Ruben Ortlam

Developing and Evaluating Smart Agents for a Foosball Table Game

OTTO VON GUERICKE UNIVERSITÄT MAGDEBURG

INF FAKULTÄT FÜR INFORMATIK

Intelligent Cooperative Systems
Computational Intelligence

Developing and Evaluating Smart Agents for a Foosball Table Game

Master Thesis

Ruben Ortlam

October 25, 2022

Supervisor: Prof. Dr.-Ing. habil. Sanaz Mostaghim

Advisor: Dr.-Ing. Christoph Steup

Ruben Ortlam: *Developing and Evaluating Smart Agents for a Foosball Table Game*
Otto-von-Guericke Universität
Intelligent Cooperative Systems
Computational Intelligence
Magdeburg, 2022.

Contents

List of Figures **III**

List of Tables **V**

1 Introduction **1**

1.1 Motivation 1

1.2 Research Questions 1

1.3 Structure 2

2 Fundamentals **3**

2.1 Related Work 3

2.2 The Foosball Table Game 7

2.2.1 Foosball table 7

2.2.2 Foosball Table Simulator 8

2.2.3 Simple Algorithms for Comparison 11

2.2.4 Evaluation 16

2.3 Basic Concepts 17

2.3.1 Game AI 17

2.3.2 Markov Decision Process 19

2.3.3 Evolutionary Algorithms 19

2.3.4 Q-Learning 21

3 Evolutionary Algorithms **23**

3.1 Method 23

3.1.1 Evolutionary Algorithm 23

3.1.2 Random Search 25

3.1.3 Rolling Horizon Evolutionary Algorithm 26

3.2 Hyperparameter Experiments . . . 27
3.2.1 EA Agent . . . 28
3.2.2 RHEA Agent . . . 34
3.3 Discussion . . . 37

4 Deep Q-Learning 39
4.1 Algorithm Description . . . 39
4.1.1 Basic Algorithm Structure . . . 39
4.1.2 Action Space . . . 40
4.1.3 State Space . . . 41
4.1.4 Neural Network Framework . . . 42
4.1.5 Deep Q-Learning . . . 42
4.1.6 Experience Replay . . . 44
4.1.7 Epsilon Decay . . . 45
4.1.8 Bootstrapping . . . 45
4.1.9 Solo . . . 45
4.1.10 Multi-Frame Input and Convolution . . . 46
4.1.11 Training Opponent . . . 47
4.1.12 Handicap . . . 47
4.1.13 Reward Function . . . 48
4.1.14 Training algorithm . . . 48
4.2 Experiments . . . 50
4.2.1 Hyperparameters . . . 50
4.3 Discussion . . . 62

5 Direct Comparison 63

6 Conclusion and Future Work 65

Bibliography 69

List of Figures

2.1 Automated Foosball Table 8
2.2 Top-down view onto the playing field of the simulation 9
2.3 Basic principle of Simple Block 13
2.4 Basic principle of Interpolated Block 13
2.5 Heatmap of ball positions between IB and SB 15
2.6 General structure of an Evolutionary Algorithm 20

3.1 Basic structure of the Evolutionary Algorithm agent 23
3.2 Individual of the Evolutionary Algorithm 24
3.3 Basic structure of the Random Search agent 26
3.4 Basic structure of the Rolling Horizon Evolutionary Algorithm . 26
3.5 Average Goal Speed results of EA against IB 29
3.6 Dominance results of EA against IB 29
3.7 Goals scored of EA against IB 30
3.8 Execution times of the EA agent 31
3.9 Heatmaps of ball positions in games between EA and IB 33
3.10 Videos of a sample game of the EA agent variants against IB . . 33
3.11 Average Goal Speed values of RHEA against IB 35
3.12 Dominance results of RHEA against IB 35
3.13 Execution times of the RHEA agent 36
3.14 Video of a sample game of the RHEA agent against IB 37

4.1 Basic structure of the DQN agent 39
4.2 Goalkeeper rod with discretized relative translation actions . . . 41
4.3 Training loss of the relative or absolute experiments 53

4.4 Training loss of the bootstrapping experiments 53
4.5 Training loss of the multi-frame/convolution experiments 54
4.6 Training loss of the opponent mode experiments 55
4.7 Training loss of the neuron layer experiments (Part 1) 56
4.8 Training loss of the neuron layer experiments (Part 2) 57
4.9 Training loss of the reward function experiments 58
4.10 Training loss of the Solo experiments 58
4.11 Average Goal Speed results of DQN against SB 60
4.12 Dominance results of DQN against SB 61

5.1 Direct Comparison Heatmaps 63

List of Tables

2.1 Related Work 4

3.1 EA Hyperparameters 28
3.2 Final EA Hyperparameters 32
3.3 RHEA Hyperparameters 34
3.4 Final RHEA Hyperparameters 37

4.1 Training Hyperparameters 50
4.2 DQN Experiment Hyperparameters 51
4.3 DQN neuron layer values for the hyperparameter experiments . 52

