that serves as the foundation for the following chapters of this thesis. The main subject of this chapter includes the rules of Cego, core concepts of game theory, the basics of machine learning (ML), key concepts and algorithms of RL, as well as the topics MC simulation and representational state transfer (REST) application programming interface (API).

The following Chapter ***3, Analysis of the RL Environment***, takes a closer look at the RL framework that was utilized for the implementation of the game environment and the later model training. This chapter explains the reasons for the selection of the specific RL environment and outlines the main attributes that set it apart from other contemporary frameworks. In addition, this chapter discusses the algorithms provided by this framework and their characteristics and features in detail.

In the subsequent Chapter ***4, Game Implementation into RL Environment***, details about the game implementation, serving as the base for the training, are provided. More importantly, this chapter discusses certain design decisions that were implemented to improve model training. Subjects are, e.g., the action and state encoding of the game environment, the reward system, and the division of sub-problems.

Chapter ***5, Analysis of the Game Environment***, describes experiments and analysis in an effort to establish a better understanding of the game behavior. This chapter serves as a foundation for the subsequent evaluation and puts the results into context.

After the analysis of the game environment, follows the process of ***hyperparameter pptimization (HPO)*** in Chapter ***6***. This chapter discusses the scope and methodology of HPO employed, to ensure a proper and fair comparison of the RL algorithms.

The next topic discussed in Chapter ***7*** is the ***Training and Evaluation*** process. This chapter provides a quantitative comparison of the discussed algorithms based on one specified sub-problem of the game Cego. Finally, based on this quantitative comparison, the best performing algorithm and parameters are identified. Afterward, the further training process of models for the remaining sub-problems is examined.

In the penultimate Chapter ***8***, the ***implementation details of the API***, realized to make the AI models available to the game engine, are provided. The focus was to provide a clear and elegant way through which the knowledge learned by the RL models can be used by an artificial opponent within the game engine.

Ultimately, Chapter ***9*** provides a ***retrospective of the work***. The contents of this chapter include a summary of the findings and results. In addition, there is outlook on possible future work, and the classification of the findings in the broader context of research.

2 Background

2.1 Game Theory

2.1.1 Two-Player Zero-Sum Game

A zero-sum game is strictly competitive [25]. The rewards of all players combined always add up to zero. Within a two-player game, this means that if one player gets points, the other player loses the same number of points.

2.1.2 Information in Games

Games can be differentiated in terms of how much information the players have about the game state [25], [26]. It is possible to classify games into *perfect information games* and *imperfect information games*.

Within perfect information games, all players have flawless knowledge about all aspects of the current and previous game states [25], [26]. An example of a perfect information game is chess.

The counterpart to the perfect information game is the imperfect information game [25], [26]. The players do not have all the information about the current and previous game states. For instance, a game that has elements based on chance is an imperfect information game. Card Games are a good example of imperfect information games since the players do not know what cards have been dealt to the other players.

2.1.3 Sequential Games

In sequential games players choose actions one after the other [27]. Since there is an order of actions, no simultaneous moves are possible.

2.1.4 Nash Equilibrium

Nash equilibrium describes a state where every player selects rationally the most advantageous decision against the opponent. Therefore, every player follows a static optimal strategy [26] [27].

2.1.5 Representation of Games

There are two common formalizations to represent games. The *normal form* and the *extensive form* [26]–[28].

The normal form, also called the *strategic form*, is a game representation that can be described as a matrix including the number of players N, the strategy space S, and the utility-function u [26]–[28]. The result is a tuple (N, S, u).

In the Normal Form, all players must decide on a strategy simultaneously without knowing what the other players are doing [27]. However, many games are sequential, so players can react to actions taken by other players. The extensive form allows the representation of games, including their sequential aspects. Games in extensive form can be visualized in *game trees* that show when players take action and what their information state is at a point in time [26]–[28].

2.2 ML

ML describes algorithms that use data to learn and improve a specific task without further programming or intervention [29]. Algorithms for ML can help manage and understand simple or more complex subjects and are able to adjust to changing environments. These reasons make ML very attractive for most of the recent AI research.

One major aspect of Machine Learning is the training stage [29]. Depending on the subcategory, a training set is required to feed the learning algorithm. Ideally, the result is an instance that learned the task.

2.2.1 ANN

Artificial neural networks (ANNs) are systems used for function approximation in ML [29]. They imitate biological neurons and are often used for solving tasks of varying complexity. In their simplest form, Artificial Neurons are computational components that have weights attached to one or more inputs and a step function. This artificial neuron, also known as **threshold logic unit (TLU)** (see Figure 1), produces an output. An example of a step function is the Heaviside function, which returns *true* if a certain threshold is reached and *false* if the opposite is the case. A single neuron, for example, is sufficient to solve a linear binary classification problem.

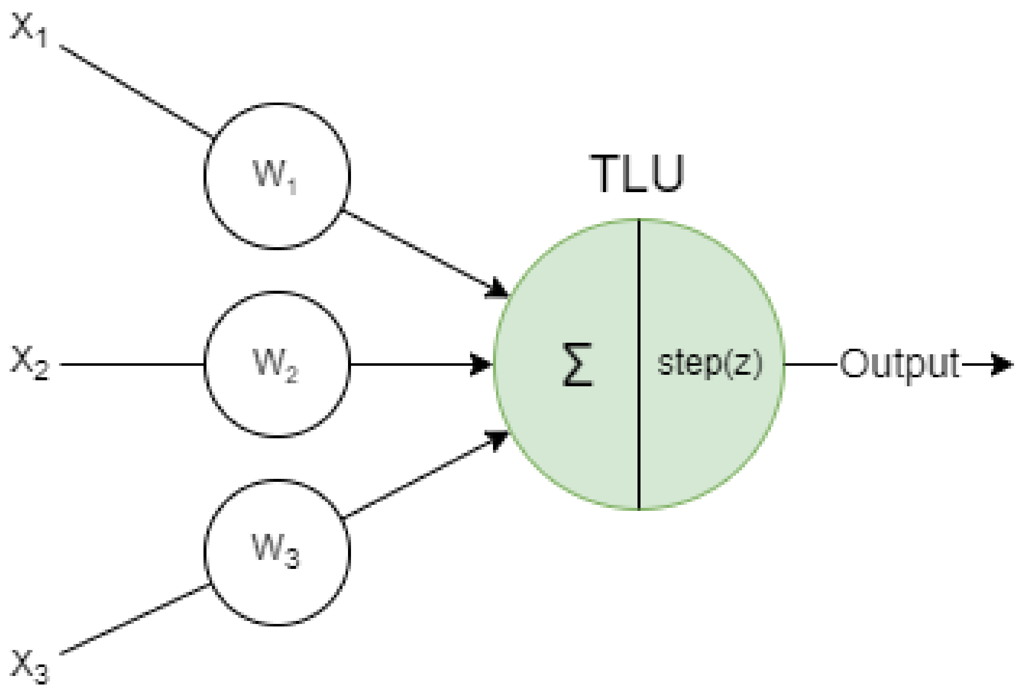


Figure 1: Example of a TLU
based on [29]

The neurons of one layer are usually extended by adding a bias neuron [29]. The resulting equation of a single layer can be described as follows:

$$h_{W,b}(X) = \Phi(XW + b) \tag{1}$$

X describes the matrix of input features [29]. W contains all the weights from connecting all the input neurons with the neurons within the layer. Each neuron is assigned a bias vector. After the alteration, the step function is now called an *activation function*, and it is assigned the symbol Φ.

ANNs get trained by considering one training instance at a time [29]. If an output neuron produced a wrong prediction, the weights from the input get adjusted in a direction that leads to the correct answer. The rate by which the weights are adjusted is called the *learning rate*.

One layer ANNs are called *perceptrons* [29]. Neurons can also be grouped and stacked into multiple layers. The result is a **multilayer perceptron (MLP)** (see Figure 2). An MLP consists of an input layer, one or more hidden layers (layers of TLUs), and an output layer. Every hidden layers has a bias neuron, and each of its neurons is connected to each neuron of the next layer.

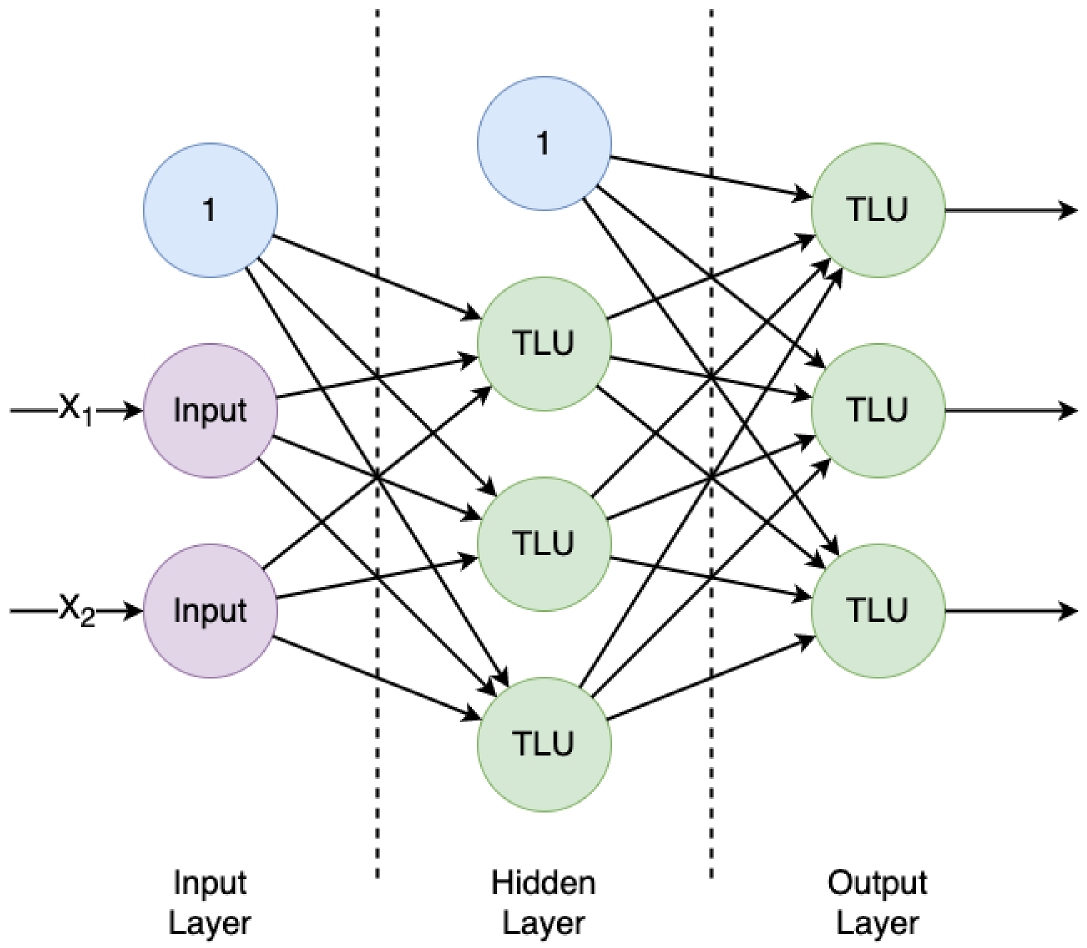


Figure 2: MLP
based on [29]

Deep neural networks (DNNs) are MLPs with multiple hidden layers [30].

2.2.2 Gradient Descent

Gradient Decent is an iterative optimization method, commonly used in machine learning algorithms [29], [30]. Suppose there is a function $E(\omega)$ of a vector of variables called a gradient vector that consists of the following partial derivations [29], [30]:

$$\nabla_\theta E = \left[\frac{\partial E}{\partial \theta_1}, \frac{\partial E}{\partial \theta_2}, \ldots, \frac{\partial E}{\partial \theta_d}\right]^T \tag{2}$$

At first, θ gets initialized randomly. Each iteration θ gets adjusted in the opposite direction $\hat{\theta}$ [29], [30]:

$$\Delta\theta_i = -\eta \frac{\partial E}{\partial \theta_i}, \forall i \tag{3}$$

The factor η is the *learning rate* that determines at which rate the direction is adjusted. Machine learning algorithms use this technique to minimize cost functions [29]. Setting the learning rate too high results in overshooting the minima. Set the learning rate too low, and it may take considerably longer to reach the minima. Depending on the selection of the cost function, there may also be a risk of hitting platos or local minima, which prevents the algorithm from converging. Therefore, the choice of the learning rate and the cost function is crucial.

2.2.3 Backpropagation

Backpropagation is the algorithm used by ANNs to learn [29]. The following steps describe the backpropagation process [29]:

1. Group training data in mini-batches and process through the full training set in batches. One full pass is called an epoch.
2. The mini-batch is passed through the network from front to back, saving all the intermediate results on the way through each layer.
3. Calculate the error that describes the difference between the calculated output and the desired output with a *loss function.*
4. Compute how much each connection contributes toward the error by propagating that error from back to front, creating an error gradient.
5. Perform a gradient descent step to adjust the weights using the error gradient.

Repeating these steps over multiple episodes allows ANNs to minimize the specified loss function for learning specific tasks, resulting in a model that can solve the problem [29].

2.3 RL

RL is a subdomain of ML [31]. Its central problem is to map certain situations to actions in a way that maximizes a reward value. Furthermore, RL is a collection of algorithms, methods, and approaches that tackle this task. Part of solving RL-Problems is the weighting of exploration and exploitation. Each problem domain starts with an unknown environment regarding how to map actions and situations to rewards. The potential outcomes of the environment must be explored, and already discovered experiences must be weighed against new ones. Actions that lead to rewards at later time steps should also be accounted for. Weighting these aspects defines RL.

2.3.1 Key Components of RL

An RL system [31], consists of:

- A *policy*,
- A *reward*,
- A *value function*, and
- A potential *model* of the environment.

The *policy*, often denoted as π, is the mapping of environment states to specific actions an agent can take within the environment [31], [32]. Consequentially, the policy defines the behavior of the agent. Often, policies are stochastic.

The agent receives a numeric value from the environment at each time step, which is called the *reward* [31], [32]. The main goal of the agent is to maximize the accumulated reward. Therefore, the agent can act goal-oriented.

Whereas the reward defines the direct reward an agent gets, the *value function* also takes into account the expected future rewards [31], [32]. The value of a state defines what reward the agent can expect, depending on how likely it is for the agent to hit a state with high reward values.

A *model* imitates the characteristics of a real environment [31], [32]. It can be used for pre-planning the behavior of certain actions before actually performing them. RL Methods that use an underlying model for planning are called *model-based*. Their counterparts are *model-free* methods that rely on trial and error learning without the need for a model for learning.

2.3.2 Discounting Rewards

When the value of a state is the expected return [31], [32]. Then the return G of a current state could be the total reward from this current state until the final state. However, if there is no final state, the return G is infinite. Therefore, some value functions choose an approach where future rewards are discounted. In this case, the calculation of G can be described as follows:

$$G \doteq \sum_{k=0}^{T} \gamma^k r_k \tag{4}$$

The parameter γ is called the discount factor and has the constraints $0 \geq \gamma \geq 1$ [31], [32]. In case the terminal state T is infinite, γ decreases the value of future rewards and ensures that G approaches a finite number.

2.3.3 Exploration vs Exploitation

Every chosen action has a value (see Subsection 2.3.1) attached to it [31]. The selection of the action is a maximization problem. Selecting an action greedily can be described in the following way:

$$A_t \doteq \arg\max_a Q_t(a) \tag{5}$$

In this case, the policy is exploited [31]. Each time step selects the action that has the highest value. The problem with this approach is that no new experiences can be found, therefore the policy can't be improved.

An $\epsilon - greedy$ policy solves this problem by introducing a probability ϵ, which defines the chance of exploring a random action and evaluating new experiences instead of exploiting current experiences [31], [32]. The adjustment of ϵ allows balancing exploration and exploitation.

2.3.4 MDP

The *Markov decision process (MDP)* is a mathematical formalism for the RL problem and defines the following core concepts [31]–[33]:

- ***Agent***: The learning actor.
- ***Environment***: The object that the agent interacts with.
- ***State***: The representation of the environment the agent is observing at a certain point in time.
- ***Action***: The measure that an agent can take.
- ***Reward***: The numerical value that an agent receives.

The relation between the concepts is described in the agent-environment interface (see Figure 3) [31]. The agent observes the state of the environment at a certain time step t. The time step is part of a sequence of discrete values, $t = 0, 1, 2, 3. \dots$ The agent selects an action, A_t which is in $A(s)$. An agent who performs this action will receive a reward R_{t+1} one time step later. Additionally, the *state* will be updated to S_{t+1}.

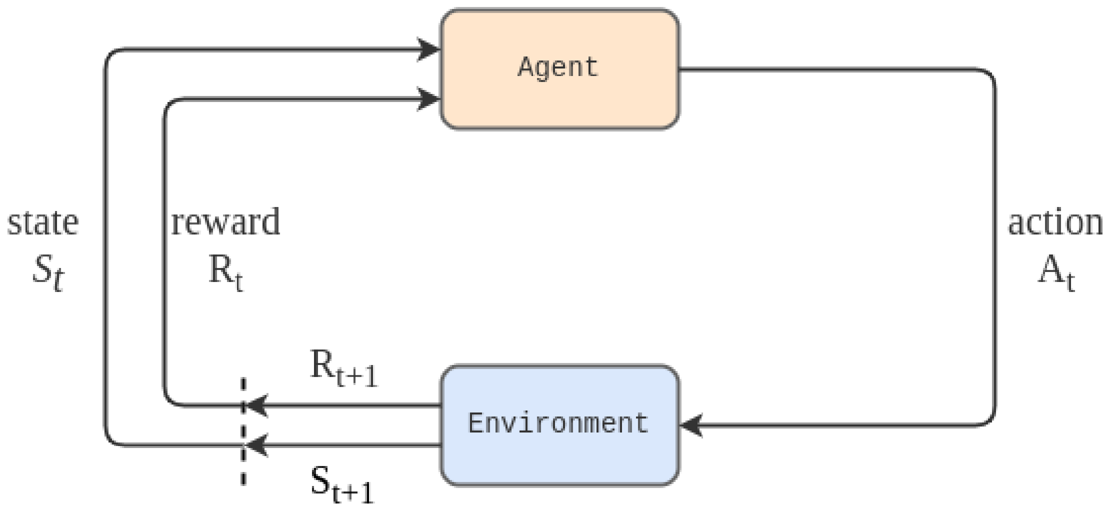


Figure 3: Relations in an MDP System
based on [31]

In many problems, the MDP is finite, which means the *S, A, and R* have a finite amount of elements [31], [32]. Here, well-defined probabilities are attached to R_t and S_t, related to the chosen *action* by the *agent* and the previous *state*. The probability of landing in the new *state* $s' \in S$ and receiving the *reward* $r \in R$ can be described as follows [31]:

$$p(s', r|s, a) \doteq Pr\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\}. \tag{6}$$

The function p describes the dynamic and, therefore, the transition model of the MDP [31], [32].

2.4 Algorithms of RL

2.4.1 MC Methods

MC methods can be applied to find the most optimal policies for RL-Problems. The core action is the averaging of sample returns [31], [32]. MC methods are extensively researched and well suited to certain situations.

In RL, MC Methods sample entire episodes of the environment and then average the reward [31], [32]. Therefore, it is advisable to have game episodes that always terminate. There are two main variants, *first-visit* MC and *every-visit* MC. The difference in those variants is how they handle states that are visited multiple times in a single episode. In first-visit MC, only the reward of the first visit to the state is considered for calculating the average return. In every-visit MC, the return of every state visit is considered for the average return calculation.

The algorithm for first-visit MC is shown in Algorithm 1 [31]. A policy π serves as an input to be evaluated. Then, the value array is initialized with random values and the returns are initialized as an empty list. At each training iteration, a full episode is generated based on the policy, and the reward of this episode is initialized with 0. Starting with the last state and ending with the first state, each state reward is observed and added to the returns of the state. The value of the state is defined as the average reward over all returns. In first-visit MC, the state return will only be considered if the state hasn't been visited before in this episode.

```
Input: policy $\pi$ to be evaluated
Initialize $V(s) \in \mathbb{R}$, arbitrarily, for all $s \in S$
Initialize $Returns(s) \leftarrow$ an empty list, for all $s \in S$
for iteration = 1, 2, ... do
    Generate an episode following $\pi$
    $G \leftarrow 0$
    for t= T-1, T-2, ..., 0 do
        $G \leftarrow G + R_{t+1}$
        if not $S_t \in \{S_0, S_1, ..., S_{t-1}\}$ then
            Append $G$ to $Returns(S_t)$
            $V(S_t) \leftarrow average(Returns(S_t))$
        end if
    end for
end for
```

Algorithm 1: First Visit MC
based on [31], [32]

2.4.2 TD Learning

Before discussing the Algorithm Q-Learning in detail, the underlying idea of Q-Learning is discussed first, *temporal-difference (TD) learning* [31], [32]. It combines the idea extracted from MC techniques of sampling experiences for learning without prior knowledge of the environment's behavior with the *dynamic programming (DP)*[1] idea of *bootstrapping*[2].

One major difference and advantage between TD over MC Methods is that the total return of an entire episode is not required as knowledge [31], [32]. Additionally, a model of the system and therefore of all probability transitions is not required either, unlike in DP. The online[3] learning approach can be described by the following function [31]:

$$V(S_t) \leftarrow V(S_t) + \alpha \left[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\right] \tag{7}$$

[1] DP describes a set of approaches that use a perfect model of the system in order to compute optimal solutions.
[2] Bootstrapping describes the method of updating estimates based on other estimates [31].
[3] online learning describes the process of updating parameters after each instance [30]

The value of a state can be calculated by observing the current reward, and the discounted reward prediction from the neighbor state, with γ being the discount factor, and α the rate at which the state values are updated. This process can be repeated iteratively for states until the value-table converges to optimal values.

2.4.3 Q-Learning

Q-Learning is a model-free RL-Algorithm that uses the idea of TD to find an optimal policy [31], [32], [34], [35]. For Q-Learning, a specific action-state-pair is evaluated by the received reward or penalty combined with the estimated value of the resulting next state. By repeating this process for all states and actions, the agent creates a map that contains an estimate of all state-action values. This map can be used to select the most advantageous outcome. This approach to TD learning works independently of the policy in place for deciding on which action to take. For the updating of value estimates, Q-Learning uses the following rule [31]:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right] \qquad (8)$$

Here, γ defines the discount factor and α is the learning rate.

Algorithm 2 shows the Q-Learning algorithm [31], [32], [35]. After deciding on a policy, which could for example be $\epsilon - \textit{greedy}$, the mappings for the q-values of the state-action pairs are initialized with 0. This lookup table is also called Q-table[1]. Then, a certain number of episodes are looped. For each episode, the environment is reset and a starting state S is selected. A new action is sampled by the policy and the resulting follow-up state is selected until an end state has been reached. For each action one receives a resulting reward R, and each action leads to a new state S'. The second to last step of this inner loop is to update the Q-Value with the most recent observation. In the end, the state S gets updated to the state S'.

[1] Depending on the implementation, the Q-table can sometimes include only the value for each state without consideration of the action [32]

```
Input: a policy that uses the action-value function,
    $\pi(a|s, Q(s,a))$
Initialize $Q(s,a) \leftarrow 0, forall s \in S, a \in A(s)$
for iteration = 1, 2, ... do
    Initialize environment to provide $S$
    while $s$ is not terminal do
        Choose $A$ from $S$ using $\pi$, breaking ties randomly
        Take action $A$ and observe $R$, $S'$
        $Q(S,A) \leftarrow Q(S,A) + \alpha\,[R + \gamma \max_a Q(S',a) - Q(S,A)]$
        $S \leftarrow S'$
    end while
end for
```

Algorithm 2: Q-Learning
based on [31], [32]

2.4.4 FSP

This model-free algorithm is based on *fictitious play*[36]. An algorithm based on *self-play* that focuses on learning the opponent's average policy and predicting the most advantageous move. This algorithm can converge to nash equilibrium in certain types of games, such as two-player zero-sum games.

Fictitious games are typically defined in their normal form. However, for games that can be described in extensive form, this representation is exponentially less efficient in terms of computation. Heinrich et al. addressed this problem with *extensive form fictitious play (XFP)* [37], which allows the updating of fictitious player strategies in extensive form and, therefore, with linear time and space complexity relative to an iteration. At each iteration, the XFP algorithm calculates the *best response* based on the opponent's average strategies and then updates the average strategy on the basis of this best response. Meanwhile, the algorithm keeps the property of converging to Nash equilibrium in games with the fictitious play property.

Even though XFP lowers the computation compared to fictitious play, it can still be computationally expensive for high-dimensional games. In addition, Heinrich et al. introduced *fictitious self play (FSP)* [37], an approach to approximating XFP with machine learning techniques. The best response calculation is implemented as an RL algorithm that learns by playing against the average strategies of other players. The average strategy is realized as an SL algorithm that approximates the player's transition model to realize the average strategy of the player.

2.4.5 MRL

In some RL algorithms, such as Q-Learning [34], a single agent is considered [31], [32], [38]. Other agents either do not exist or are seen as part of the environment. This is a naive approach to an environment where more than one agent is involved because these agents can alter the environment state as well.

Algorithms such as FSP [37] take multiple actors into account. Agents in this context must adapt and react to other agents' decisions [28], [38]. Depending on the goals the agent seeks, they may be able to compete or cooperate. Algorithms focusing on this problem are called *multiagent reinforcement learning (MRL)* algorithms.

2.5 Cego

Cego is played with *Tarock* cards [1]. The name Tarock comes from one of the card suits in the game, called *tarocco* in Italian. In German, they are called *Trümpfe*, which translates to *trumps*. For this thesis, English terms are used.

A crucial role in the game is played by the *blind cards* (Ger. *Blinden*) where Cego gets its name from [1]. The word Cego most likely has its origin in the Latin word *caecus*, which later resulted in the Spanish word *ciego* and the Portuguese word *cego*. The blind cards are a particular characteristic that sets Cego apart from other Tarock card games. Cego also provides its own terminology, which can examined in Appendix A.

The rule set of the game varies heavily across regions [1], [39]. This work focuses on the rules of the Cego Online [2] which is (except for minor changes) based on the unified rules set of the *Schwarzwaldmeisterschaft* [40]

2.5.1 Fundamentals

Cego is played by four players [39], [41], [42]. The objective of each player is to receive the most points. At the start, the players decide on the team constellations in the bidding phase, which also decides which of the numerous sub-games of Cego is played. In those rounds, the players have the opportunity to increase the risk and potentially to receive more points. In most sub-games, one player competes against three other players.

2.5.2 Card Set

The 54 cards can be divided into five suits:

- 22 trump cards,
- 8 cards for each of the following suit,
  - *clubs* (Ger. *Kreuz*),
  - *spades* (Ger. *Pik*),
  - *hearts* (Ger. *Herz*),
  - *diamonds* (Ger. *Karo*).

A visual overview of the cards is provided in Appendix B. When examining the trump cards closer, all but one are numbered 1–21. The one unnumbered card, known as the *Gstieß*, is the highest valued trump [2], [39], [42].

The suit cards can further be subdivided into *pictured cards* and *empty cards*. There are four for each of the suits: clubs, spades, hearts, and diamonds. *King* (Ger. *König*), *Queen* (Ger. *Dame*), *Cavalier* (Ger. *Reiter*), and *Jack* (Ger. *Bube*) are the symbols of the pictured cards. The four empty cards of the red suits are counted from one to four and the black suits are counted from seven to ten. This counting system is based on the number of symbols visible on each card.

2.5.3 Ranking of Cards

One of the main objectives (in most cases) is to win tricks with the dealt cards until no cards are left in the player's hand and a winner can be decided [39], [41], [42]. For this use case, the cards have clear ranks in the following descending order:

1. Gstieß,
2. Trump cards from 21 to 1 descending,
3. King,
4. Queen,

5. Cavalier,
6. Jack,
7. Empty cards,
    - Black cards from 10 to 7 *descending*, or
    - Red cards from 1 to 4 *ascending.*

2.5.4 Winning Tricks

When a sub-game has been decided on, the starting player, who placed the last bid, plays a card of their choice from their hand [39], [41]. After that, the next player in order plays a card until every player has played a card. The play direction is counterclockwise. After the first player has played a card, the other players must follow suit (Ger. *Bekennen*). If the player cannot follow suit, the player must play a trump card. If a player has no trump card in their hand, they can play any other card of the hand cards left. In the last case, the played card has no value.

The player who played the highest ranked card (see Subsection 2.5.3) in the trick wins it [39], [41]. The starting player in the next trick round is the player who won the last trick round. Rounds are played until no cards are left to play or a sub-game specific winning or losing condition has occurred.

2.5.5 Card Values

The player who wins the trick receives the value of the cards as points [41], [42]. The card values are as follows [2]:

- Gstieß, *21-trump*, *1-trump*, and King each 4.5, points,
- Queen: 3.5 points,
- Cavalier: 2.5 points,
- Jack: 1.5 points,
- All other cards: 0.5 points.

2.5.6 Game Setup

At the start of the game, ten cards are laid upside down in the middle of the desk [39], [41], [42] (see Figure 4). These are the blind cards. The remaining cards are dealt to the players. Each player gets eleven cards.

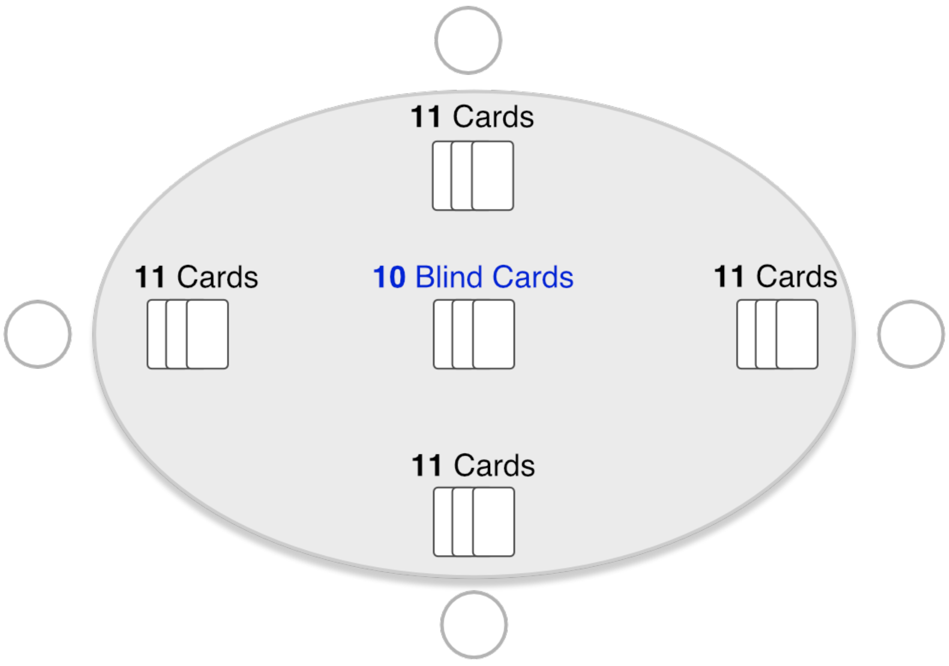


Figure 4: Cego Game Setup

2.5.7 Normal Sub-Games

Normal sub-games can be defined as game variants where one player plays against three other players to receive the highest number of points [39], [42]. It can be differentiated between two normal game variants:

- ***Solo***: The single-player must play with the cards in their hand. The blind cards stay unknown and are added to the player's point score at the end of the round.

- ***Standard-Cego***: The single-player must play with the blind cards. The remaining cards, called *Legage*, are added to the player's point score at the end of the round. Usually, this game mode is called *Cego*, the same as the card game itself, but for the sake of preventing confusion, this work names this game mode differently.

The sub-game Standard-Cego can be played in different modes, which can be bid for in the bidding phase [2], [39], [42].

- ***Cego***: Two cards are kept on the hand deck. The blind cards are picked up. One card of the deck must be laid aside to the Legage.
- ***Halbe***: One card must be played. One card is kept on the hand deck. One card must be laid aside to the Legage. After that, the player can exchange the laid out card for another card with the same color. The first played card can't win a trick.
- ***Eine***: One card is kept on the hand deck.
- ***Eine Leere***: One card must be played at the start and can't win a trick.
- ***Zwei Leere***: Two cards must be played at the start and can't win tricks. The lowest trump from the blind cards must be added to the Legage. In both cases, the single-player is the starting player.
- ***Zwei Verschiedene***: The difference to **Zwei Leere** is that the cards must be of a different suit and the highest ranked trump must be discarded toward the Legage.
- ***Kleiner Mann***: Differs from **Eine Leere** in that the played card is the card*1-trump* and can not be exchanged for another card.

Both Standard-Cego and Solo allow scoring up to 79 points [2], [42]. Therefore, the single-player must score at least 40 points to win.

2.5.8 Special Sub-Games

In some cases, a sub-game may be played outside the standard games [39], [41], [42]. The sub-games are shown in Table 1.

Sub-Game	Play Style	Condition
Ultimo	The player who declares the mode must win the last trick with the *1-trump* card.	Can be declared in the first bidding phase when nobody has declared Solo yet.
Piccolo	The player who declares the mode must win exactly one trick.	Can be declared right after the first bidding phase when nobody has called Solo yet.
Bettel	The player who declared the mode wins when they win no tricks.	Can be declared right after the first bidding phase when nobody has declared Solo yet.
Räuber	All players play against each other. The blind cards are out of the game. The Loser is the player who receives the most points.	Can only be called after the second bidding round when everyone passed and the Vorhand would have to play Cego by default.

Table 1: Sub-Games in Cego

2.5.9 Bidding Phases

After the setup, the bidding phases start [39], [41], [42]. The bidding can be divided into two phases. In the first phase, it is decided whether someone wants to play with their hand cards. Bidding begins with the *Vorhand*[1]. They either say *Fort* (Eng. *gone*) if they don't want to play with their hand cards, or they say *Ultimo* or *Solo* if they want to play with their cards. If a player says *Solo*, the other players have the chance to say *Gegensolo*. In this case, the second bidding phase starts. All players except the player who called *Solo* can bid to play with the blind cards. If no one calls *Gegensolo*, the sub-game Solo is played.

An addition implemented in Cego Online is a separate phase between the first and second bidding phase. This phase is where players have the chance to call either the game mode *Bettel* or *Piccolo* [2].

When none of the players want to play *Solo*, *Ultimo*, *Bettel*, or *Piccolo*, the *Vorhand* must say *Cego* [39], [41], [42]. This starts the second bidding phase. The players bid on who gets to play a standardCego variant. The higher the bid, the harder it gets for the player. However, as an exchange, the player can receive more points at the end of the game because each level is attached to a point multiplier. In case the player doesn't want to play with the blind cards, that player says *Gut* (eng. *good*) and passes the chance on to the next player. When a player wishes to

[1] The Vorhand is the player to the right of the dealer.

bid, they must call the next higher step, such as *Halbe*. Then, it is the turn of the player who currently holds the highest bid. That player must either say *Gut* and leave the bidding to the other players, or they say *Selber* to call the current highest bid. When the player calls, the other player again has the chance to bid one step higher or pass. This process continues until every player has had the chance to place a bid. The player with the highest bid is allowed to play with the blind cards.

2.5.10 Counting of Points

After a standard sub-game e.g., Standard-Cego or Solo is played, the number of points each player wins or loses depends on how high the point difference of the losing side is relative to the 40 point mark [2], [39]–[42]. The calculation is in steps of 5 and then rounded up to 10. In case the single-player receives no trick, the number of points won by the winner and lost by the loser is 40.

Each bidding level is correlated with a multiplier:

- Cego: x1,
- Halbe: x2,
- Eine: x3,
- Eine Leere: x4,
- Zwei Leere: x5,
- Zwei Verschiedene: x6,
- Kleiner Mann: x7.

This factor is each increased by one when Gegensolo is played. A player who declared *Solo* wins by a factor of 2 and loses by a factor of 1.

Special game modes have a fixed number of points that winners gain and losers lose:

- Bettel: 30 points,
- Piccolo: 40 points,

- Utlimo: 80 points,
- Räuber: 40 points (The player who called the mode loses by the factor of 2).

2.6 MC Simulation

MC simulations [43] can be applied to solve numerical problems, where randomness can be used as a tool. This is when, for example, solving the problem directly by other means is not possible. It can also be applied to problems that are too complex to be addressed analytically. [44], [45]. In addition, it is possible to use this method to analyze certain changes within the environment.

In order to perform simulations, a model that imitates the real-life system is required [44], [45]. With this model, multiple random samples can be created. By analyzing these samples, approximations of certain probabilities can be made. MC simulations can be applied to probabilistic problems or, in general, problems where it is practical to identify the statistic distribution of a certain phenomenon by observing random n samples of the full environment space.

2.7 REST API

APIs are rule sets that ensure and regulate communication between systems [46], [47]. REST APIs are APIs that follow the restrictions of REST originally described by Fielding [48].

2.7.1 REST Properties

REST APIs have the following constraints [46], [48], [49]:

- **Client and server** have to be separated from each other.
- Communication must be **stateless**. A request contains all the information needed to be understood, and the server does not manage sessions.
- Responses should be **cacheable** and the server should inform the client when they are allowed to cache it. This is to improve efficiency and scalability.
- There should be a **unified interface** for the request of resources. Consequentially, the same request should always lead to the same resource[1].

[1] A resource is an entity that has an identifier [46], [49].

- Systems are **layered**. Information is restricted to a single layer. Client and server do not notice intermediaries between each other.
- (optional) Responses can contain code such as scripts or applets that can be executed. This is also known as **code on demand**.

2.7.2 Utilization of REST APIs

In the implementation, the communication between client and server takes place with *Hypertext Transfer Protocol (HTTP)*[1] requests [46], [50]. *create, read, update, and delete (CRUD)* operations are used to create, read, update, and write resources. POST requests create, GET requests read, PUT requests update, and DELETE requests delete resources. The current state of the representation can be returned in a human or machine-readable manner. A popular return format is *JavaScript Object Notation (JSON)*, which fulfills both requirements. Headers and parameters within requests include relevant information, such as metadata, the *Uniform Resource Identifier (URI)*[2], and authorizations. A well-designed REST-API returns HTTP status codes to indicate the success or failure of the request.

[1] HTTP is the protocol that allows the exchange of resources through the internet [49].
[2] The URI is the identifier that is directly mapped to a resource [49].

3 Analysis of the RL Framework

3.1 Deciding on an RL Framework

The first question that must be answered is what programming language to use for the specific problem of training a card game AI. The Cego Online 2.0 game environment is implemented with TypeScript, but there are almost no libraries for RL in either TypeScript or the interoperable JavaScript programming language, and none that are still maintained. Python offers a large selection of maintained RL frameworks [18]–[21], [23], [51]. Based on this factor, the decision falls on *Python*. An additional requirement that is set is that the frameworks have to be heavily focused on gaming problems. Two frameworks fit into this description.

The first framework in question is OpenSpiel [23], a framework that covers various kinds of games and algorithms to solve RL problems. It is a comprehensive framework for perfect and imperfect information games, single and multiagent RL. To analyze the environment and the underlying training dynamics, OpenSpiel provides analysis tools for evaluation. Games are implemented in C++, algorithms are implemented in Python as well as C++. The API is available for C++, Python, Julia, Go, and Rust.

OpenSpiel provides a simplified API for implementing various games and algorithms, but focuses heavily on the learning rather than the implementation aspect of RL problems [23]. It also has more dependencies than the other RL Framework that is discussed later. The extensive form representation of games, OpenSpiel uses, is useful for the generalization of various games, but not necessary when addressing a specific problem. For these reasons, the framework may pose an obstacle to the later export of trained models.

An alternative is RLCard [24], [51], a framework that, similar to OpenSpiel, provides an interface for implementing novel games but unlike OpenSpiel, it focuses on a particular type of game, the card game. Overall, RLCard provides fewer algorithms than OpenSpiel. In contrast, it's purely based on Python and has fewer dependencies than OpenSpiel.

Overall, RLCard is a lightweight framework specialized in the specific problem of the card game, which is fitting for the problem definition of this thesis. Based on the fact that it provides an implementation for visualizing and comparing trained models with each other called *RLCard-Showdown* [24], it is likely that a trained model can be exported and imported into other environments.

When directly comparing the two frameworks, the advantages of RLCard outweigh the advantages of OpenSpiel for this particular problem. Therefore RLCard is the framework of choice for this thesis.

3.2 RLCard

RLCard focuses on imperfect information games and multi-agent problems with sparse reward and large state and action spaces [51]. It is an open-source toolkit for card game problems. RLCard is usable within a multi-agent setting where each player move is carefully documented or within a single agent setting where the other players are simulated with, for example, pre-trained models.

3.2.1 Game Implementation Interface

RLCard does not enforce it, but proposes a unified structure of classes for card games [51]. The following classes are defined:

- ***Player***: An agent that can interact with the game.
- ***Game***: A complete representation of a game for the specific domain.
- ***Round***: A sub-sequence of actions within the game that can be derived from the natural game structure.
- ***Dealer***: Manages the randomness of the card deck and deals them to the players.
- ***Judger***: An independent entity that makes major decisions that affect the outcome of a game. Part of its responsibility is deciding which player gets which amount of reward at the end of the game.

3.2.2 Game Environment

The environment interface serves as an endpoint to the algorithms and tools of RLCard [51]. The class that implements this interface has to define a representation of a game state that players can observe from their point of view at a certain time step. A state representation consists of the following components:

- ***Legal Actions***: A list of actions that the player is allowed to take.
- ***Observation***: An encoding of the observed game state. The observation is a single or multidimensional array of float values.

For various games, different information may be of importance and therefore the state encoding has to be adjusted accordingly [51]. Additionally, the possible actions have to be encoded. Each action is mapped to an index.

Each aspect, such as state and action encoding, the reward system, and game rules, can be customized for individual problems [51].

3.2.3 The Training Interface

Algorithm 3 shows the pseudocode of how RLCard generally handles the training of agents [51]. First, the dependencies must be imported. After that, the environment and the agents must be initialized. A loop, that can be endless or limited to a certain number of episodes, manages the actual training. Each loop, the trajectory, and the payoffs are saved, which can later be used to train the agents.

```
# create environment
env = rlcard.make('cego')

#initialize agents
agents= [Agent(), ...]
env.set_agents(agents)

while True:
    # generate data from the environment
    trajectory, payoffs= env.run()
    # algorithm specific code for training agents ...
```

Algorithm 3: RLCard – Training of Agents
based on [51]

3.3 Evaluation of Models

The evaluation of environments and their agents is a difficult task. Agents' policies are usually measured by their *exploitability* [52]. This measurement searches for the most effective response to a policy, or more precisely, tries to exploit that policy. For this task, the game tree has to be traversed, which makes this measurement computationally expensive, especially for large game environments [53].

RLCard instead uses tournaments that compare models against random agents[1] or other models [51]. The size of tournament games can be adjusted and then the resulting average reward over these games is determined. Based on the law of large numbers [54], the accuracy of the reward average should increase when increasing the sample size. However, a larger sample size negatively affects the computation time.

3.4 Algorithms of RLCard

RLCard implements the four algorithms CFR [52], DQN [55], NFSP [56], and DMC [10].

Out of the named algorithms, only the latter three were considered for usage on Cego. CFR is an algorithm based on regret minimization [52]. It requires full traversal of the game tree at hand, making it computationally expensive. RLCard only uses this algorithm on the game *Leduc poker*, a simplified version of a poker game [51]. Meanwhile, Cego is a complex card game with a large game tree. As a consequence, this algorithm is unsuitable for solving Cego under reasonable time and hardware requirements.

The later three algorithms are all based on DNNs. This is practical for imperfect information that usually have a large game tree and many possible states [31], [32], [51]. Algorithms, e.g., Q-Learning or MC methods, would require a large Q-table for all states or state-action pairs that would grow to an unmanageable size for large environments. The following algorithms solve this problem by using DNNs to manage Q-tables instead. This change allows these algorithms to handle larger environments.

[1] Random agents are agents that select actions at random when it's their turn.

3.4.1 DQN

DQNs combines the Algorithm of Q-Learning with DNNs [55]. DQN makes use of a technique called experience replay, where the experience of the agent at each time step is stored. When applying Q-Learning, instead of directly sampling from a policy, the experience is sampled from this memory at random. The experience is filled by the agent selecting new actions based on an ϵ-greedy policy. For training, a fixed length of steps is sampled.

The loss function used for training the ANN is as follows [32], [55]:

$$L(\theta) \doteq \mathbb{E}_{\pi}\left[(y - Q(s, a; \theta))^2\right] \tag{9}$$

Given an observation, y is the expected target value [32], [55]. $Q(s, a; \theta)$ is the estimation of the ANN based on the weights θ. Through gradient descent, the ANN minimizes this loss function. With this mechanism, the ANN, called a Q-network, learns to approximate the optimal Q-function.

RLCard implements one specific improvement [57] that is designed to tackle an often observed problem of DQNs. Using Double DQNs [58] reduces the problem of overestimating actions values and, as a result, improves the performance of DQNs. The Double DQN expands on the idea of using a target network introduced by Mnih et al. [59]. Every r steps the online-network that selects actions copies the parameters of the target-network that evaluates and estimates the q-values. This subsequently stabilizes training.