1 Introduction

1.1 Motivation

Foosball tables serve as simple competitive entertainment in many bars and homes around the world. The game is easy to learn, but hard to master, requiring a combination of fast reactions, speed, and a knack for precise manipulation of the ball within the limited control the game offers its players. It presents a unique opportunity for research into reinforcement learning on a hardware game, multi-agent cooperation or competition. Furthermore, the reality gap and differences in ability between humans and machines can be analyzed. Results of research in this area are relevant for highly dynamic (robotics) control problems.
An automated Foosball table was created at the Otto-von-Guericke Universität Magdeburg and further work is being done to enable fully automated games without human supervision.
This thesis aims to serve as a first step into the application of state-of-the-art game AI methods on the Foosball Table. Instead of the actual game, it focuses on a simplified simulation developed alongside the hardware. At a later point, the methods and findings can be transferred over to the automated Foosball Table.

1.2 Research Questions

This thesis aims to answer or contribute to the following research questions:

Question 1 How well can a Rolling Horizon Evolutionary Algorithm algorithm play the game?

Question 2 How well can a Deep Q-Learning algorithm play the game?

Question 3 Which game AI algorithms work on the Foosball Table game, and how do they compare?

1.3 Structure

This thesis is structured as follows: At first, several related papers are presented and compared to the Foosball Table problem in Section 2.1 It is shown how their results relate to this thesis and may be of use for it. Afterwards, the automated Foosball table and the simulation are presented in Section 2.2, alongside two simple algorithms that are already in use and can be used as a reference for evaluations. Furthermore, some methods to evaluate the performances of agents are also introduced here. Section 2.3 describes the fundamental principles that the following algorithms are based on, before the first of the two algorithms is introduced and evaluated in Chapter 3, the Rolling Horizon Evolutionary Algorithm agent. This chapter also describes a different Evolutionary Algorithm agent that was created as a stepping stone to the more complex algorithm. Then, the Deep Q-Network agent is introduced and evaluated in Chapter 4, before all five presented algorithms are compared using the metrics introduced earlier and their differences highlighted in Chapter 5. Finally, all findings are discussed in relation to the research questions in Chapter 6 and future research beyond the scope of this thesis is proposed.

2 Fundamentals

2.1 Related Work

Table 2.1 shows an overview of papers that worked on problems related to the Foosball Table game. The column **Method** shows the acronym of the way or ways that the paper approached its problem. **Environment** displays the problem that was solved in the paper, while **Classification** provides a broad category that it fits into. **Decision Time** and **Action Space** broadly classify the presented problem by the time the presented agents have to decide and the size of the action space they have to search. This is important when comparing them to the problem presented here because the real-time requirements of the Foosball Table problem need rapid decision-making that not every method might support.

Table 2.1: Related Work

Source	Method	Environment	Classification	Decision Time	Action Space
[6]	ACN	Foosball Table	Physics Simulation, Hardware	5-33ms	Continuous
[19]	DQN	Foosball Table	Physics Simulation, Hardware	133ms	3
[18]	RS/EA/RHEA/MCTS	PTSP	Video Game	40ms	6
[9]	RHEA	GVGAI	Video Game Set	480 simulations	Up to 5
[23]	DQN	Air Hockey	Robot Simulation	50ms	25
[15]	DQN	Atari	Video Game	Unlimited	4-18
[11]	DQN	Atari	Video Game	Unlimited	Various
[27]	DQN	Atari	Video Game	Unlimited	3-18
[12]	DQN	Atari	Video Game	Unlimited	Up to 18
[8]	MADRL	Doubles Pong	Video Game	Unlimited	3
This thesis	RS/EA/RHEA/DQN	Foosball Table	2D Physics Simulation	8.3ms	Various

Foosball Table Automation [6] used a mechanized foosball table to investigate the transferability of reinforcement learning on a subtask from a unity-based three-dimensional simulation onto the physical table. The subtask they chose was hitting the goal using only the attack rod, while the focus was on a simple transfer process from simulation to real for-industry use, without further manual steps. An actor-critic network (ACN) was used to learn this task, combined with policy gradients for a continuous action space.
[19] further built on this and implemented and tested a goalkeeper network to block balls going towards it. They used a Double Deep Q-Network (DQN) network and tested it on the same simulation and hardware. These papers focus on the transfer of a solution to a narrow problem from simulation to hardware, the reality gap, while this thesis focuses on solutions to the problem of controlling all four rods through an entire game, not just one situation. The simulation used here is also much simpler than the unity-based three-dimensional situation in the previous papers.
Another paper investigating a similar physics-based problem is [23]. They used a Double DQN to learn to strike a puck in air hockey into the opponent's goal, and proposed using a teacher policy for a part of the learning process to guide the network towards more feasible solutions.
These papers showed that reinforcement learning can be used to solve subtasks for a foosball table agent, in a simulation or on the hardware. Scaling this up to the whole game remains an interesting problem that has not been solved yet. Since the simulation used in this paper is simplified in a different domain, using a similar approach on a larger problem may be possible.