The DQN implementation in RLCard defines the following hyperparameters [24], [57]:

- ***Replay Memory Size (int)***: The size of replay memory to sample from.
- ***Replay Memory Init Size (int)***: The number of random experiences to sample when initializing the replay memory.
- ***Update Target Estimator Every (int)***: The interval size of how often to copy the parameters of the Q-estimator to the target-estimator.
- ***Discount Factor (float)***: The discount factor γ.
- ***Epsilon Start (float)***: The chance to sample random actions ϵ at the start of training. This value decays over time.
- ***Epsilon End (float)***: The minimum ϵ at the end of the decaying process.
- ***Epsilon Decay Steps (int)***: The number of steps to decay epsilon over.
- ***Batch Size (int)***: The size of the batch to sample from the replay memory.
- ***MLP Layer (list)***: The dimension of the MLP.
- ***Learning Rate (float)***: The rate at which parameters are adjusted.

3.4.2 NFSP

NFSP extends FSP (see Subsection 2.4.4) by using ANNs for the approximation [56]. In the context of the training environment, each player is an NFSP-Agent. Initially, the agent manages two separate memories. Afterward, two separate ANNs are trained. The first ANN, in RLCard implemented as DQN, is trained with the data of the first memory M_{RL}. It represents the player's best responses as policy β. The second ANN is fed with data from the second memory M_{SL}. This network maps game states to action probabilities and therefore realizes the player's average response, denoted as policy π.

In contrast to FSP that only samples moves from the players' average responses, NFSP implements a mechanism that allows for using a mixture of the best response and average strategy policy [56].

NFSP uses two techniques to improve performance. Firstly, it uses *reservoir sampling* [60], which only adds experience to M_{SL} when it follows the best response. Secondly, it utilizes *anticipatory dynamics* [61] and, for that, implements an anticipatory parameter η. In practice, this means that the agent chooses actions with the following policy:

$$\sigma \equiv (1-\eta)\hat{\pi} + \eta\hat{\beta} \tag{10}$$

The parameter η allows balancing generating new experiences, and evaluating the best responses, as well as using the best responses to update the average strategy.

Algorithm 4 displays a detailed view of NFSP [56]. Before starting the training, the memories and networks are initialized. At the beginning of a training episode, there is a chance θ that the actions are sampled from, $\epsilon - greedy(Q)$ otherwise it is sampled from Π. Every time step an action is sampled, and the resulting reward r_{t+1} and follow-up state s_{t+1} is observed. All experiences are added to M_{RL}. Experiences are only added to M_{SL} when they follow the policy $\epsilon - greedy(Q)$. Each time step, gradient descent is performed for the parameters in Π and for the parameters in Q. The target network is updated periodically (see Subsection 3.4.1).

```
Initialize game $\Gamma$ and execute an agent via RUNAGENT for each player in the game
function RUNAGENT($\Gamma$)
    Initialize replay memories $M_{RL}$ (circular buffer) and $M_{SL}$ (reservoir buffer)
    Initialize average-policy network $\Pi(s,a|\theta^{\Pi})$ with random parameters $\theta^{\Pi}$
    Initialize action-value network $Q(s,a|\theta^{Q})$ with random parameters $\theta^{Q}$
    Initialize target network parameters $\theta^{Q'} \leftarrow \theta^{Q}$
    Initialize anticipatory parameter $\eta$
    for each episode do
        generate random float $r$ in range(0, 1)
        if $r < \theta$ then
            Set policy $\sigma \leftarrow \epsilon - greedy(Q)$
        else
            Set policy $\sigma \leftarrow \Pi$
        end if
        Observe initial information state $s_1$ and reward $r_1$
        for $t = 1, T$ do
            Sample action $a_t$ from policy $\sigma$
            Execute action $a_t$ in game and observe reward $r_{t+1}$ and next information state $s_{t+1}$
            Store transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in reinforcement learning memory $M_{RL}$
            if agent follows best response policy $\sigma = \epsilon - greedy(Q)$ then
                Store behaviour tuple $(s_t, a_t)$ in supervised learning memory $M_{SL}$
            end if
            Update $\theta^{\Pi}$ with stochastic gradient descent on loss
                $L(\theta^{\Pi}) = \mathbb{E}_{(s,a)\sim M_{SL}}[-\log \Pi(s,a|\theta^{\Pi})]$
            Update $\theta^{Q}$ with stochastic gradient descent on loss
                $L(\theta^{Q}) = \mathbb{E}_{(s,a,r,s')\sim M_{RL}}[(r + \max_{a'} Q(s',a'|\theta^{Q'}) - Q(s,a|\theta^{Q}))^2]$
            Periodically update target network parameters $\theta^{Q'} \leftarrow \theta^{Q}$
        end for
    end for
end function
```

Algorithm 4: NFSP Algorithm
based on [56]

The hyperparameters are partly identical to DQN. In addition, NFSP introduces the following hyperparameters [24], [57]:

- . . . DQN Hyperparameters.
- ***Hidden Layers Sizes (list)***: The size of the hidden layers for the average policy network.
- ***Reservoir Buffer Capacity (int)***: The size of the reservoir buffer of the average policy network.
- ***Anticipatory Param (float)***: The hyperparameter that balances SL and RL networks. The closer the number is to one the more weight is put on the best policy network (DQN) for the action sampling.
- ***Batch Size (int)***: The batch size for the average policy network.
- ***SL Learning Rate (float)***: The learning rate for the average policy network.
- ***Min Buffer Size To Learn (int)***: The minimum size needed for the average policy network to learn.

3.4.3 DMC

DMC is an algorithm that uses ANNs for the approximation of every-visit MC [10]. For balancing exploration and exploitation, DMC uses $\epsilon - greedy$ policy. The managed Q-network employs mean squared error (MSE) as the loss function. In realization, the Q-Network is an MLP. DMC is argued to be suitable for card games because it bypasses the weakness of MC methods to not be able to process incomplete episodes. This is because card games episodes are supposed to have an end state. Therefore, the problem of incomplete episodes does not occur. It is also believed to be more inefficient than other algorithms such as Q-Learning. To counteract this problem, DMC parallelizes the sampling process.

The learning process algorithm of DMC is shown in Algorithm 5. One requirement is the usage of shared buffers that allow sharing data between multiple running processes. Initially, global Q-networks are initialized [10]. In each iteration, for each player, it is observed whether the player's buffer contains complete episodes. If this is the case, the weights of the players can be adjusted by gradient descent with the MSE loss function, where the target is the provided data.

```
Input: Shared buffers $B_n$ with $B$ entries for players [0..n]
    and size $S$ for each entry, batch size $M$ , learning rate $\psi$
Initialize global Q-networks $Q_0^G$, ..., $Q_n^G$
for iteration = 1, 2, ... until convergence do
    for $p \in P$ do
        if the number of full entries in $B_p \geq M$ then
            Sample a batch of $\{s_t, a_t, r_t\}$ with $M \times S$ instances
    from $B_p$ and free the entries
            Update $Q_p^G$ with MSE loss and learning rate $\psi$
        end if
    end for
end for
```

Algorithm 5: DMC Algorithm - Learning Process
based on [10]

One or multiple actor processes (see Algorithm 6) create and provide the data. For every actor, a local Q-network and a local buffer for each player are initialized [10]. In every iteration, the local and global Q-Networks are synchronized beforehand. For the selection of an action, $\epsilon - greedy$ is used relative to the relevant player Q-network. The effect of this action is then observed and saved in the local buffer. When a player's game terminates, the episode is moved to the global buffer when space exists. For the implementation, many of the processes are multithreaded in order to improve training speed.

```
Input: Shared buffers $B_n$ with $B$ entries for players [0..n]
    and size $S$ for each entry, exploration hyperparameter $\epsilon$,
    discount factor $\gamma$
Initialize local Q-networks $Q_0$, ..., $Q_n$ and local buffers $D_0$,
    ..., $D_n$
for iteration = 1, 2, ... do
    Synchronize $Q_0$, ..., $Q_n$ with the learner process
    for t = 1, 2, ... T do
        $Q \leftarrow$ one of $Q_0$, ... , $Q_n$ based on position
        $a_t \leftarrow \epsilon - greedy(Q(s_t, a))$
        Perform $a_t$, observe $s_{t+1}$ and reward $r_t$
        Store $\{s_t, a_t, r_t\}$ to one of $D_0$, ..., $D_n$ accordingly
    end for
    for t = T-1, T-2, ... 1 do
        $r_t \leftarrow r_t + \gamma r_{t+1}$ and update $r_t$ in one of $D_0$, ..., $D_n$
    end for
    for $p \in P$ do
        if $D_p$.length $\geq$ $L$ then
            Request and wait for an empty entry in $B_p$
            Move $\{s_t, a_t, r_t\}$ of size $L$ from $D_p$ to $B_p$
        end if
    end for
end for
```

Algorithm 6: DMC Algorithm - Actor
based on [10]

The DMC implementation defines the following hyperparameters [10], [24], [57]:

- ***Num Actor Devices (int)***: The number of devices to use for simulations.
- ***Num Actors (int)***: The number of actors for each simulation device.
- ***MLP Size (int)***: The dimension of the MLP for each agent.
- ***Total Frames (int)***: The total number of environment frames to train.
- ***Exp Epsilon (float)***: The exploration Factor.
- ***Batch Size (int)***: The mini-batch size for learning.
- ***Unroll Length (int)***: The unroll length (time dimension).
- ***Num Buffers (int)***: The number of shared memory buffers.
- ***Num Threads (int)***: The number of learning threads.

- ***Max Grad Norm (int)***: The max norm of gradients.
- ***Learning Rate (float)***: The learning rate.
- ***Alpha (float)***: The *RMSProp*[1] smoothing constant.
- ***Momentum***: The RMSProp momentum.
- ***Epsilon***: The RMSProp epsilon.

[1] RMSProp is an. optimization algorithm utilized in ANNs [62]

4 Game Implementation Into RL Framework

Cego is a complex game with many branching sub-games. Initially, the number of possible starting states is calculated to better understand this complexity. This can be done with the binomial coefficients for each player's hand as well as for the blind deck [63], [64]. The following calculation describes the number of unique Cego deals, and therefore possible initial information sets $|I_0|$:

$$\begin{aligned} |I_0| &= \binom{54}{11} \cdot \binom{43}{11} \cdot \binom{32}{11} \cdot \binom{21}{11} \cdot \binom{10}{10} \\ &\approx 2.51 \cdot 10^{34} \end{aligned} \tag{11}$$

In addition to the various sub-games and card-playing options for each player, the full game tree can be considered relatively large. Another problem is the large action space of Cego. 54 unique cards can be played. When bidding, the player has several options to call game modes. Within the various Standard-Cego variants, the single-player can also keep, discard, and exchange cards. The division of the game into sub-problems could reduce this complexity.

For this thesis, the decision was made to split up each sub-game into a sub-problem. Because the bidding phase is vastly different compared to the other sub-problems and heavily dependent on the result of those, it requires separate consideration. Therefore, the bidding phase is excluded from the discussion.

4.1 Cego Within the Game Theory Context

Cego can, in most aspects, be described as a two-player zero-sum game. Technically, it is played by four players, but these players are divided into two teams. Whenever one team wins, the other team loses. One party consists of one player and one party consists of three players. Räuber is a game mode in which everyone plays against each other. However, there still is a clear winner and a loser side.

Cego remains a classic trick card game where, in the beginning, the cards are shuffled and dealt face down to each player. A player does not know what cards were dealt to the other players. Therefore, Cego is an imperfect information game.

In Cego, players play cards one by one in order, resulting in Cego being a sequential game. A Cego game could, thus, be described in extensive form.

4.2 Implementation of the Environment

As discussed in Subsection 3.2.2 an environment that simulates the game must be implemented in order to train AI models. This process requires careful thought of the actions that the model can select and the encoding that the agent observes as an information state. This section discusses the final action and state encoding approach.

4.2.1 Action Encoding

First, the question has to be answered, what aspect of the game should be learned by trained AI models?

The exclusion of the bidding phase simplifies the action space and only the playing of cards must be addressed. For the sub-games, it is reasonable to encode each card as a different action, as is done in other game implementations in RLCard [57]. But a different problem occurs when observing the sub-game Standard-Cego and its comparative sub-forms. At the start of a round, the player has to decide on what cards to keep, what cards to discard, what cards to lay out, and sometimes what cards to exchange. As a result, this would add four more possible actions for each card and would increase the action space by the multiplier of four.

The Zha et al. [10] discuss the negative effect large action spaces have on the training of RL models. Moreover, this is a small sub-problem that only applies to the single-player mode and only at the beginning of the game. For these reasons, the decision was made to exclude these types of actions from the action encoding.

The resulting action encoding is shown in Table 2. Each action is assigned an index.

Actions/ Cards	Indices
Clubs	0 – 7
Spades	8 – 15
Hearts	16 – 23
Diamonds	24 – 31
Trumps	32 – 53

Table 2: Action Encoding of Cards

4.2.2 State Encoding

There are numerous aspects of the game state that can be observed in a game of Cego. A few examples are:

- The hand cards the current player has,
- The cards in the current trick,
- The card that has to be served,
- The card that currently wins the trick.

There may be more information that can be observed and is not mentioned in the list, but not all information may be visible or of relevance to the player.

As a result, a careful evaluation and selection of the observed game state is required. It should contain enough information for the algorithm to make proper assumptions and learn from. However, it should also contain just enough information to reduce complexity and not hinder the training of qualitative models [51] [25]. The format of the state should also be machine-readable, preferably *ones* and *zeros*.

The final encoding is shown in Table 3. The Game State is converted and encoded to an Array of the Size *336* with binary values. Each index encodes a different part of the information state. It describes what an agent observes in the environment. Encodings 1 through 6 each have a size of 54. *One* indicates the presence of one of 54 cards and *zero* represents its absence. Each card can be encoded as an index. The exact card encoding is represented in Appendix C. For example, at the beginning of a sub-game, the encoding of the hand should contain 11 *ones* and 43 *zeros*.

Based on the agent's knowledge of already known cards, encoding 2 contains all the cards that can still be played by other players other than themselves. So the playable cards are the result of the following equation:

$$\begin{aligned} playebleCards = & allCards \\ & - playedCards \\ & - handCards \\ & - legageCards \end{aligned} \tag{12}$$

An argument could be made that it would be equally effective to encode all the known cards, but arguing from a human point of view, it is logical to focus on the cards that still could be played by other players rather than the cards already out of the game.

Because the position of the card in the trick should affect the player's behavior, relative to who played the card, each of the cards in the trick is encoded independently (4, 5, 6). The winning card (2) is encoded separately to emphasize that particular card. The last encodings 7–9 contain information about the players. As an example for (8), consider that player 2 of 4 currently wins the trick round. The resulting encoding of (8) would be $[0, 1, 0, 0]$.

Num.	Description	Indices
1	The agent's hand cards	0 – 53
2	The cards that still can be played by other players	54 – 107
3	The card that currently wins the trick	108 – 161
4	The first card in the trick	162 – 215
5	The second card in the trick	216 – 269
6	The third card in the trick	270 – 323
7	The players in the same team	324 – 327
8	The player that currently wins the trick	328 – 331
9	The player who started the trick round	332 – 335

Table 3: Information State Encoding

4.3 Reward Handling

When considering the Cego sub-games, there are two methods to calculate the end reward of agents.

4.3.1 Variant 1 – One Point per Won Game

The first option is to consider whether a player has won or lost a game. A winner receives *one* point at the end of the game. Losing results in receiving no reward. This reward system is feasible for most game modes, but may not be well suited for the game modes Solo and Standard-Cego.

4.3.2 Variant 2 – Using the Point System of Cego

The second possibility is to calculate the reward by the value system the card game uses. Each time a player wins a trick, they receive the value of the trick as a reward. If the single-player wins the trick, only that player gets the points. Additionally, the single-player gets points for the Legage. Because the other players play together, if one player of the team wins, all the other players receive the reward as well. That means that the reward of the three opponents is mirrored.

Considering the starting points before the first trick round started are $[15, 0, 0, 0]$. The single-player earned Legage points. In the first round, player number 3 won a trick with the value of 5 points, so the resulting points after that round are $[15, 5, 5, 5]$.

This point system allows for more possible outcomes in rewards. The maximum reward an agent can receive is 79, and the minimum is 0. This is also suitable in a practical sense because in the game modes Standard-Cego and Solo a higher earned point count directly leads to winning more points at the end of the game. As a result, the AI should be able to learn these nuances in point differences. Nevertheless, it should be noted that the agents still only receive the reward after the game. What sequence led to the increase of points has to be figured out by the agent. Moreover, this reward calculation is only feasible for the game modes Solo and Standard-Cego. The other game modes require the reward system described in subsection 4.3.1. The other sub-games effectively produce two outcomes for each player.

4.4 Implementation of the Game Logic

An environment class communicates with a game class that represents the game logic. The implementation orients on the structure described in Subsection 3.2.1.

The following list describes the implemented classes:

- ***CegoCard***: Represents a Cego card.
- ***CegoRound***: Represent a single trick round within a Cego sub-game.
- ***CegoPlayer***: A player in a Cego game.

- ***CegoGame*** (abstract): An abstract class for the representation and implementation of a single sub-game. Its implementations are the classes *CegoGameStandard*, *CegoGameSolo*, *CegoGameUltimo*, *CegoGamePiccolo*, *CegoGameBettel*, and *CegoGameRaeuber*.
- ***CegoJudger*** (abstract): An abstract class that is used for judging a Cego sub-game. Its implementations are the classes *CegoJudgerStandard*, *CegoJudgerUltimo*, *CegoJudgerPiccolo*, *CegoJudgerBettel*, and *CegoJudger-Raeuber*.
- ***CegoDealer***: This class is responsible for dealing cards for the various sub-games.

An overview of the classes and the relations between classes can be seen in Figure 5. It should be noted that this class diagram does only display the main game classes. It does not include helper functions and utility classes used, for example, for the simulations of games, the evaluations, and the mapping of game states.

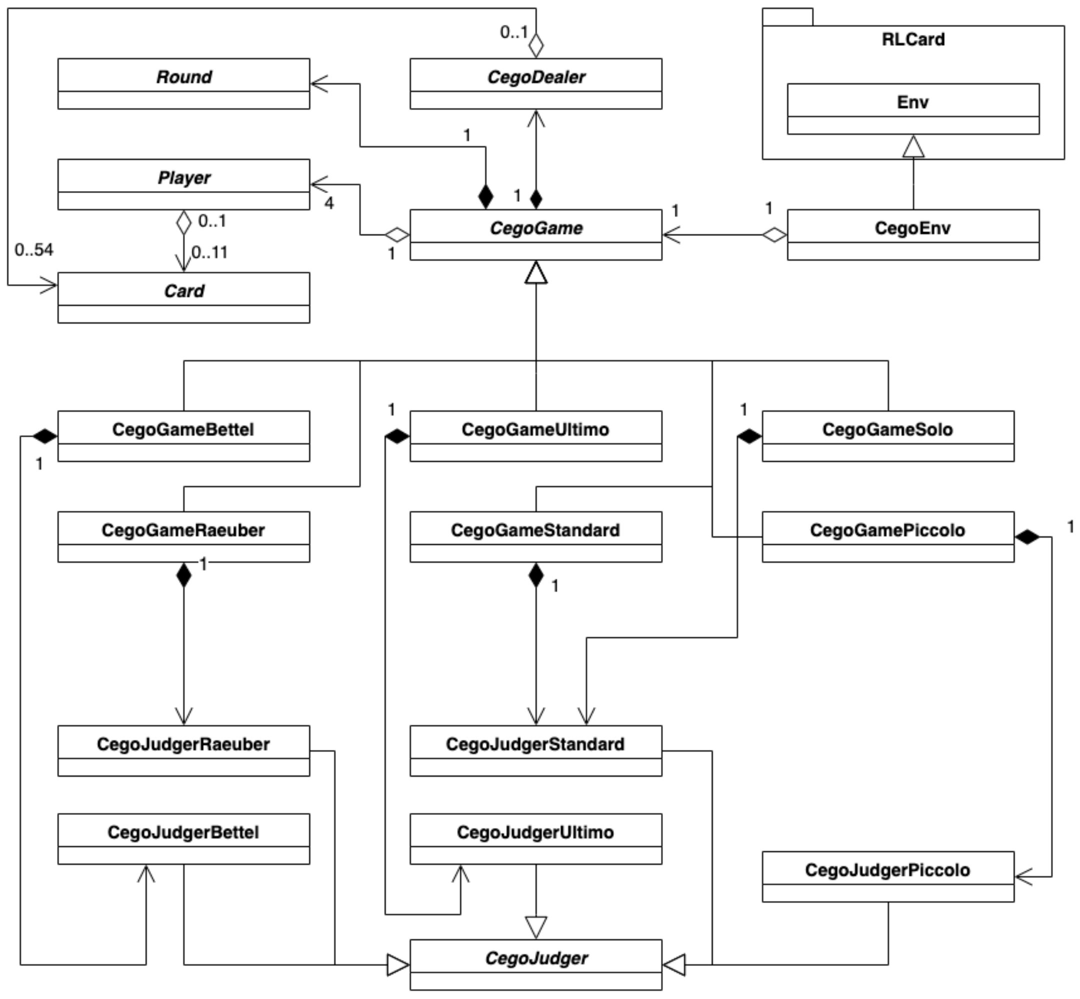


Figure 5: Class Diagram

The following subsections will describe these classes and their main emphases in more detail.

4.4.1 CegoCard

The focus of this class lies in the comparison and evaluation of cards. This class has two main attributes: suit and rank. For game state encoding, cards can be encoded into their index representation.

The class also defines a static method, which returns the card that currently wins the trick. The logic of this method is shown in Figure 6. The method is called when a card is played and a current winning card exists. Otherwise, the played card wins by default. It is assured beforehand that players are only allowed to play legal cards. First, the current winning card and the played card are compared by their suit. If the suits are the same, the cards can be compared by rank. If not, that means that either only the played card is a trump or the played card has no value relative to the winning card. This is because the player did not follow suit. As a result, the new winning card is determined.

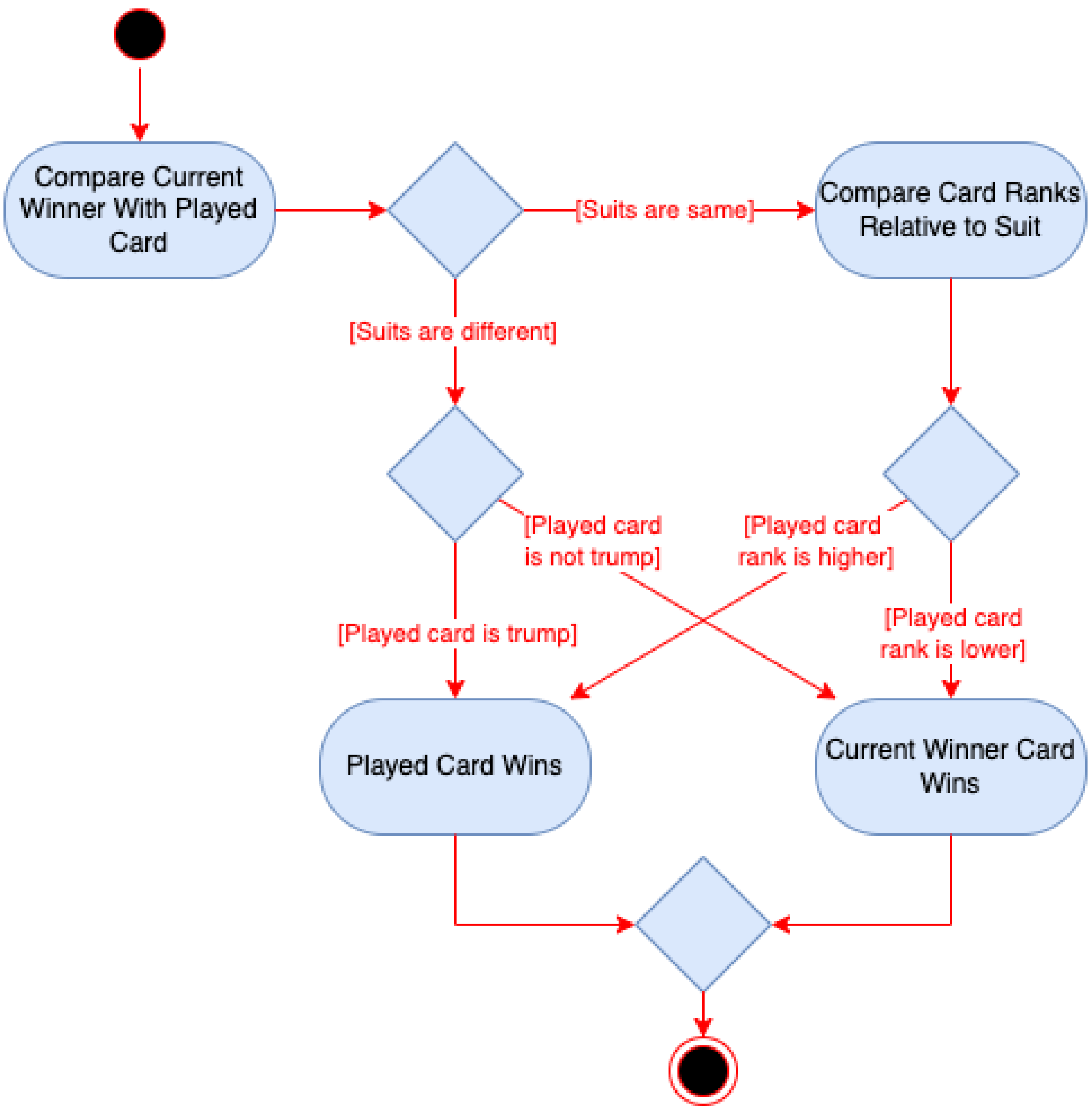

Compare Current Winner With Played Card
[Suits are same]
Compare Card Ranks Relative to Suit
[Suits are different]
[Played card is not trump]
[Played card rank is higher]
[Played card is trump]
[Played card rank is lower]
Played Card Wins
Current Winner Card Wins

Figure 6: Activity Diagram – Deciding on the Winning Card

A particular difficulty is the difference in rankings between red and black cards. This problem was solved by providing different lookup tables for the different suits.

4.4.2 CegoRound

Independent of the selected sub-game, rounds in Cego are always played the same way. The class handles the actual playing of trick rounds. An object of this class manages the current cards in the trick and the card, and the player who wins the trick. It also manages what legal action each player can take.

Figure 7 visualizes the legal action logic. First, it has to be checked if there already is a first card in the trick. This card is defined as the target card. If not, all the hand cards are legal to play. Otherwise, a comparison must be made to determine whether the suits of the hand cards fit the suit of the target card. If true, all the cards that fit this condition can be selected as valid cards. When that's not the case, it has to be checked if there are trump cards at hand. If true, only these trump cards are legal. When none of these conditions are met, all hand cards that remain can be considered legal.

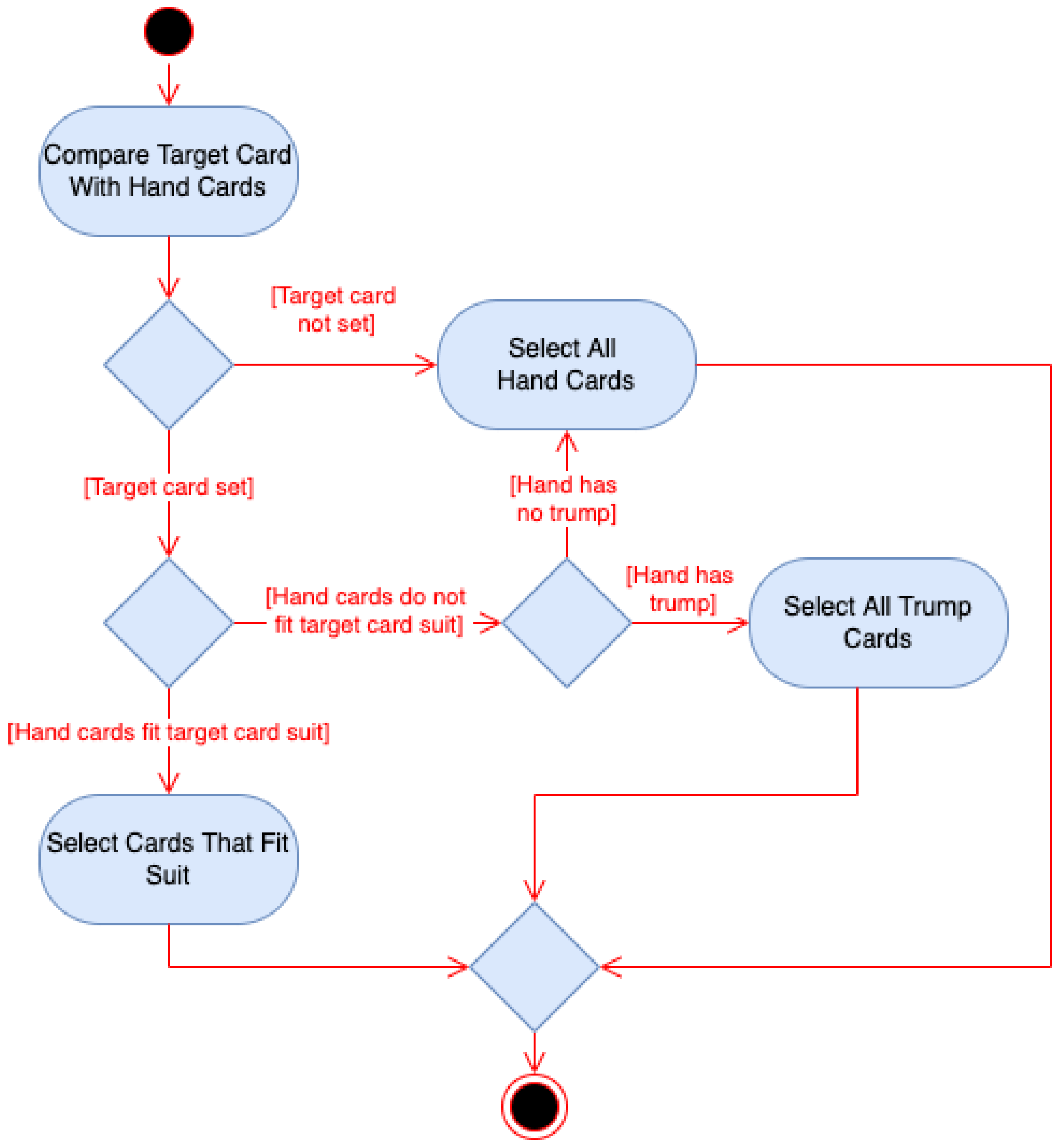

Compare Target Card With Hand Cards
[Target card not set]
Select All Hand Cards
[Target card set]
[Hand has no trump]
[Hand cards do not fit target card suit]
[Hand has trump]
Select All Trump Cards
[Hand cards fit target card suit]
Select Cards That Fit Suit

Figure 7: Activity Diagram – Legal Action Selection

4.4.3 CegoPlayer

An object of this class represents a player. The player gets dealt cards at the beginning of the game by the dealer and saves them in the attribute *hand*. In most game modes, there is one single-player, which is indicated by a boolean *is_single_player*. If the played sub-game is Standard-Cego, the single-player also knows the Legage cards. The player saves this information in the attribute *legage*.

4.4.4 CegoGame

Because the sub-games have many similarities, most of the game logic is abstracted into a class named *CegoGame*. One similarity is, for example, a Cego game always has four players. The player in the first position is (in all but the game mode, Räuber) considered the single-player. The other players are positioned relative to that player. The second player is the player to the right of the single-player, and so on.

The class manages a whole sub-game. It instantiates the judger, the dealer, the round, and the players and initiates a game. Additionally, this class has the responsibility to check whether the sub-game has terminated, to maintain what information the players have, and the full information state of the game. The main differences between the sub-games are implemented in child classes.

The *CegoGameStandard* implementation represents not only Standard-Cego but can also represent all its comparative forms described in 2.5.7. For training and testing purposes, a rule-based approach to creating the Standard-Cego single-player's hand deck was implemented. The two highest ranked cards of the hand deck are kept and the lowest ranked card of the blind cards is discarded.

The other implementations are *CegoGameSolo* for the sub game Solo, *CegoGameUltimo* for the game mode Ultimo, *CegoGamePiccolo* for the game mode Piccolo, *CegoGameBettel* for the game mode Bettel, and *CegoGameRaeuber* for the game mode Räuber.

4.4.5 CegoDealer

The CegoDealer takes care of dealing cards to the player as well as dealing the blind cards. Depending on the sub-game, rules based on heuristic knowledge were implemented to filter the games reasonably and reduce the number of possible games. The dealer takes care of managing these heuristics. What kind of heuristics were used are discussed later in this chapter.

4.4.6 CegoJudger

This class serves as an interface for the various kinds of game evaluations for the different sub-games. Most sub-games handle win conditions vastly differently and, therefore, the interface has to be implemented accordingly. *CegoJudgerStandard* implements the judger for the sub-games Standard-Cego and Solo, which are evaluated in the same way. The other game modes each implement their own judger class.

One significant difference between Räuber and other game modes is that in the other modes, there can be either one winner when the single player won or three winners when the single player lost. In Räuber, there are always three winners and one loser. This should be noted when looking at later results.

4.5 Taking into Account Heuristic Knowledge

To minimize the number of possible games and therefore game states, rules based on heuristics were implemented for the sub-games. These rules intend to filter games that are unlikely to be played by experienced Cego players. Additionally, these heuristics aim to increase the winning chance of the sometimes disadvantaged single-player.

4.5.1 Standard-Cego

In order to bid on Standard-Cego, a player should have at least 15–17 value points on their hand [65]. When the player's hand value points add up to a higher number, that player can try to bid for higher stakes.

For the implementation, only when the single-player has at least 15 points on their hand the game is considered.

4.5.2 Solo

The heuristic for Solo can be defined by the rule of the *Solo obligation* (Ger. *Solopflicht*) [41] which is often used in regions, but not in the Schwarzwaldmeisterschaft [40]. The player has to play Solo when they have at least 8 trump cards, or when they have 7 trump cards, and two of them are higher or equal compared to the rank of *17-Trump*, and they have only two colors.

4.5.3 Ultimo

When considering the game rules, the Ultimo player has an advantage when they are able to control the game flow. This flow can be effectively controlled with strong trump cards. In addition, a trump card, when played at the start of the trick round, forces the other players to follow suit and play their trump cards. Therefore, for the single-player to have a reasonable chance at winning Ultimo, the player should have a strong trump.

This work defines a strong trump as having at least 8 trump cards, with two cards higher or equal to the card with the rank of *17-Trump*. This and the general Ultimo condition to have the *1-trump* card are combined to form the total Ultimo game selection rule.

4.5.4 Bettel

The Bettel single-player is more likely to win no tricks when the hand cards have a low chance of winning a trick. Accordingly, for the game mode Bettel, an examination of the trick win probabilities of all possible cards was made that is discussed in the next chapter. Based on this examination, the following approach was taken. The next chapter defines *high cards* and *low cards*. For a player to play Bettel they are only allowed to have cards in their hand that can be defined as low cards.

4.5.5 Piccolo

This game mode takes the Bettel game selection as a base with the addition that one of the cards has to be a *high card*. This shall make it easier for the AI player to win a single trick, and it also helps to differentiate between the game modes for a potential rule-based bidding system approach.

4.5.6 Räuber

For Räuber, two approaches can be taken. One can assume, based on the fact that Räuber is the last sub-game that can be called, that all other game modes are not valid. In conclusion, the game selection can be described as all possible start games, minus other game modes being possible. This approach might offer better performance results in the game states included in this set. But it is uncertain how the DNN may act, when other non AI players decide to play this game mode in cases that are not part of this set of starting states.

As an alternative, the simple approach could be taken to ignore all heuristics for this game mode. Because this is the only game mode where each player plays on their own, it is probably the most evenly matched game mode of all. There is therefore no need to level player advantages. As a consequence, this approach might provide an AI model that is more robust to unexpected game states. Therefore, a heuristic-less approach was selected for this sub-game.

5 Analysis of the Game Environment

This chapter introduces metrics for evaluating the quality of AI models. In addition, this chapter makes MC simulations to analyze the behavior of the game environment.

5.1 Metrics

For comparison of teams and players, two metrics were defined based on the reward systems discussed in 4.3. Given a team A is compared with the opposing team B, the following metrics are defined:

- ***Winning percentage (WP)***: The number of games won by the team divided by the number of games played [66].
- ***Average points per game (APPG)***: The average amount of points gained per game, calculated by dividing the total amount of points accumulated by the team through the number of games played.
- ***Training improvement over time (TIOT)***: This metric is measured by making a linear regression of the training progress points of a model and extracting the slope [67].

The APPG can be deemed as the more significant metric for the sub-games Standard-Cego and Solo, because of the implications the point score has in those sub-games (see Subsection 2.5.10). For these sub-games, the WP is added because it can help to put the results into larger contexts for a game mode overlapping comparison. Sub-games outside Standard-Cego and Solo only use the WP.

The TIOT is only utilized in the HPO process.

5.2 Approximating Probabilities of Card Trick Win Percentages

To better understand how likely it is for a card to win a trick, MC simulations were made. In 10^6 games, four random agents play against each other in a round of *Solo* without heuristic game selection. This game mode was chosen over Standard-Cego because of the card selection process for the single-player in Standard-Cego, which may falsify the approximation results. Overall, these modes have close similarities, and therefore they can be substituted. The random seed of the environment is 12. For the number of games, it is saved how often a card is played, the number of tricks played, and how often a card wins a trick round.

5.2.1 Probability – Specific Card Winning a Trick When Played

The probability that a card with the index i was played in a trick round is denoted as $P(CP_i)$. $P(W_i)$ is the probability that the card with the index i won a trick. Accordingly, the probability that when card i was played, it won the trick can be defined as $P(W_i|CP_i)$. The following calculation approximates this probability over several simulation games with x_i card trick wins and y_i card occurrences for the card i in the case of $y_i > 0$:

$$P(W_i|CP_i) \approx \frac{x_i}{y_i} \tag{13}$$

It is important to note that this probability is a generalization of a more complex problem. Win probabilities of cards change on round per round and positional bases. This simplification serves to establish an overall understanding of the strength of the cards.

The approximation of $P(W_i|CP_i)$ with the results from the 10^6 simulated games leads to the results reported in Table 4. The approximation of probability $P(W_i|CP_i)$ where $i =$*Gstieß-Trump* is 100% as is expected because it is the highest ranked card and therefore has to win every trick if played. The chances of winning a trick when played are roughly equal for cards of the same rank. It can be assumed that with an infinite number of games, the chances of the same ranked cards would approach the same value.

i	$\approx P(W_i \| CP_i)$ in %	i	$\approx P(W_i \| CP_i)$ in %
Gstieß-Trump	100	3-Trump	19.87
21-Trump	90.68	2-Trump	18.89
20-Trump	82.26	1-Trump	18.13
19-Trump	74.41	Cavalier-Heart	14.18
18-Trump	67.33	Cavalier-Spade	14.16
17-Trump	60.99	Cavalier-Diamond	14.1
16-Trump	55.12	Cavalier-Club	14.08
15-Trump	49.94	Jack-Club	7.71
14-Trump	45.29	Jack-Heart	7.68
13-Trump	41.18	Jack-Diamond	7.66
King-Diamond	38.85	Jack-Spade	7.64
King-Heart	38.83	10-Club	4.03
King-Spade	38.75	1-Diamond	4.0
King-Club	38.73	10-Spade	4.0
12-Trump	37.48	1-Heart	3.98
11-Trump	34.3	2-Heart	2.28
10-Trump	31.46	9-Club	2.27
9-Trump	29.04	2-Diamond	2.25
8-Trump	26.83	9-Spade	2.24
7-Trump	25.0	8-Club	1.61
Queen-Diamond	24.23	8-Spade	1.6
Queen-Club	24.22	3-Diamond	1.59
Queen-Spade	24.21	3-Heart	1.57
Queen-Heart	24.11	4-Heart	1.15
6-Trump	23.56	7-Club	1.14
5-Trump	22.21	4-Diamond	1.14
4-Trump	20.91	7-Space	1.13

Table 4: Approximation of $P(W_i|CP_i)$ for All Cards
(Results are rounded to two decimal places)

From the simulation results, it can be observed that trumps do not always have a higher chance of winning a trick compared to cards of a different suit. As an example, kings have a higher chance of winning a trick than trump cards, with a number lower than 13.

The second approximation of probabilities will further prove that trumps are not always more likely to win tricks than different colored cards.

5.2.2 Probability – Trick Was Won by a Specific Card

The probability that a card was played in a trick and won the trick can be defined as $P(W_i \cap CP_i)$. The probability $P(W_i)$ can be derived from its total probability [63]:

$$P(W_i) = P(W_i \cap CP_i) + P(W_i \cap \overline{CP_i}) \tag{14}$$

Because $P(W_i \cap \overline{CP_i}) = 0$, therefore, $P(W_i) = P(CP_i \cap W_i)$. In the inspection of the simulation, this probability can be approximated with the card wins x_i over n games and the constraint of 11 trick rounds in a game with the following calculation:

$$P(W_i) \approx \frac{x_i}{n \cdot 11} \tag{15}$$

As with the approximation before, this probability is a generalization of the full problem.

The approximation results of this probability over 10^6 games are shown in Table 5. The trick win probability of *Gstieß-Trump* almost aligns with the generalized probability of an individual card i being played in a trick, which can be calculated with the number of cards played in a trick $dim(t)$ and the number of total individual cards $dim(c)$ in the following manner:

$$\begin{aligned} P(CP_i) &= \frac{dim(t)}{dim(c)} \\ &= \frac{4}{54} \\ &\approx 0.0741 \end{aligned} \tag{16}$$

Because *Gstieß-Trump* has a 100% chance of winning the trick, the probability of $P(W_i)$ should align with $P(CP_i)$. The observation supports this.

As can be observed, the order of cards is almost the same as in Table 4, with minor differences in cards that have the same rank. In general, it can be assumed that kings and queens overall have a higher chance of winning tricks than some trump cards.

i	$\approx P(W_i)$ in %	i	$\approx P(W_i)$ in %
Gstieß-Trump	7.41	3-Trump	1.47
21-Trump	6.72	2-Trump	1.41
20-Trump	6.09	1-Trump	1.34
19-Trump	5.51	Cavalier-Spade	1.05
18-Trump	4.99	Cavalier-Club	1.05
17-Trump	4.52	Cavalier-Heart	1.05
16-Trump	4.08	Cavalier-Diamond	1.04
15-Trump	3.7	Jack-Diamond	0.57
14-Trump	3.35	Jack-Club	0.57
13-Trump	3.05	Jack-Spade	0.57
King-Diamond	2.88	Jack-Heart	0.57
King-Heart	2.88	1-Diamond	0.3
King-Club	2.87	1-Heart	0.29
King-Spade	2.87	10-Club	0.29
12-Trump	2.78	10-Spade	0.29
11-Trump	2.54	2-Heart	0.17
10-Trump	2.33	9-Club	0.17
9-Trump	2.15	9-Spade	0.17
8-Trump	1.99	2-Diamond	0.17
7-Trump	1.85	3-Heart	0.12
Queen-Diamond	1.79	8-Spade	0.12
Queen-Club	1.79	3-Diamond	0.12
Queen-Spade	1.79	8-Club	0.12
Queen-Heart	1.79	4-Heart	0.09
6-Trump	1.74	4-Diamond	0.08
5-Trump	1.64	7-Club	0.08
4-Trump	1.55	7-Space	0.08

Table 5: Approximation of $P(W_i)$ for All Cards
(Results are rounded to two decimal places)

5.3 Defining High and Low Cards

Using the approximations in the previous sections, it is possible to abbreviate what this thesis defines as low and high cards in Cego. In this work, we define *low cards* as the biggest possible set of unique cards that combined account for less than 20% of all trick wins. The set of cards that is left out of this condition is defined as *high cards*.

The division of high and low cards is shown in Figure 8. For the probability values, Table 5 was used. All cards are sorted from the highest to the lowest trick win probability. The values are extracted from Table 5. Overall, the 30 lowest trick win chance cards (blue) account for around 18.28% of all trick wins and the 24 highest trick win chance cards (violet) account for around 81.72% of all trick wins.

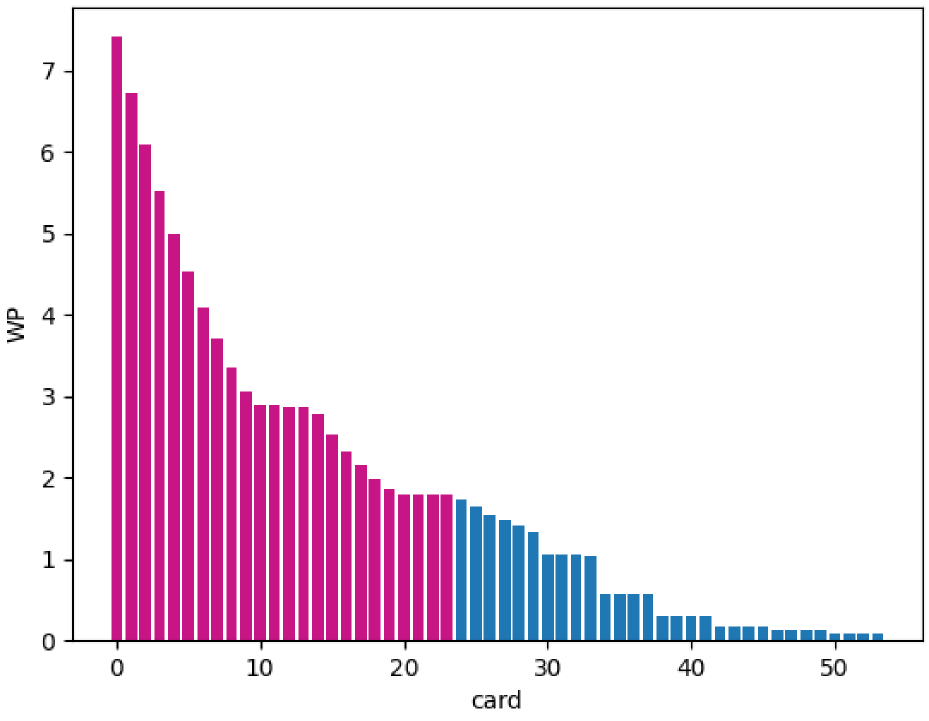


Figure 8: Visualization of High and Low Cards

With this formulation, the game selection rules for the sub-games Bettel and Piccolo can be implemented successfully.

5.4 Analyzing Player Advantages in Cego

For these MC simulations, the advantages of player factions will be studied. In addition, the effects of the heuristics defined in Section 4.5 are analyzed and put into context. Through these simulations, it can be demonstrated that the implemented heuristics improve the chances of winning for the single-player. Additionally, the results of these experiments can serve as benchmarks for the comparison of algorithms. AI models should be able to achieve higher scores than specified in this section.

Table 6 highlights the results. The winning players are marked as bold. For each game mode, 10^6 games were simulated with random agents and random seed 12, once with the consideration of heuristic knowledge (see Table 6 *Heuristics*) and once without the consideration of heuristic knowledge (see Table 6 *!Heuristics*). Considering heuristic knowledge increases the single-player winning chance across all game modes. The only game mode that does not consider heuristic knowledge, Räuber, demonstrates a roughly evenly matched WP for all players. In the Piccolo mode, winning chances are significantly improved, although the advantage doesn't rise to a full advantage for single-player. The sub-game Bettel displays the highest improvements when considering heuristics.

Player Position	Standard-Cego	
	APPG !Heuristics	APPG Heuristics
1	38.761	**40.795**
2	**40.239**	38.205
3	**40.239**	38.205
4	**40.239**	38.205

Player Position	Solo	
	APPG !Heuristics	APPG Heuristics
1	30.788	**48.996**
2	**48.212**	30.004
3	**48.212**	30.004
4	**48.212**	30.004

Player Position	Ultimo	
	WP !Heuristics	WP Heuristics
1	2.369	8.42
2	**97.631**	**91.58**
3	**97.631**	**91.58**
4	**97.631**	**91.58**

Player Position	Bettel	
	WP !Heuristics	WP Heuristics
1	6.633	**73.081**
2	**93.367**	26.919
3	**93.367**	26.919
4	**93.367**	26.919

Player Position	Piccolo	
	WP !Heuristics	WP Heuristics
1	17.986	39.275
2	**82.014**	**60.725**
3	**82.014**	**60.725**
4	**82.014**	**60.725**

Player Position	Räuber
	WP !Heuristics
1	73.848
2	**74.839**
3	**75.425**
4	**75.887**

Table 6: Player Advantages in Cego Sub Modes, With and Without Heuristics Considered

(Results rounded to three decimal places)

6 HPO

The Implementations of DQN and NFSP in RLCard were only lightly tuned for testing [51]. DMC on the other hand, offers a tuned base for the game DouDizuh [10] which has some similarities to Cego in terms of gameplay.

For the process of HPO, the decision was made to re-tune the hyperparameters of DQN as well as NFSP to ensure the best possible training results for these algorithms. As a method for optimizing hyperparameters, a random search approach [68], [69] is applied. The training of DMC is based on the pre-tuned hyperparameters provided by Zha et al. [10].

6.1 The Random Search Framework

DQN and NFSP each sample from a set range of hyperparameters. The number of samples that are taken is 20. The same hyperparameters can't be sampled twice. To compare, the reward system discussed in 4.3, this process is repeated one time with the first reward system and one time with the second reward system. This is done for each algorithm. In conclusion, 40 unique models are trained for each algorithm.

6.1.1 Training Process

Each sampled hyperparameter set is applied to train a model for 50,000 episodes. Every 500 episodes, the current model is compared against random agents within a tournament of 1,000 games. The average tournament reward is saved as the current training progress. As a result, a training graph is created to compare the training results. All models in this chapter were trained on the same sub-game Standard-Cego with heuristic game selection and the same random seed 12.

The environment is initialized, with the single-player acting as the algorithm and the other players as random agents. This approach differs from the NFSP approach of initializing an NFSP-agent for each player (see 3.4.2). Because Cego is a four player game, four NFSP-agents compared to one would significantly increase the computation time. In addition, it might not be helpful for the optimization process to initialize four agents with the same hyperparameter sets because of unpredictable interplay. For these reasons, it was decided against this approach.

6.1.2 Base of Comparison

As a base of comparison, two metrics were considered.

The first metric is the TIOT. Here, the problem occurs of having to compare two different reward bases with contrasting value ranges. To solve this problem and make the slopes comparable, all values were normalized in the range of the maximum and minimum possible values relative to the reward system specified.

The second metric is the APPG, a resulting sampled model receives, when comparing the model in a tournament against random agents. When the training of a model terminates, the final model plays 1,000 tournament games against random agents, each with the random seeds 12, 17, 20, 30, and 33.

From each metric, a ranking can be generated. The resulting final rank is created by weighting the rank of the TIOT and the rank of the APPG with the factor of 0.5 to produce an average ranking. The model-parameters that received the highest average rank are used for the final training of a representing model for the considered algorithm.

6.2 HPO of DQN

6.2.1 Searching Space

For the random search, the following searching space was defined:

- ***Replay Memory Size (int)***: $50000, 100000, 200000$
- ***Replay Memory Init Size (int)***: 100
- ***Update Target Estimator Every (int)***: $1000, 2000, 10000$
- ***Discount Factor (float)***: $0.75, 0.8, 0.95, 0.99$
- ***Epsilon Start (float)***: 1
- ***Epsilon End (float)***: $0.1, 0.05, 0.01$
- ***Epsilon Decay Steps (int)***: 50000
- ***Batch Size (int)***: $32, 64$

- ***MLP Layer (list)***: $[512, 512, 512], [512, 256, 128], [512, 512]$
- ***Learning Rate (float)***: $1 \cdot 10^{-4}, 5 \cdot 10^{-5}, 1 \cdot 10^{-5}, 5 \cdot 10^{-6}$

6.2.2 Search Results

After training 20 models of hyperparameter samples for each of the defined reward systems, the models that use the second Variant (see 4.3.2) in general performed better. Table 7 shows the five best results. The APPG was rounded to two decimals and the TIOT to three decimals, including a multiplier of 10^{-4}. Model 19, with the reward system 2, performed the best overall.

Rank	Model Name	Average Rank	APPG	TIOT
1	Model 19 - Reward System 2	3	43.02	$4.276 \cdot 10^{-4}$
2	Model 9 - Reward System 2	5.5	42.75	$3.428 \cdot 10^{-4}$
3	Model 3 - Reward System 2	6.5	43.10	$2.722 \cdot 10^{-4}$
4	Model 7 - Reward System 2	8	42.45	$3.266 \cdot 10^{-4}$
4	Model 11 - Reward System 1	8	41.74	$5.003 \cdot 10^{-4}$
...	...	...	...	...

Table 7: Random Search – Results of the Best Five DQN Models
(Models are numbered from 0 to 19 per reward system)

6.3 HPO of NFSP

Initial experiments demonstrated that NFSP is more challenging to hyperparameter tune compared to DQN. A reason for that is the greater number of available hyperparameters. Conveniently, the implementation of NFSP in RLCard combines an RL network that implements DQN and an SL average policy network.

In order to reduce the number of hyperparameters to optimize, the already optimized parameters in Section 6.2 are taken for the DQN and only the NFSP-specific hyperparameters are tuned.

6.3.1 Searching Space

The searching space for the HPO of the NFSP network was defined as follows:

- ***Hidden Layers Sizes (list)***: $[128, 128], [256, 256], [512, 512], [512, 512, 512]$

- ***Reservoir Buffer Capacity (int)***: $20000, 50000, 100000, 200000$
- ***Anticipatory Param (float)***: $0.1, 0.2, 0.25, 0.35, 0.5$
- ***Batch Size (int)***: $128, 64, 32$
- ***SL Learning Rate (float)***: $1 \cdot 10^{-3}, 1 \cdot 10^{-4}, 5 \cdot 10^{-5}, 1 \cdot 10^{-5}, 5 \cdot 10^{-6}$
- ***Min Buffer Size To Learn (int)***: 100

6.3.2 Search Results

Table 8 presents the five best results. Again, there is a clear winner. Similar to the DQN results, models that use the second variant perform better. This time, Model 5 of the models that used the second reward system performed the best.

Rank	Model Name	Average Rank	APPG	TIOT
1	Model 5 - Reward System 2	6.5	42.25	$2.55 \cdot 10^{-4}$
2	Model 2 - Reward System 2	9.5	42.06	$2.485 \cdot 10^{-4}$
3	Model 10 - Reward System 2	11.0	42.13	$2.278 \cdot 10^{-4}$
4	Model 19 - Reward System 2	11.5	42.01	$2.292 \cdot 10^{-4}$
5	Model 9 - Reward System 2	13.0	42.08	$2.104 \cdot 10^{-4}$
...	...	...	...	...

Table 8: Random Search – Result of the Best Five NFSP Models (Models are numbered from 0 to 19 per reward system)

6.4 Result of the HPO

The HPO demonstrated that, under the defined circumstances, the second reward system performed better when training models. Visual graphs of the training results are shown in Appendix D. In conclusion, through the process of random search, it was possible to find hyperparameters for the algorithms DQN and NFSP that can maximize the expenses of training.

7 Comparison and Training

7.1 Comparison

This section compares the algorithms introduced in Section 3.4, DQN, NFSP, and DMC with each other. For each algorithm, a comparison model is initially trained. Instead of comparing models for all sub-problems, the comparison will focus on the sub-game Standard-Cego. In addition, the heuristics introduced in Subsection 4.5.1 are used for training.

After training, the models are compared with each other in two evaluation stages discussed later in this section.

7.1.1 Training Conditions for Model Comparison

For training DQN and NFSP, the first agent is defined as the algorithm and the other agents as random agents. The models were trained in steps of 50,000 episodes. After 50,000 episodes, the APPG of all the training checkpoints is calculated and compared with the APPG of the 50,000 episodes before. If the average is lower, the training terminates. The comparison is then made with the model of the 50,000 steps before, where no significant decline in APPG occurred. The hyperparameters determined in Chapter 6 are used for training. The evaluations of the training progress occurred every 1,000 episodes over 1,000 evaluation games.

For the training of DMC, the hyperparameters introduced by Zha et al. [10] are utilized. All four agents for each player are trained. This can be done with only marginal decreases in computation time because the implementation of DMC trains the agents in parallel.

Checkpoints of the model weights are created every 100 minutes during the training process. The checkpoints serve to visualize training progress. For every checkpoint of the player 1 model, a tournament consisting of 1,000 games is conducted against random agents and the resulting APPG is saved as a progress point. The random seed for the tournament environment is 15.

7.1.2 DQN Training

Table 9 defines the hyperparameters used for training. The size of the *epsilon decay steps* parameter was intentionally adjusted to account for longer training durations.

Hyperparameter	Value
Random Seed	20
Replay Memory Size	100,000
Replay Memory Init Size	100
Update Target Estimator Every	10,000
Discount Factor	0.95
Epsilon Start	1
Epsilon End	0.1
Epsilon Decay Steps	100,000
Batch Size	32
MLP Layer	[512, 512]
Learning Rate	10^{-5}

Table 9: Final Hyperparameters for DQN Training

After 450,000 episodes, the training terminated. Figure 9 visualizes the training progress until episode 400,000, when the model was improving. The shown training progress accounted for 35,244,044 time steps in total. The figure shows a heavy increase in APPG in the early training steps from about 40.5 to 43 until around 500,000 time steps. After that, the average tournament reward increased marginally from about 43 to 43.5, with fewer improvements the longer the training progressed.

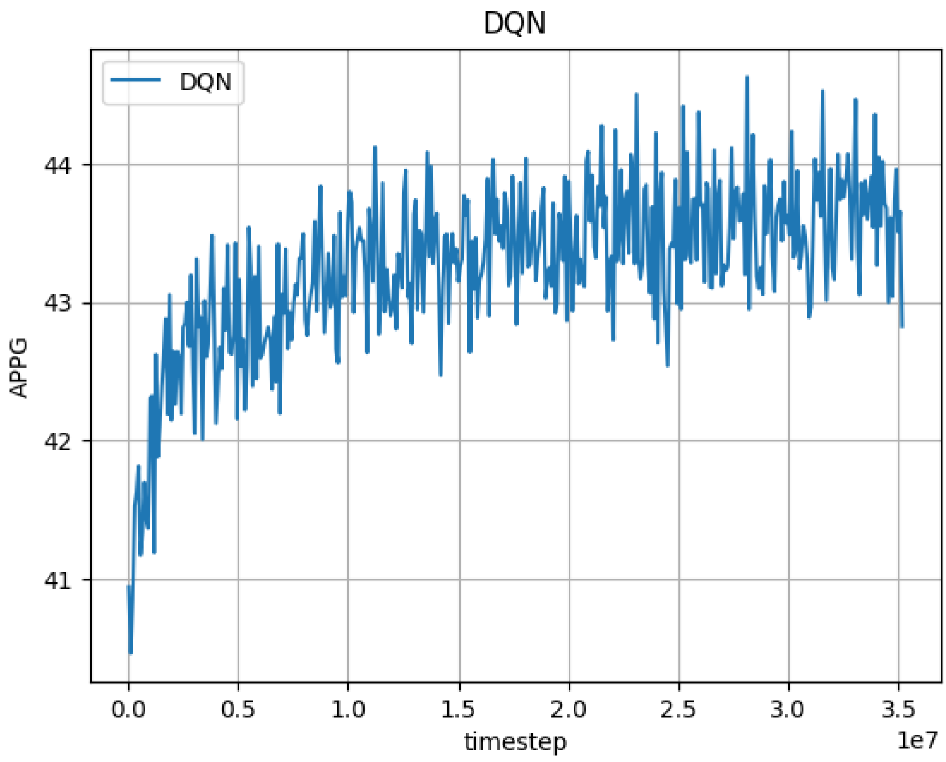


Figure 9: DQN Training Progress

7.1.3 NFSP Training

All defined hyperparameters used for training can be seen in Table 10. Same as in DQN, the *epsilon decay steps* parameter was adjusted to account for the higher number of training episodes.

Hyperparameter	Value
Random Seed	20
SL Hidden Layers Sizes	[512, 512]
SL Reservoir Buffer Capacity	100000
Anticipatory Param	0.5
SL Batch Size	32
SL Learning Rate	10^{-4}
SL Min buffer size to learn	100
RL Replay Memory Size	100,000
RL Replay Memory Init Size	100
RL Update Target Estimator Every	10,000
RL Discount Factor	0.95
RL Epsilon Start	1
RL Epsilon End	0.1
RL Epsilon Decay Steps	100,000
RL Batch Size	32
RL MLP Layer	[512, 512]
RL Learning Rate	10^{-5}

Table 10: Final Hyperparameters for NFSP Training

The training process terminated after 250,000 episodes. Studying the training progress (see Figure 10) after 200,000 episodes and 17,644,044 time steps, comparable observations of the DQN training progress (see Figure 9) can be made with a similar improvement graph over fewer episodes. Other than DQN, the APPG has more variance and maxes out at an average of around 42.5.

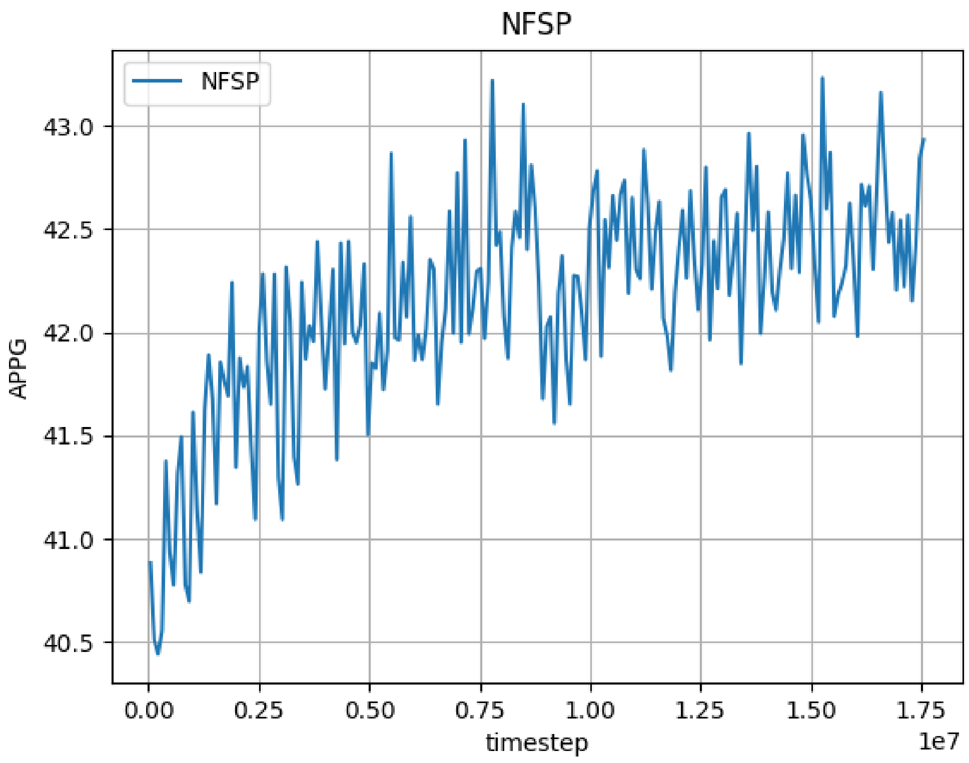


Figure 10: NFSP Training Progress

7.1.4 DMC Training

Table 11 highlights the hyperparameters used for the training with DMC. The number of *total frames* was adjusted to $2.5 \cdot 10^9$ because of hardware and time restrictions.

Hyperparameter	Value
Random Seed	20
Num Actor Devices	1
Num Actors	5
MLP Size	[512, 512, 512, 512, 512]
Total Frames	$2.5 \cdot 10^9$
Exp Epsilon	0.01
Batch Size	32
Unroll Length	100
Num Buffers	50
Num Threads	4
Max Grad Norm	40
Learning Rate	0.0001
Alpha	0.99
Momentum	0
Epsilon	0.00001

Table 11: Final Hyperparameters for DMC Training

In Figure 11, high improvements can be seen until around checkpoint model 30. After that point, the improvements between models are marginal. After around checkpoint 60 the observed APPG fluctuates in the range of 46 and 47. Overall, a higher APPG is reached compared to the training progressions of DQN (see Figure 9) and NFSP (see Figure 10).

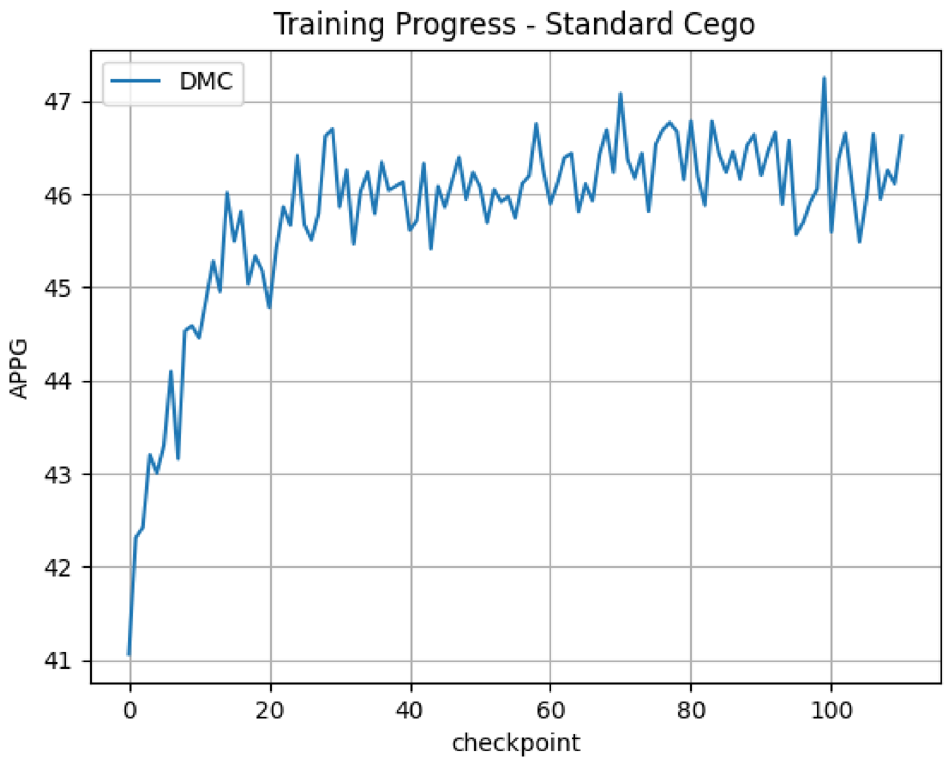


Figure 11: DMC Training Progress

7.1.5 Base of Comparison

The comparison of the algorithms consists of two phases. Each phase consists of 250,000 games split into groups of 50,000 games with the random seeds 12, 17, 20, 30, and 33. When all games are played, the calculated APPG of agents is saved and compared.

The first phase compares each model in a vacuum. Trained models play against random agents.

The second phase compares every model of Player 1 with every other model of Player 1 following the first phase. Considering an example where the first model is denoted A and the second model is denoted B, the player positions are as follows:

- Player 1: *A*,
- Player 2: *B*,
- Player 3: Random agent,
- Player 4: Random agent.

When considering this configuration for 250,000 games, it would result in the problem that both *A* and *B* were trained as players one, but *B* would have to take the disadvantageous position. For this reason, at first, this configuration is played for 250,000 games and afterward the positions of *A* and *B* are swapped and another 250,000 games are conducted. The result provides a comparison where both models played an even number of games in both advantaged and disadvantaged positions.

7.1.6 Phase 1 – Results

The results in the first phase (see Table 12) illustrate that DMC scored the highest APPG by an average margin of 2.619 points per game more compared to the next best algorithm, DQN. DMC also scored the highest WP with 69.958%. The last place is occupied by NFSP with an APPG difference of an average -1.334 points per game and a WP of 57.658%. Overall, DMC performed the best out of all algorithms considered in the first phase under the specified conditions.

Algorithm	APPG	WP
DMC	46.462	69.958
DQN	43.843	62.761
NFSP	42.509	57.658

Table 12: Evaluation Phase 1 – Results
(All values rounded to three decimal places.)

7.1.7 Phase 2 – Results

Table 13 represents the results of the second phase. The Displayed *APPG* is from the model A perspective. DMC scores the highest against both DQN and NFSP. It is also observable that, over the specified sample size, the APPG aligned well with the WP. The best result was achieved by DMC against NFSP with an APPG of 46.911 and a WP of 71.529%.

B / A	DQN		DMC		NFSP	
	APPG	WP	APPG	WP	APPG	WP
DQN	-	-	42.133	57.115	44.033	63.506
DMC	46.54	70.139	-	-	46.787	70.85
NFSP	42.520	57.94	40.771	51.871	-	-

Table 13: Evaluation Phase 2 – Results
(All values rounded to three decimal places.)

Table 14 combines the results observed in Table 13 and calculates the average of all games in phase 2 for each algorithm. Over all games played in phase 2, DMC achieved the highest score. It should be noted that DMC scores better against NFSP or DQN than against all random agents (see Figure 12). NFSP has placed last in terms of APPG and WP. Additionally, the value difference between DQN and NFSP is smaller than the difference between DMC and DQN.

Algorithm	APPG	WP
DMC	42.105	58.001
DQN	38.777	48.136
NFSP	37.618	43.864

Table 14: Evaluation Phase 2 – Results Relative to Each Algorithm
(All values rounded to three decimal places.)

7.1.8 Discussion of Results

When observing the training progress, all algorithms are able to improve and produced better average values than random agents (see Section 5.4).

Despite NFSP being an MRL algorithm and DQN being a single agent RL algorithm, DQN is able to overall perform better than NFSP within an MRL environment. This further proves the versatility and adjustability of DQN.

In both phases, DMC is the winner and therefore scores the highest number of points within the specified boundaries. This indicates that DMC out of the tested algorithms achieves the highest level of effectiveness in the problem domain. Because of these findings, this work continues training models for the other sub-games of Cego with DMC.

It should be noted that the results are dependent on the sub-game that was examined. The position of the trained player can also affect the results of this evaluation. To fully prove the superiority of DMC, the HPO would have to be repeated for each player and sub-game for all algorithms for a complete comparison within the domain Cego. This is outside the scope of this thesis. Furthermore, **henderson_deep_2019** emphasized the substantial effect the initial random seed can have on the training results of RL algorithms **henderson_deep_2019**. For this reason, these results should be viewed in the context of this comparison's random seeds. Nevertheless, one advantage that is not displayed in the results is the parallel train of DMC. The algorithm was able to train multiple player models at the same time, whereas other algorithms might require multiple training runs with different hyperparameters.

7.2 Training AI Models for All Sub-Games

7.2.1 Training Conditions

The training of DMC has proven to be time-consuming and, as can be seen in Figure 11, the training increases at later stages are minimal. The decision was made to reduce the number of total frames from $2.5 \cdot 10^9$ to $1.5 \cdot 10^9$ for the training of the other sub-games. Whenever possible, heuristics are taken into account (see Subsection 4.5.1) for the game selection in each sub-game. In addition to the changes mentioned above, the hyperparameters defined in Subsection 7.1.4 are used for training.

According to these conditions, for each sub-game, four models were trained, one for each player.

7.2.2 Evaluation Process

To analyze the performance of the models, the following experiments are performed. For each sub-game, four tournaments are conducted with the following setups:

1. Player 1 = DMC model; Other players = random agents
2. Player 2 = DMC model; Other players = random agents
3. PLayer 1 = DMC model; Other team = 1 DMC model + 2 random agents
4. All players = DMC models

The goal of all setups is to create a full picture of the AI dynamics in a sub-game. The first two setups analyze the dynamic of a single AI in both single-player and non-single-player positions against random agents. The last two setups analyze the dynamics of setups with multiple AI players.

Because in Räuber, the variance between setups is lower, because each player plays on their own, the number of tournaments is reduced to three setups:

1. Player 1 = DMC model; Other player = random agents
2. PLayer 1 = DMC model; Other team = DMC model + random agents
3. All players = DMC models

For each setup, 50,000 games are played with the random seeds 31, 43, 67, 78, and 112, resulting in a total of 250,000 games per setup. Over all tournaments, the APPG is calculated when it is part of the sub-game specifications, otherwise, the WP is determined.

7.2.3 Evaluation Results

Table 15 presents the results of the evaluation. The winning side of each simulation is marked in bold. The results are compared to the benchmarks defined in subsection 5.4 with heuristics considered.

As can be seen in the 1. setup result of Standard-Cego, the single-player AI achieves a significantly better APPG. The single-player AI has an advantage when playing against a non-single-player AI. However, when playing against all AI agents, the single-player AI has a disadvantage. A non-single-player AI slightly increases the APPG compared to the benchmark when competing against random agents.

Against random agents, the single-player receives an APPG of 54.26. The advantage in all setups stays with the single-player. A game of all AI players reduces the APPG of the single-player down to 44.93.

In the game mode Ultimo, the single-player has a WP increase of over 1000% compared to the benchmark. The WP of 94.74% decreases by around 69.42% when playing against a team of other AIs. Further analysis of this AI's behavior is presented in Appendix F. From a non-single-player position, the AI has a slightly increased WP.

When observing the result of the game mode Bettel, it can be seen that the advantage stays with the single-player, with the increases and decreases depending on the player setups. In a game of all AI agents, the advantage of the single-player decreases to a WP of around 60.78%.

Contrary to Bettel, in Piccolo, the advantage always stays at the side playing against the single-player. When the single-player AI plays against all random agents, the WP stays under 50%.

A single AI Räuber player against random agents has a chance of around 14.3% of losing. In a setup of two AIs, the WPs of the two players decrease but, in addition, the WPs of the two random agents also decrease. In a game of all AIs, the WPs are almost evenly matched.

Player Position	**Standard-Cego**: **APPG** Setups: 1.	2.	3.	4.
1	**46.48**	38.02	**43.2**	37.98
2	32.52	**40.98**	35.8	**41.03**
3	32.52	**40.98**	35.8	**41.03**
4	32.52	**40.98**	35.8	**41.03**

Player Position	**Solo**: **APPG** Setups: 1.	2.	3.	4.
1	**54.26**	**46.01**	**51.42**	**44.93**
2	24.74	32.99	27.58	34.07
3	24.74	32.99	27.58	34.07
4	24.74	32.99	27.58	34.07

Player Position	**Ultimo**: **WP** Setups: 1.	2.	3.	4.
1	**94.74**	7.61	**88.06**	**69.42**
2	5.26	**92.39**	11.94	30.58
3	5.26	**92.39**	11.94	30.58
4	5.26	**92.39**	11.94	30.58

Player Position	**Bettel**: **WP** Setups: 1.	2.	3.	4.
1	**78.94**	**67.3**	**75.79**	**60.78**
2	21.06	32.7	24.21	39.22
3	21.06	32.7	24.21	39.22
4	21.06	32.7	24.21	39.22

Player Position	**Piccolo**: **WP** Setups: 1.	2.	3.	4.
1	47.65	34.32	43.76	28.76
2	**52.35**	**65.68**	**56.24**	**71.24**
3	**52.35**	**65.68**	**56.24**	**71.24**
4	**52.35**	**65.68**	**56.24**	**71.24**

Player Position	**Räuber**: **WP** Setups: 1.	2.	3.
1	**85.70**	**82.94**	**74.46**
2	**71.52**	**83.85**	74.26
3	71.36	**66.97**	**75.02**
4	**71.41**	66.25	**76.27**

Table 15: Sub-Game Evaluation Results
(Results rounded to two decimal places)

The progress of each sub-game training can be observed in Appendix E.

7.2.4 Discussing the Results

The Ultimo single-player AI is be able to exploit its hand cards effectively. This could indicate a synergizing and reinforcing effect between the heuristic game selection and the AI.

Both Standard-Cego and Solo single-player models increase their APPG by similar substantial margins compared to the benchmarks.

The heuristic game selection for the sub-game Bettel already has led to a high increase of the WP toward the single-player, which is further increased by the results of the single-player model.

The Piccolo single-player model increases on the benchmark but, on average, wins less than half of the games. One possible reason could be that the heuristic game selection does not sufficiently increase game chances for the single-player. Further constraints might be needed to increase the WP to over 50%.

The decision of disregarding a heuristic game selection process in Räuber did not hinder the models from learning to increase their WP. Overall, the Räuber AIs achieved satisfying WP results.

In all sub-games, it could be observed that the reward increase of the single-player AI was always bigger than the reward increase of a non-single-player AI. However, in all sub-games, it can be noticed that when combining non-single-player AIs in game setups, they increase their average reward. This observation indicates some level of cooperative behavior between AI agents on the same team.

Across the board, the models improve in all sub-games over the defined benchmarks.

8 Implementation of AI-Models Into the Game Engine

8.1 Current State

Models were trained for each sub-game and player of that sub-game. The models can make a prediction based on the current game state and the legal actions provided as input (see Figure 12). Part of the game state is the ID of the current player, the cards of the player, the cards that haven't been played, the trick cards, the winning card, the winning player ID, and the ID of the player that started the trick round. This information has to be in the format described in Subsection 4.2.2. The resulting prediction is the card that should be played.

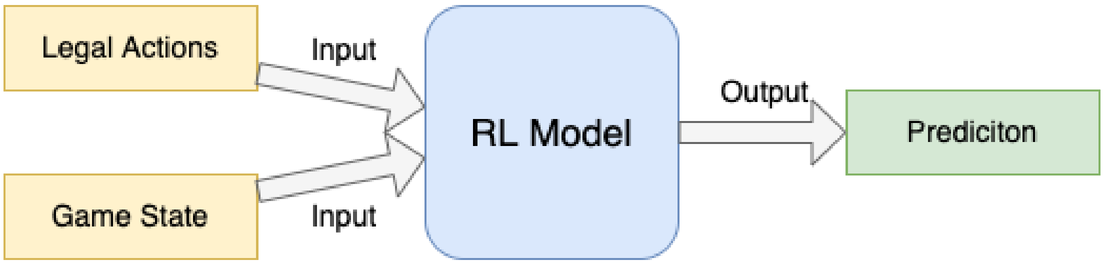


Figure 12: Utilization of RL Models

8.2 Discussing Possible Solutions

The game environment is developed in TypeScript and is running on a server. The game will be available as a web app. These conditions allow for two possible solutions.

The first possible solution would be to convert the model into a format that can be used within the web environment and directly implement it into the game. A format that allows this is Open Neural Network Exchange (ONNX) 2022, a standardized and widely supported format for ML models. The problem is that not all models may be portable to this format if they do not fall into the specified standard. Currently, RLCard models do not support this format.

Another possibility is to provide a REST API. This has the advantage of decoupling the AI from the rest of the game environment. Consequentially, this allows for easier extension, modification, and testing. A disadvantage is that for using the model, the environment that runs the API has to support Python. The server structure in use fulfills this requirement. In addition, the usage of containerization technology, such as Docker [71], can significantly reduce deployment and dependency risks [72].

In conclusion, the decision was made to implement an independent REST API because of the reasons outlined before.

8.3 Requirements of the API

The user should be able to send a request for which game mode they play and what player position they have. The API can then use the fitted model to predict an action. The user shall provide all the information about the current game state needed to make a prediction. This is to fulfill the REST property of statelessness. As a result, the player gets the prediction of the next appropriate move to play.

8.4 Deciding on a Framework

In the next step, a Python framework is selected that facilitates the implementation of a REST API. The frameworks of discussion are *Django REST Framework* [73], *Flask* [74], and FastAPI [75].

Django REST Framework uses the core of the Web Framework *Django* [76] and its features to expose REST API Endpoints. One feature of Django is the Model View Template (MVT)[1] paradigm [77], [78]. In addition, it comes with many built-in functionalities, such as database querying and rendering. Unfortunately, these features are not necessary for this particular problem and therefore a framework with more flexibility and fewer pre-implemented features is more well suited.

[1] In the MVT paradigm the **M**odel describes the datamodel, the **V**iew handles the request logic, and a **T**emplate is defined for dynamic site generation [77]

In contrast to Django, where in some cases certain solutions to specific problems might be preferred [78], Flask puts the responsibility on the developer to solve project-specific problems [74], [79]. Thus, it is flexible in how certain tasks are solved. Flask is scalable and functionalities can be added through extensions when needed. However, this high level of flexibility might hinder the development of best practice solutions that fulfill all requirements with few to no bugs.

FastAPI's features are its simple design that enhances the fast implementation of features, and editor support for rapid development [75]. But its main advantage over the two other frameworks is that it is considerably faster and more efficient [80]. Despite its young age, it has already been used by known tech companies, such as Netflix [81] and Uber [82]. In addition, the simplicity of FastAPI fits the purpose of a small and structured API.

Overall, FastAPI fits the requirements of the API the closest. It has various advantages over other contemporary frameworks, and therefore it was selected as the framework of choice.

8.5 API Design

8.5.1 Endpoints

For each sub-game, an endpoint was defined. By doing so, the API user can distinguish between the sub-games. Each endpoint is reachable by an HTTP GET request. The endpoints and their associated sub-games are represented in Table 16.

Endpoint	Sub-Game
/cego/	Standard-Cego
/solo/	Solo
/ultimo/	Ultimo
/bettel/	Bettel
/piccolo/	Piccolo
/raeuber/	Räuber

Table 16: Mapping of Sub-Games to Endpoints

Alternatively, all the endpoints could have been implemented as POST requests. The advantage of that approach would be that a request body in, for example, JSON format could be sent over to the server. Except, this would be contrary to the usual REST API CRUD operation approach. The nature of the request is a read operation. The user tries to get information from the server about what move to play next. Therefore, GET requests are appropriate.

Thoughts on how this API could be implemented, in combination with a rule-based bidding phase, are provided in Appendix G.

8.5.2 Providing the Game State

To make a prediction, the game state has to be provided. This is done by query parameters that are attached to the Uniform Resource Locator (URL). The following query parameters are supported:

- ***hand_cards (required)***: Contains all the hand cards of the player.
- ***played_cards***: Contains all cards that have been played.
- ***trick_cards***: Contains all the cards in the trick. Provided in the order they were played.
- ***legage_cards***: Contains all cards that are part of the Legage.
- ***round_starter_id (required)***: The index of the player who started the trick round (integer in the range $[0 \leq x \leq 3]$).
- ***current_player_id (required)***: The index of the player who has to play a card (integer in the range $[0 \leq x \leq 3]$).
- ***single_player_id***: The index of the player who plays on their own (integer in the range $[0 \leq x \leq 3]$), **or**
- ***raeuber_id***: The index of the player who called *Räuber* if the sub-game is played (integer in the range $[0 \leq x \leq 3]$).

Two of these parameters are sub-game specific. The *legage_cards* parameter is only required when the endpoint */cego/* was called. In the case of calling the endpoint */raeuber/*, the *single_player_id* must be substituted with the *raeuber_id*.

Information about the card sets, such as in *hand_cards* or *trick_cards*, is provided in a human-readable way. Each card is assigned a code, and multiple card codes are separated by a comma. For example, a valid declaration of the hand cards would be "$[URL]/cego/?hand\_cards = 1 - trump, 17 - trump, b - h\&...$". The valid card codes accepted by the API are represented in Appendix C.

8.5.3 Prediction Function

All endpoints have similar requirements. Their only difference is the parameters they require. Therefore, a *predict* function was abstracted that maps the request to the correct player model, obtains the unformatted game state as input, and produces the prediction.

Internally, this function loads the correct model. Then it encodes the unformatted game state into a numerical array and analyzes the legal moves. After this, the model can be fed with the encoded game state and determined legal actions in order to create a prediction. The prediction is then returned as a string value.

8.5.4 Formatting the Game State

To make a prediction, the model expects the game state in the correct format. This game state can be described as a key-value map with the following keys:

- ***obs***: The game state as a numerical array.
- ***legal_actions***: a map of action indexes, where the predicted action values can be attached to.
- ***raw_legal_actions***: The list of valid actions in code form.

In the same way that the game implementation encodes the game state (see subsection 4.2.2), the API does so as well. The query parameters provide all the information needed to generate it. As an example, playable cards can be evaluated based on hand cards, trick cards, played cards, and Legage cards. In addition, the winner ID of the player does not have to be included in the request. The winner and winner card evaluation occurs on the API side by evaluating the trick cards (see Subsection 4.4.1). The API also takes care of the evaluation of legal actions (see Subsection 4.4.2).

8.5.5 Response Generation

After the prediction function has returned the prediction in string format, the prediction is then wrapped into JSON format. For example, when the card *15-trump* was predicted, the following result is returned:

```
{
    "action": "15-trump"
}
```

8.6 Validation

To ensure the correct usage of the API, the following validation rules are enforced:

- The hand cards must be provided.
- The IDs of the current player and the player who started the trick round are necessary.
- The starting player ID has to be provided when the game mode is not Räuber
- If the game mode is Räuber, the ID of the player who called the mode has to be provided.
- All card codes have to be valid and, therefore, are encodable into indices.
- If the sub-game is Ultimo, the hand deck has to include *1-trump*.
- The number of hand cards provided can't exceed 11.

In case validation is not met, the HTTP status code 400 (Bad Request) is returned, otherwise, the HTTP status code is 200 (OK) [83].

8.7 Testing

For the API, tests were created that ensure the correct behavior of the endpoints. The test cases mainly test that the validations defined in Section 8.6 behave the expected way and that all endpoints return the correct status code. For testing, the FastAPI *TestClient* [75] was utilized in combination with *pytest* [84].

8.8 Deployment

A Dockerfile [72] is provided for the deployment. This Dockerfile can be used to create an image with all the required dependencies. Based on this image, a container with the running API can be initialized on a server or cluster.

9 Conclusion

9.1 Summary

This work demonstrates an approach to planning and implementing a Cego environment for the training of models with deep RL algorithms. It is proposed to split Cego into sub-problems for each possible game mode. For the action encoding, it is suggested to encode each card separately. Furthermore, a state encoding recommendation is provided in the context of the RL framework RLCard. To shrink the number of possible game states for each sub-problem, this work defines rules for each game mode based on heuristic knowledge.

To offer a better understanding of probabilities in the game of Cego, this thesis performed simulations of the game environment. With these simulations, it was possible to approximate card win probabilities. These card win approximations were used to formulate a definition of high and low cards in the domain of Cego. In addition, benchmarks for the various sub-games were determined.

Furthermore, this work hyperparameter optimized the algorithms DQN and NFSP for the game mode Standard-Cego. Models on the sub-game of Standard-Cego were trained with optimized hyperparameters for one player position with the algorithms DQN, NFSP, and DMC. Afterward, the trained models were compared. It is concluded that the DMC agent outperform the agents trained with the other two algorithms. DQN and NFSP have similar performances, with DQN performing slightly better.

After identifying DMC as the algorithm that works best under the defined parameters. Models for the left sub-games were trained. Every trained model provides a significant increase over the average reward benchmarks against random agents. The model for the sub-game Ultimo achieves the highest increase in performance and reaches a WP of almost 95% from the single-player perspective. All sub-game models, but the Piccolo AI, achieve a higher than average APPG or WP from the single-player perspective relative to the point system.

As the final step, this work implemented a REST API that defines endpoints for each sub-game. This API allows sending a game state and receiving the predicted action that leads to the best potential outcome.

9.2 Outlook

This thesis mainly focused on the diverse sub-games of Cego. In future work, the problem of the bidding phase shall be discussed in more detail. The bidding phase can be entirely rule-based. It can also be considered training a bidding phase model based on SL algorithms. An approach like this can potentially classify beginning game states into certain sub-games, depending on the expected reward that each game mode is able to provide.

Only limited groundwork has been done on the understanding of probabilistic behaviors in Cego. Much game knowledge relies on heuristics. Statistical analysis of the game is needed in order to create quantitative knowledge for more specialized research questions.

Further improvements can also be made for the game mode of Standard-Cego. Currently, the environment generalizes the bidding levels of Standard-Cego into one mode for the purpose of simplifying the problem. An interesting question would be whether the mechanic of keeping cards and discarding cards could be automated and realized within an RL approach. The focus of improvement would be on encoding and dealing with an enlarged action space. However, the DMC algorithm has already proven that it can deal with large action spaces [10].

In addition, followup work shall provide more comparisons of different algorithms outside the scope of this thesis. Moreover, revisiting the hyperparameters of DMC and providing hyperparameter optimization with modern hyperparameter optimization methods, such as Bayesian optimization techniques [85] could lead to better performing models. The results have demonstrated that the success of training can vary for different sub-games. More specifically, the model of Piccolo needs more improvements in order to win more than half of the games from the single-player perspective. Based on the findings, it is proposed to optimize the hyperparameters for each sub-game separately with the goal of maximizing each problem.

Another research question that hasn't been answered yet is whether the AI can beat a human player and, if so, what skill level of players the AI is able to beat. Further research and evaluation will be needed to answer this question. First tests with real players are planned in the near future for the AI implementation within Cego Online 2.0. So, first feedback and testing results may follow soon.

9.3 Broader Impact

In comparison to other German card games like *Skat*, the AI research on the game of Cego has been sparse. This thesis provides the first-ever RL-based AI for the game that might challenge professional players.

The final model is based on the algorithm, DMC, which is recent and hasn't been tested much in other imperfect information games. Indeed, this work proves that DMC can be implemented in other card games and additionally indicates similar satisfying results in games, such as Cego. Furthermore, this work proves that the implementation of DMC in real-life problems with card game-like properties may be feasible in future work.

I encourage researchers to test their skill level in Cego against the AI implementation in Cego Online 2.0 when it is released. Researchers will be encouraged to provide their studies on the game and to challenge the findings of this project.

References

[1] Gerold Blümle. “Das badische Nationalspiel Cego.” In: (2016), p. 16. URL: https://www.cego-online.de/tl_files/documents/Zegolang160414.pdf.

[2] Jirka Dell’Oro-Friedl, Martin Wangler, Nicole Dretvic, et al. *Cego-Online: Startseite - Cego online.* 2013. URL: https://www.cego-online.de/startseite.html (visited on 05/15/2022).

[3] Adobe. *Adobe Flash Player End of Life.* 2021. URL: https://www.adobe.com/products/flashplayer/end-of-life.html (visited on 07/24/2022).

[4] Cego Team. *Entwicklung gestartet!* Entwicklung gestartet! 2021. URL: https://cego-reloaded.blogspot.com/2021/03/entwicklung-gestartet.html (visited on 07/24/2022).

[5] David Silver, Aja Huang, Chris J. Maddison, et al. “Mastering the game of Go with deep neural networks and tree search.” In: *Nature* 529.7587 (Jan. 28, 2016), pp. 484–489. ISSN: 0028-0836, 1476-4687. DOI: 10.1038/nature16961. URL: http://www.nature.com/articles/nature16961 (visited on 07/29/2022).

[6] David Silver, Julian Schrittwieser, Karen Simonyan, et al. “Mastering the game of Go without human knowledge.” In: *Nature* 550.7676 (Oct. 2017). Number: 7676 Publisher: Nature Publishing Group, pp. 354–359. ISSN: 1476-4687. DOI: 10.1038/nature24270. URL: https://www.nature.com/articles/nature24270 (visited on 07/29/2022).

[7] David Silver, Thomas Hubert, Julian Schrittwieser, et al. “Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm.” In: (2017). Publisher: arXiv Version Number: 1. DOI: 10.48550/ARXIV.1712.01815. URL: https://arxiv.org/abs/1712.01815 (visited on 08/24/2022).

[8] M. Ginsberg. “GIB: Steps Toward an Expert-Level Bridge-Playing Program.” In: *IJCAI.* 1999.

[9] Michael Bowling, Neil Burch, Michael Johanson, et al. “Heads-up limit hold’em poker is solved.” In: *Science* 347.6218 (Jan. 9, 2015). Publisher: American Association for the Advancement of Science, pp. 145–149. DOI: 10.1126/science.1259433. URL: https://www.science.org/doi/10.1126/science.1259433 (visited on 07/29/2022).

[10] Daochen Zha, Jingru Xie, Wenye Ma, et al. “DouZero: Mastering DouDizhu with Self-Play Deep Reinforcement Learning.” In: *arXiv:2106.06135 [cs]* (June 10, 2021). arXiv: 2106.06135. URL: http://arxiv.org/abs/2106.06135 (visited on 04/10/2022).

[11] Manfred Quambusch. *Gewinnen beim Skat gläserne Karten.* OCLC: 75456320. 1990. ISBN: 978-3-7919-0353-8.

[12] Peter Lincoln. *Skat lernen.* 1., Edition. Stuttgart: Urania, June 1, 2004. 96 pp. ISBN: 978-3-332-01510-2.

[13] Stephan Grandmontagne. *Meisterhaft Skat spielen. 1: Allgemeine Spiellehre.* Saarbrücken: Grandmontagne, 2005. 161 pp. ISBN: 978-3-00-017743-9.

[14] Dr Siegfried Harmel. *Skat-Zahlen.* Dr. Siegfried Harmel, Jan. 1, 2016.

[15] Sebastian Kupferschmid and Malte Helmert. “A Skat Player Based on Monte-Carlo Simulation.” In: *Computers and Games.* Ed. by H. Jaap van den Herik, Paolo Ciancarini, and H. H. L. M. (Jeroen) Donkers. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2007, pp. 135–147. ISBN: 978-3-540-75538-8. DOI: 10.1007/978-3-540-75538-8_12.

[16] Timothy M. Furtak. *Symmetries and Search in Trick-Taking Card Games.* ERA. 2013. DOI: 10.7939/R3H98ZQ52. URL: https://era.library.ualberta.ca/items/df9f1055-df09-43ed-910b-38da4e9d0792 (visited on 07/29/2022).

[17] Stefan Edelkamp. “Challenging Human Supremacy in Skat.” In: *Proceedings of the International Symposium on Combinatorial Search* 10.1 (2019). Number: 1, pp. 52–60. ISSN: 2832-9163. URL: https://ojs.aaai.org/index.php/SOCS/article/view/18502 (visited on 07/29/2022).

[18] Matthias Plappert. *Keras-RL.* Publication Title: GitHub repository. 2016. URL: https://github.com/keras-rl/keras-rl.

[19] Alexander Kuhnle, Michael Schaarschmidt, and Kai Fricke. *Tensorforce: a TensorFlow library for applied reinforcement learning.* Published: Web page. 2017. URL: https://github.com/tensorforce/tensorforce.

[20] Eric Liang, Richard Liaw, Philipp Moritz, et al. “RLlib: Abstractions for Distributed Reinforcement Learning.” In: (2017). Publisher: arXiv Version Number: 4. DOI: 10.48550/ARXIV.1712.09381. URL: https://arxiv.org/abs/1712.09381 (visited on 08/24/2022).

[21] Jason Gauci, Edoardo Conti, Yitao Liang, et al. "Horizon: Facebook's Open Source Applied Reinforcement Learning Platform." In: (2018). Publisher: arXiv Version Number: 5. DOI: `10.48550/ARXIV.1811.00260`. URL: `https://arxiv.org/abs/1811.00260` (visited on 08/24/2022).

[22] Greg Brockman, Vicki Cheung, Ludwig Pettersson, et al. "OpenAI Gym." In: (2016). Publisher: arXiv Version Number: 1. DOI: `10.48550/ARXIV.1606.01540`. URL: `https://arxiv.org/abs/1606.01540` (visited on 08/24/2022).

[23] Marc Lanctot, Edward Lockhart, Jean-Baptiste Lespiau, et al. *OpenSpiel: A Framework for Reinforcement Learning in Games.* Sept. 26, 2020. arXiv: `1908.09453[cs]`. URL: `http://arxiv.org/abs/1908.09453` (visited on 08/24/2022).

[24] Daochen Zha, Kwei-Herng Lai, Songyi Huang, et al. "RLCard: A Platform for Reinforcement Learning in Card Games." In: *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20.* Ed. by Christian Bessiere. International Joint Conferences on Artificial Intelligence Organization, July 2020, pp. 5264–5266. DOI: `10.24963/ijcai.2020/764`. URL: `https://doi.org/10.24963/ijcai.2020/764`.

[25] E. N. Barron. *Game theory: an introduction.* Second edition. Wiley series in operations research and management science. Hoboken, New Jersey: John Wiley & Sons, Inc, 2013. 555 pp. ISBN: 978-1-118-53389-5 978-1-118-21693-4.

[26] Martin J. Osborne and Ariel Rubinstein. *A course in game theory.* Cambridge, Mass: MIT Press, 1994. 352 pp. ISBN: 978-0-262-15041-5 978-0-262-65040-3.

[27] Manfred J. Holler, Gerhard Illing, and Stefan Napel. *Einführung in die Spieltheorie.* 8. Auflage. Lehrbuch. Berlin: Springer Gabler, 2019. 478 pp. ISBN: 978-3-642-31963-1 978-3-642-31962-4.

[28] Yoav Shoham and Kevin Leyton-Brown. *Multiagent systems algorithmic, game-theoretic, and logical foundations.* OCLC: 932122159. 2012. ISBN: 978-0-521-89943-7 978-0-511-81165-4.

[29] Aurélien Géron. *Hands-on machine learning with Scikit-Learn and TensorFlow: concepts, tools, and techniques to build intelligent systems.* First edition. OCLC: ocn953432302. Beijing ; Boston: O'Reilly Media, 2017. 551 pp. ISBN: 978-1-4919-6229-9.

[30] Ethem Alpaydin. *Maschinelles Lernen.* 2. Auflage. De Gruyter Studium. Berlin ; Boston: De Gruyter Oldenbourg, 2019. 633 pp. ISBN: 978-3-11-061788-7.

[31] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning: an introduction*. Second edition. Adaptive computation and machine learning series. Cambridge, Massachusetts: The MIT Press, 2018. 526 pp. ISBN: 978-0-262-03924-6.

[32] Phil Winder. *Reinforcement learning: industrial applications of intelligent agents*. First edition. Beijing Boston Farnham Sebastopol Tokyo: O'Reilly, 2021. 379 pp. ISBN: 978-1-09-811483-1.

[33] Martijn van Otterlo and Marco Wiering. "Reinforcement Learning and Markov Decision Processes." In: *Reinforcement Learning: State-of-the-Art*. Ed. by Marco Wiering and Martijn van Otterlo. Adaptation, Learning, and Optimization. Berlin, Heidelberg: Springer, 2012, pp. 3–42. ISBN: 978-3-642-27645-3. DOI: `10.1007/978-3-642-27645-3_1`. URL: `https://doi.org/10.1007/978-3-642-27645-3_1` (visited on 04/12/2022).

[34] Christopher Watkins. "Learning From Delayed Rewards." In: (Jan. 1, 1989).

[35] Christopher J. C. H. Watkins and Peter Dayan. "Q-learning." In: *Machine Learning* 8.3 (May 1, 1992), pp. 279–292. ISSN: 1573-0565. DOI: `10.1007/BF00992698`. URL: `https://doi.org/10.1007/BF00992698` (visited on 07/02/2022).

[36] George W. Brown. "Iterative Solution of Games by Fictitious Play." In: *Activity Analysis of Production and Allocation*. New York: Wiley, 1951.

[37] Johannes Heinrich, Marc Lanctot, and David Silver. "Fictitious self-play in extensive-form games." In: *International conference on machine learning*. PMLR, 2015, pp. 805–813.

[38] Lucian Buşoniu, Robert Babuška, and Bart De Schutter. "Multi-agent Reinforcement Learning: An Overview." In: *Innovations in Multi-Agent Systems and Applications - 1*. Ed. by Dipti Srinivasan and Lakhmi C. Jain. Berlin, Heidelberg: Springer, 2010, pp. 183–221. ISBN: 978-3-642-14435-6. DOI: `10.1007/978-3-642-14435-6_7`. URL: `https://doi.org/10.1007/978-3-642-14435-6_7` (visited on 03/25/2022).

[39] John McLeod. *Cego - card game rules*. 2018. URL: `https://www.pagat.com/tarot/cego.html` (visited on 05/15/2022).

[40] Wolfgang Fürderer and Lorenz Gerda. *Über das Spiel Cego*. Cego-Schwarzwald E. V. 2019. URL: `http://cego-schwarzwald.com/Ueber-das-Spiel-Cego/desktop/` (visited on 05/27/2022).

[41] Gerhard Baumann and Gerold Blümle. *DAS CEGOSPIEL*. 2013. URL: `https://www.cego-online.de/tl_files/documents/CegoSpielregelndef1301224mit%20Anhang.pdf`.

[42] Achim Laber. *Cego*. 2011. URL: http://www.cego.de/ (visited on 05/15/2022).

[43] Nicholas Metropolis and S. Ulam. "The Monte Carlo Method." In: *Journal of the American Statistical Association* 44.247 (Sept. 1949), pp. 335–341. ISSN: 0162-1459, 1537-274X. DOI: 10.1080/01621459.1949.10483310. URL: http://www.tandfonline.com/doi/abs/10.1080/01621459.1949.10483310 (visited on 08/17/2022).

[44] Giuseppe Ciaburro. *Hands-On Simulation Modeling with Python: Develop simulation models to get accurate results and enhance decision-making processes*. ISBN: 9781838988654 OCLC: 1227483590. 2020.

[45] Peter L. Bonate. "A Brief Introduction to Monte Carlo Simulation:" in: *Clinical Pharmacokinetics* 40.1 (2001), pp. 15–22. ISSN: 0312-5963. DOI: 10.2165/00003088-200140010-00002. URL: http://link.springer.com/10.2165/00003088-200140010-00002 (visited on 08/13/2022).

[46] IBM. *REST-APIs*. May 27, 2021. URL: https://www.ibm.com/cloud/learn/rest-apis (visited on 08/14/2022).

[47] Arnaud Lauret. *The design of web APIs*. OCLC: on1076512763. Shelter Island, NY: Manning Publications Co, 2019. 364 pp. ISBN: 978-1-61729-510-2.

[48] Roy Thomas Fielding. "Architectural Styles and the Design of Network-based Software Architectures." PhD thesis. Irvine: University of California, 2000. URL: https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm (visited on 07/13/2022).

[49] Mark H. Massé and Mark Massé. *REST API design rulebook: designing consistent RESTful Web Service Interfaces*. Beijing Köln: O'Reilly, 2012. 94 pp. ISBN: 978-1-4493-1050-9.

[50] Brenda Jin, Saurabh Sahni, and Amir Shevat. *Designing web APIs: building APIs that developers love*. First edition. OCLC: on1052900222. Sebastopol, CA: O'Reilly Media, 2018. 217 pp. ISBN: 978-1-4920-2692-1.

[51] Daochen Zha, Kwei-Herng Lai, Yuanpu Cao, et al. "RLCard: A Toolkit for Reinforcement Learning in Card Games." In: *arXiv:1910.04376 [cs]* (Feb. 14, 2020). arXiv: 1910.04376. URL: http://arxiv.org/abs/1910.04376 (visited on 04/05/2022).

[52] Martin Zinkevich, Michael Johanson, Michael Bowling, et al. "Regret Minimization in Games with Incomplete Information." In: *Advances in Neural Information Processing Systems*. Vol. 20. Curran Associates, Inc., 2007. URL: https://papers.nips.cc/paper/2007/hash/08d98638c6fcd194a4b1e6992063e944-Abstract.html (visited on 07/04/2022).

[53] Michael Johanson, Kevin Waugh, Michael Bowling, et al. “Accelerating best response calculation in large extensive games.” In: *Proceedings of the Twenty-Second international joint conference on Artificial Intelligence - Volume Volume One*. IJCAI’11. Barcelona, Catalonia, Spain: AAAI Press, July 16, 2011, pp. 258–265. ISBN: 978-1-57735-513-7. (Visited on 07/04/2022).

[54] Michel Dekking. *A modern introduction to probability and statistics: understanding why and how*. Springer texts in statistics. London: Springer, 2005. ISBN: 978-1-84628-168-6.

[55] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, et al. “Playing Atari with Deep Reinforcement Learning.” In: *arXiv:1312.5602 [cs]* (Dec. 19, 2013). arXiv: `1312.5602`. URL: `http://arxiv.org/abs/1312.5602` (visited on 05/05/2022).

[56] Johannes Heinrich and David Silver. “Deep Reinforcement Learning from Self-Play in Imperfect-Information Games.” In: *arXiv:1603.01121 [cs]* (June 28, 2016). arXiv: `1603.01121`. URL: `http://arxiv.org/abs/1603.01121` (visited on 05/05/2022).

[57] RLCard. *RLCard: A Toolkit for Reinforcement Learning in Card Games*. original-date: 2019-09-05T12:48:01Z. July 2, 2022. URL: `https://github.com/datamllab/rlcard` (visited on 07/04/2022).

[58] Hado van Hasselt, Arthur Guez, and David Silver. “Deep Reinforcement Learning with Double Q-learning.” In: (2015). Publisher: arXiv Version Number: 3. DOI: `10.48550/ARXIV.1509.06461`. URL: `https://arxiv.org/abs/1509.06461` (visited on 08/24/2022).

[59] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, et al. “Human-level control through deep reinforcement learning.” In: *Nature* 518.7540 (Feb. 2015). Number: 7540 Publisher: Nature Publishing Group, pp. 529–533. ISSN: 1476-4687. DOI: `10.1038/nature14236`. URL: `https://www.nature.com/articles/nature14236` (visited on 07/16/2022).

[60] Jeffrey S. Vitter. “Random sampling with a reservoir.” In: *ACM Transactions on Mathematical Software* 11.1 (Mar. 1, 1985), pp. 37–57. ISSN: 0098-3500. DOI: `10.1145/3147.3165`. URL: `https://doi.org/10.1145/3147.3165` (visited on 07/21/2022).

[61] J.S. Shamma and G. Arslan. “Dynamic fictitious play, dynamic gradient play, and distributed convergence to Nash equilibria.” In: *IEEE Transactions on Automatic Control* 50.3 (Mar. 2005). Conference Name: IEEE Transactions on Automatic Control, pp. 312–327. ISSN: 1558-2523. DOI: `10.1109/TAC.2005.843878`.

[62] Vitaly Bushaev. *Understanding RMSprop — faster neural network learning*. Medium. Sept. 2, 2018. URL: `https://towardsdatascience.com/understanding-rmsprop-faster-neural-network-learning-62e116fcf29a` (visited on 07/26/2022).

[63] Joseph K. Blitzstein and Jessica Hwang. *Introduction to probability*. Texts in statistical science. Boca Raton: CRC Press/Taylor & Francis Group, 2015. 580 pp. ISBN: 978-1-4665-7557-8.

[64] Michael Johanson. "Measuring the Size of Large No-Limit Poker Games." In: (2013). Publisher: arXiv Version Number: 2. DOI: `10.48550/ARXIV.1302.7008`. URL: `https://arxiv.org/abs/1302.7008` (visited on 08/24/2022).

[65] Cegofreunde St. Georgen. *Taktik und Tipps*. cegofreundes Webseite! 2012. URL: `http://cegofreunde.jimdofree.com/taktik-und-tipps/` (visited on 05/11/2022).

[66] Qiqi Jiang, Kuangzheng Li, Boyao Du, et al. "DeltaDou: Expert-level Doudizhu AI through Self-play." In: *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence*. Twenty-Eighth International Joint Conference on Artificial Intelligence {IJCAI-19}. Macao, China: International Joint Conferences on Artificial Intelligence Organization, Aug. 2019, pp. 1265–1271. ISBN: 978-0-9992411-4-1. DOI: `10.24963/ijcai.2019/176`. URL: `https://www.ijcai.org/proceedings/2019/176` (visited on 06/11/2022).

[67] Peter C. Bruce, Andrew Bruce, and Peter Gedeck. *Practical statistics for data scientists: 50+ essential concepts using R and Python*. Second edition. OCLC: on1158315601. Sebastopol, CA: O'Reilly Media, Inc, 2020. 342 pp. ISBN: 978-1-4920-7294-2.

[68] James Bergstra and Yoshua Bengio. "Random Search for Hyper-Parameter Optimization." In: *Journal of Machine Learning Research* 13.10 (2012), pp. 281–305. ISSN: 1533-7928. URL: `http://jmlr.org/papers/v13/bergstra12a.html` (visited on 07/08/2022).

[69] Sigrún Andradóttir. "Chapter 20 An Overview of Simulation Optimization via Random Search." In: *Handbooks in Operations Research and Management Science*. Vol. 13. Elsevier, 2006, pp. 617–631. ISBN: 978-0-444-51428-8. DOI: `10.1016/S0927-0507(06)13020-0`. URL: `https://linkinghub.elsevier.com/retrieve/pii/S0927050706130200` (visited on 08/16/2022).

[70] ONNX. *ONNX.ai*. original-date: 2017-09-07T04:53:45Z. July 13, 2022. URL: `https://github.com/onnx/onnx` (visited on 07/13/2022).

[71] Docker, Inc. *Docker*. May 10, 2022. URL: https://www.docker.com/ (visited on 08/14/2022).

[72] IBM. *Docker*. June 17, 2022. URL: https://www.ibm.com/cloud/learn/docker (visited on 08/14/2022).

[73] Encode. *Django REST framework*. original-date: 2011-03-02T17:13:56Z. Aug. 14, 2022. URL: https://github.com/encode/django-rest-framework (visited on 08/14/2022).

[74] Armin Ronacher. *Flask*. original-date: 2010-04-06T11:11:59Z. Aug. 14, 2022. URL: https://github.com/pallets/flask (visited on 08/14/2022).

[75] Sebastián Ramírez. *FastAPI 0.79*. original-date: 2018-12-08T08:21:47Z. July 14, 2022. URL: https://github.com/tiangolo/fastapi (visited on 07/14/2022).

[76] Django Software Foundation. *Django*. original-date: 2012-04-28T02:47:18Z. July 14, 2022. URL: https://github.com/django/django (visited on 07/14/2022).

[77] Ben Shaw, Saurabh Badhwar, Andrew Bird, et al. *Web development with Django: learn to build modern web applications with a Python-based framework*. Birmingham: Packt Publishing, 2021. 775 pp. ISBN: 978-1-83921-250-5.

[78] MDN. *Django introduction*. 2022. URL: https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Introduction (visited on 08/14/2022).

[79] Miguel Grinberg. *Flask web development: developing web applications with Python*. 2nd edition. OCLC: on1030283575. Sebastopol, California: O'Reilly, 2018. 291 pp. ISBN: 978-1-4919-9173-2.

[80] TechEmpower. *TechEmpower Web Framework Performance Comparison*. www.techempower.com. 2022. URL: https://www.techempower.com/benchmarks/#section=test&runid=7464e520-0dc2-473d-bd34-dbdfd7e85911&hw=ph&test=query&l=dbgidb-35r&a=2 (visited on 07/16/2022).

[81] Kevin Glisson, Marc Vilanova, and Forest Monsen. *Introducing Dispatch*. Medium. Feb. 24, 2020. URL: https://netflixtechblog.com/introducing-dispatch-da4b8a2a8072 (visited on 07/16/2022).

[82] Piero Molino. *Ludwig v0.2 Adds New Features and Other Improvements to its Deep Learning Toolbox*. Uber Engineering Blog. July 24, 2019. URL: https://eng.uber.com/ludwig-v0-2/ (visited on 07/16/2022).

[83] MDN. *HTTP response status codes*. 2022. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Status (visited on 08/16/2022).

[84] Holger Krekel, Bruno Oliveira, Ronny Pfannschmidt, et al. *pytest 7.1.2*. 2004. URL: `https://github.com/pytest-dev/pytest`.

[85] Tong Yu and Hong Zhu. “Hyper-Parameter Optimization: A Review of Algorithms and Applications.” In: (2020). Publisher: arXiv Version Number: 1. DOI: `10.48550/ARXIV.2003.05689`. URL: `https://arxiv.org/abs/2003.05689` (visited on 08/24/2022).

Eidesstattliche Erklärung

Ich versichere, dass ich die vorstehende Arbeit selbständig verfasst und hierzu keine anderen als die angegebenen Hilfsmittel verwendet habe. Alle Stellen der Arbeit, die wörtlich oder sinngemäß aus fremden Quellen entnommen wurden, sind als solche kenntlich gemacht.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form in keinem anderen Studiengang als Prüfungsleistung vorgelegt oder an anderer Stelle veröffentlicht.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben kann.

Philipp Oeschger
Furtwangen im Schwarzwald, 31.08.2022

Appendix

Appendix A Cego Dictionary

This section of the appendix names and describes Cego terminologies. The original German terms are used. The terms were extracted from the following sources: [2], [41].

Gstieß	The highest trump card in the game. It usually has no number displayed.
Mond	The name of the second-highest trump card with the number 21 displayed.
Geiß	The lowest trump card that has the number 1.
Blinde	The 10 blind cards that are laid down in the middle at the beginning of the game.
Legage	The cards that in the sub-game of Standard-Cego are discarded after certain hand cards have been exchanged.
Leere	The cards that have no pictures on them.
Truck	The name of the trump cards.
Schmieren	The act of not having a card of the same color or a trump card and therefore playing any of the hand cards.
Vorhand	The player to the right of the dealer.

Appendix B Overview of All Cego Cards

Figure B1 shows an overview of all Cego cards. The cards are ordered by rank from top left to bottom right.

- ***G***: Gstieß
- ***K***: King (ger. *König*)
- ***D***: Queen (ger. *Dame*)
- ***R***: Cavalier (ger. *Reiter*)
- ***B***: Jack (ger. *Bube*)

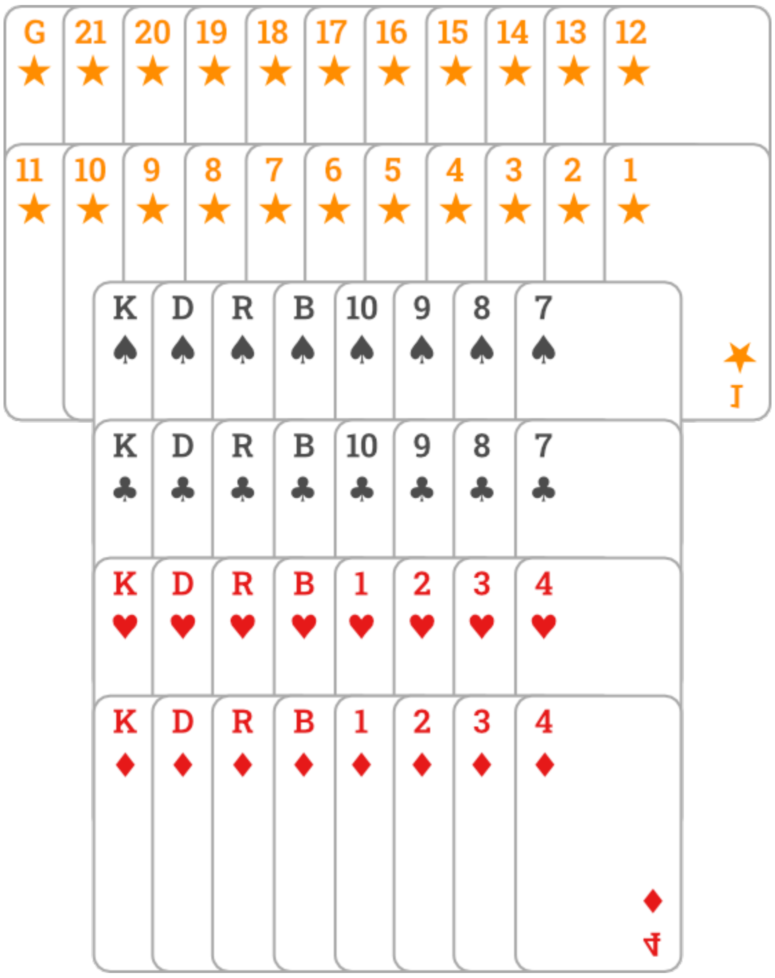


Figure B1: Cego Card Overview

Appendix C Encoding of Cards

Table C1 shows the encoding that was used for mapping cards to certain indexes for the state encoding. The keys are also the values that the user is allowed to provide to the REST API to describe the card setup of the current game state.

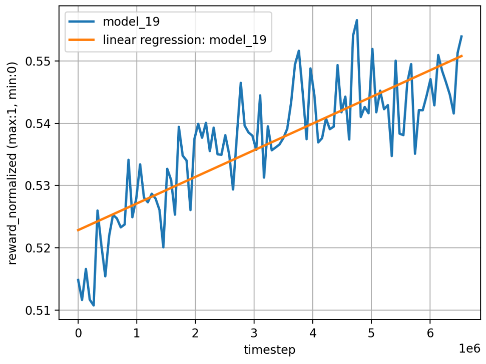
Key	Index
7-c	0
8-c	1
9-c	2
10-c	3
b-c	4
r-c	5
d-c	6
k-c	7
7-s	8
8-s	9
9-s	10
10-s	11
b-s	12
r-s	13
d-s	14
k-s	15
4-h	16
3-h	17
2-h	18
1-h	19
b-h	20
r-h	21
d-h	22
k-h	23
4-d	24
3-d	25
2-d	26
1-d	27
b-d	28
r-d	29
d-d	30
k-d	31
1-trump	32
2-trump	33
3-trump	34
4-trump	35
5-trump	36
6-trump	37
7-trump	38
8-trump	39
9-trump	40
10-trump	41
11-trump	42
12-trump	43
13-trump	44
14-trump	45
15-trump	46
16-trump	47
17-trump	48
18-trump	49
19-trump	50
20-trump	51
21-trump	52
gstiess-trump	53

Table C1: Card Encoding

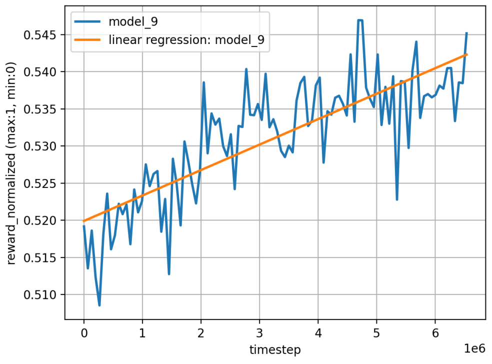
Appendix D Visualization of HPO Results

This appendix chapter visualizes the training for the five best performing models of the HPO for DQN and NFSP. Along with the training progress, the regression line is visualized. The training progresses are ordered from the highest rank (1) to the lowest rank (5). Only the first five models of each algorithm are represented.

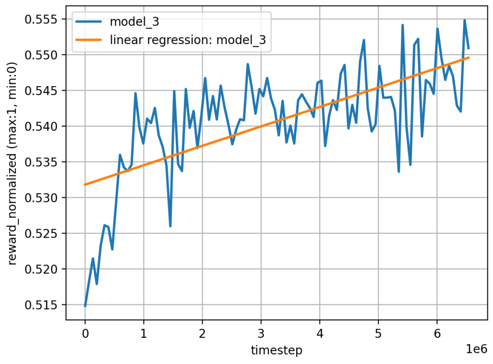
1 DQN



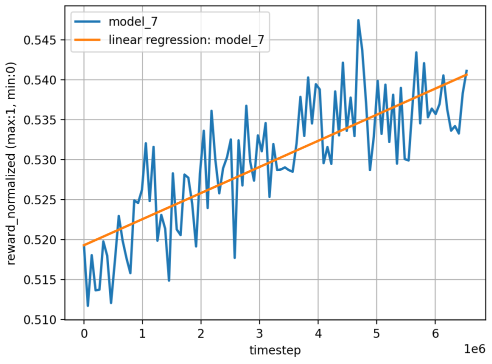
(1) Model 19 – Reward System 2



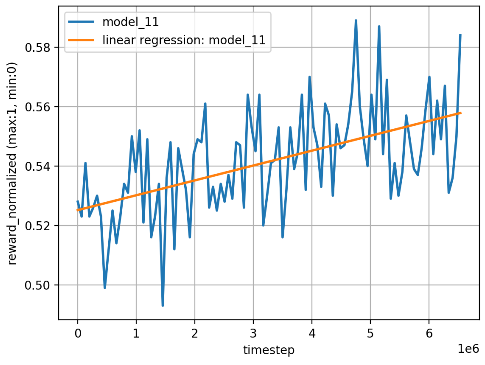
(2) Model 9 – Reward System 2



(3) Model 3 – Reward System 2



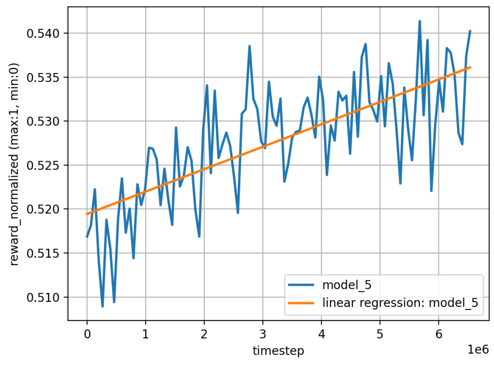
(4) Model 7 – Reward System 2



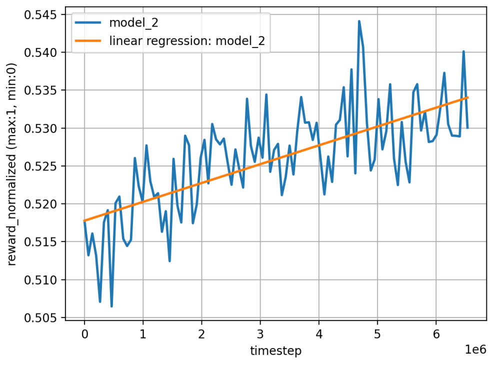
(5) Model 11 – Reward System 1

Figure D1: HPO Results – DQN

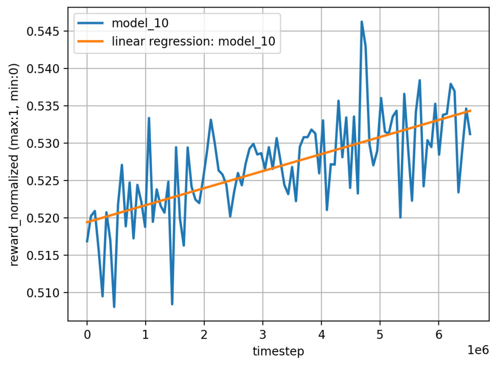
2 NFSP



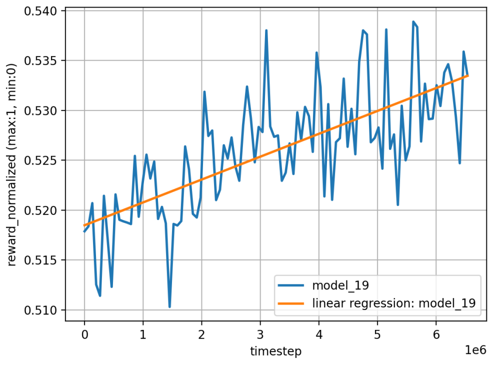
(1) Model 5 – Reward System 2



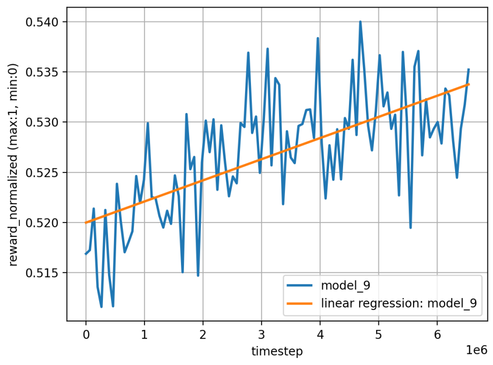
(2) Model 2 – Reward System 2



(3) Model 10 – Reward System 2



(4) Model 19 – Reward System 2

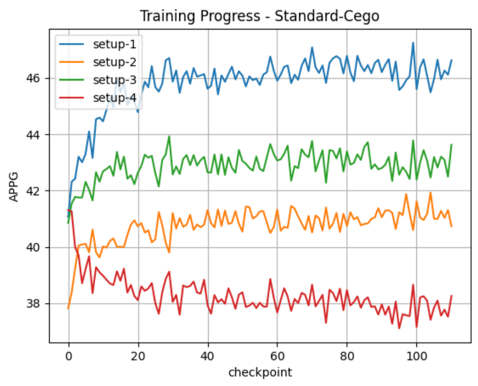


(5) Model 9 – Reward System 2

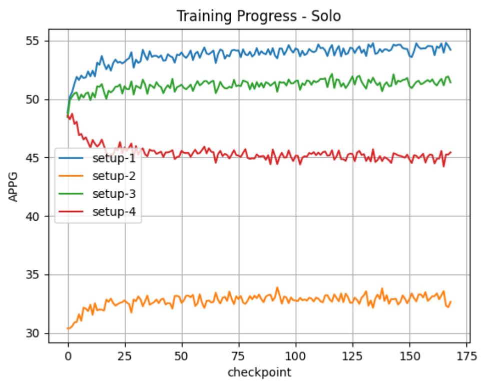
Figure D2: HPO Results – NFSP

Appendix E Progress of Sub-Game Training

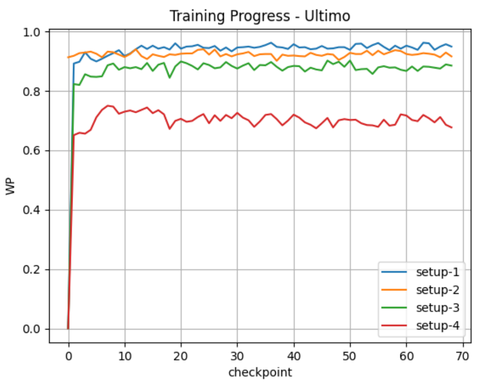
Figure E1 visualizes the training progress over the various sub-games for the different setups (see Subsection 7.2.2). The random seed used for the environment to analyze training progress is 15. Checkpoints were created after 100 minutes of training. Each progress point represents the average of 1000 evaluation game for each checkpoint. All rewards, but setup 2, are from the single-player perspective. Note that the number of checkpoints may vary because of varying training times.



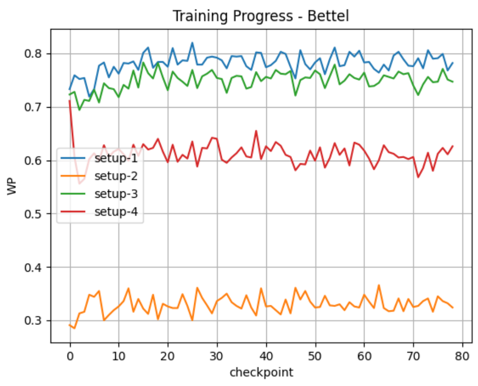
(1) Standard-Cego



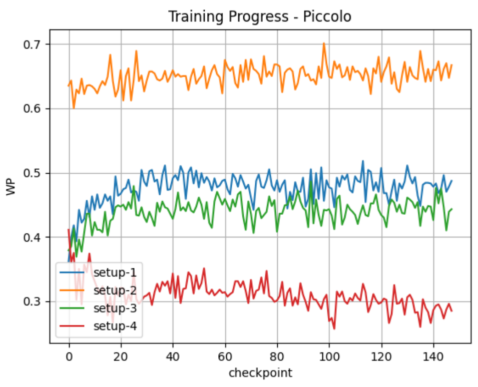
(2) Solo



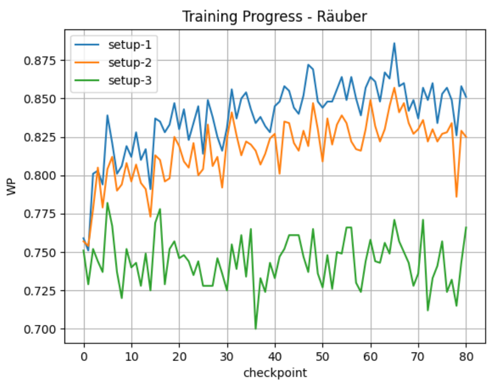
(3) Ultimo



(4) Bettel



(5) Piccolo



(6) Räuber

Figure E1: Training Progress of Sub-Games

Appendix F Analyzing the Behavior of the Ultimo AI

In the results, it could be observed that the Ultimo AI was able to achieve a high WP. In addition, this sub-game's winning conditions are well suited to analyze the AI behavior by conducting a simulation. For this, 100,000 games were simulated with a random seed of 15. It was monitored what cards are played the most in what rounds by the AI. The expectation is that the AI should have learned to play the card *1-trump* as late as possible.

The results of the card play frequencies in rounds are represented in a heat map (see Figure F1). As can be seen, the card with the index 32, which is the card *1-trump*, is indeed played in the last round as often as possible. Interestingly enough, the AI shows a strong tendency to play the card with index 53, the card *Gstieß-trump*, in the first round.

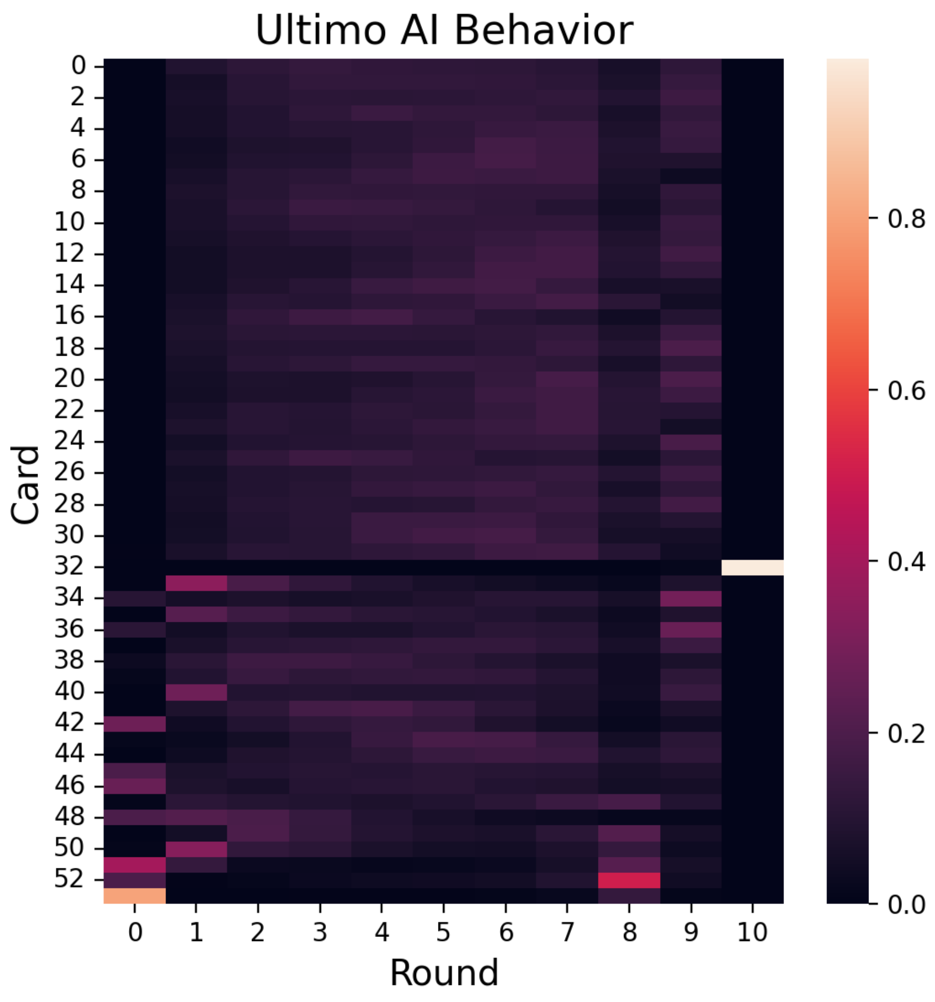


Figure F1: Heatmap – Ultimo AI Card Play Behaviour

Appendix G Considerations on the API Usage

This appendix section provides considerations and suggestions about using the API for implementing a Cego AI.

1 Adjusting the Strength of the AI

In the context of a player-friendly game, it might be useful to adjust the strength of the AI to certain difficulty levels. A simple measure of implementing a probability parameter is proposed to define the chance of picking a random action. By increasing this parameter, the level of play decreases.

It should also be noted that the AI strength varies between sub-games. In conclusion, it is advisable to define individual probability parameters for each sub-game that can be tuned separately.

2 Rule-based System

A challenge that hasn't been addressed in the main part of this thesis is how to combine these models for a rule-based system that can maximize the level of play. This section does not offer a full answer to that, but highlights certain findings. Additionally, this section offers suggestions for successfully implementing a rule base system that uses the trained models of this work.

2.1 Prioritization of Sub-Games

For this thesis, a prioritization of sub-games is proposed from the AI perspective. It should be noted that this should be considered in the context of the heuristic game selection in subsection 4.5.1. This game selection should always be the priority. Only when these game selection rules are fulfilled for multiple sub-games, the following prioritization should be considered.

1. Ultimo
2. Solo
3. Bettel
4. Standard-Cego

5. Räuber
6. Piccolo

There are several reasons to give Ultimo top priority. The first reason is that it is, out of the special sub-games, the one where the highest number of points can be achieved. Second, it can't be bid against. Third, based on the game selection rules, the rules for Ultimo might often be a sub-set of the Solo game selection rules, and therefore, it should take priority over Solo. And Lastly, the model was able to achieve a high WP, in fact, the highest out of all sub-games. Even when playing against other AIs, the single-player AI maintains a high win percentage. Therefore, Ultimo should be prioritized over all other sub-games.

Solo is a game mode that can be considered low risk and high reward (see Subsection 2.5.10). In addition, the AI was able to receive a high APPG in this game mode. It is therefore prioritized second.

Bettel is prioritized over Standard-Cego and all its sub-forms. This is mainly due to the WP averages the model is able to achieve. Furthermore, the number of possible games is lower than the number of possible Standard-Cego games, as well as the chance to be able to play this mode.

Standard-Cego takes priority over Räuber. This is fitting because a player that does not have enough potential points on his hand to play Standard-Cego probably has a high chance of not losing in Räuber. The player, in addition, already has a reasonable chance of winning Räuber because the game mode only has one of four players losing.

Because the AI's WP in Piccolo is less than 50% on the single-player side, it should only be considered as a last resort. When the player is unable to play Räuber, it might be considered. The WP might also be increased by adding more constraints to selecting Piccolo as a sub-game.

2.2 Bidding in the Sub-Game of Cego

For this problem, the lower bound constraints mentioned by the group Cegofreunde St. Georgen [65] can, for example, be used as a base:

- ***Cego***: 15 points

- ***Halbe***: 17 points
- ***Eine***: 18 points
- ***Eine Leere***: 22 points
- ***Zwei Leere***: 26 points
- ***Zwei Verschiedene***: 30 points
- ***Kleiner Mann***: 32 points

When *Gegensolo* is called, it is advisable to increase these constraints each by about 2 to 3 points. This is because of the increased risk of losing tricks against the player who called *Solo*.