Deep Q-Learning A lot of research (for example [11], [27], [20], [12]) has been done to improve Deep Q-Learning over the years since [15] successfully used it to play Atari 2600 games of the Arcade Learning Environment (ALE) [3] at a high level. [10] proposed a solution to an overestimation problem in Q-Learning: The tendency to overestimate the Q-values of actions within certain games. This was later successfully applied to Deep Q-Learning [11] and showed promising improvements when used on the ALE game set. A further improvement to this architecture was proposed by [27]: Splitting the fully connected part of the DQN into a value and an advantage stream allows the network to learn action and state values more efficiently. [20] proposed an improvement to the replay buffer by [15], which prioritizes learning from game states where the predicted Q-value differs more from the actual ones. [12] provides an overview

over the previously mentioned extensions to DQN and compares their advantages over the baseline.
While the ALE game suite is very different from the Foosball Table simulation, the impressive results of Deep Q-Learning in these games and in other domains, as well as the successful application of DQN to a related sub-problem by [19] means training a DQN agent to play an entire game of Foosball is an interesting approach not yet attempted.
[8] trained multiple agents concurrently to play Doubles Pong cooperatively. Using Multi-Agent Deep Reinforcement Learning (MADRL) the agents received the same rewards and state values, but picked values separately for their part of the game. While the game itself was rather simple, the emergence of cooperative behavior in multi-agent environments with DQN is promising for the development of a DQN Foosball Table agent. The separation of the game into multiple rods that can be controlled independently (which happens regularly in the actual game, where up to four people control one side of the table) allow a similar separation into one DQN for each rod with a global reward structure.

The Physical Travelling Salesman Problem A game that is related to the Foosball Table in terms of physics interaction and time limits is the Physical Travelling Salesman Problem (PTSP) [17]. A ship has to be directed through a two-dimensional maze to pass through several waypoints. It combines the regular Travelling Salesman Problem (TSP) planning problem, finding the shortest path for a salesman between a number of cities, with a short-term physics-based control problem involving obstacle avoidance and path planning. In each time step the agent, referred to as ship, has to pick one of six actions. These may be a combination of accelerating forwards or not and turning left, right or not turning. [18] presents and compares four methods to solve the short-term part of this problem, the local planning. They introduce macro-actions, which means repeating the same action a predetermined number of times before the agent picks a new one. This has two advantages: It reduces the search space considerably, in the PTSP example from between 6^{200} and 6^{400} to 6^{20} and 6^{40} with macro-actions of size 10 to reach a single waypoint. It also provides the agent with more time to find the next action.
[18] presents two Rolling Horizon Evolutionary Algorithm (RHEA) agents and compares them to one Monte-Carlo Tree Search (MCTS) and one Random Search (RS) agent. Both RHEA agents compare favorably to MCTS here,

performing similarly to MCTS while running fewer evaluations per step. The PTSP game is simpler than the Foosball table simulation game presented in this thesis, featuring both a smaller action space and a longer decision time. RHEA algorithms have also been applied successfully to general video game playing problems, where they have to perform in a number of different games. [9] found that within a time limit, the algorithm did not perform better than a random search in the games they picked from the General Video Game AI (GVGAI) set, but is capable of outperforming MCTS agents using the same forward simulation count limit of 480, when configured with a high population size. The tasks in this paper are different from the Foosball Table game, but the methods used and the time limit in the comparison make it relevant for applying these methods to the game in this thesis.

To be able to provide a partial answer to the third research question, which algorithms work on the Foosball Table game and how they compare, two methods need to be chosen, implemented and compared. DQN was chosen for a number of reasons: It was already used successfully to solve subtasks of the Foosball Table game by [6] and [19]. Additionally, this thesis focuses only on a simplified two-dimensional version of the game. The other method will be RHEA, because of the work done on the PTSP, which can be compared to the Foosball Table game. A similar approach to the PTSP RHEA agent by [18] may yield good results that can then be compared to the very different DQN agent.

2.2 The Foosball Table Game

This section introduces the task this thesis attempts to solve, details about the simulation, reference algorithms and evaluation methods.

2.2.1 Foosball table

The automated Foosball table was created at the Otto-von-Guericke Universität as part of a team project (shown in Figure 2.1). It consists of a retail Foosball table with a motor assembly on one side capable of controlling all eight rods. A regular computer sends commands to a microcontroller, which forwards them to the motor controllers. The ball position is captured by a

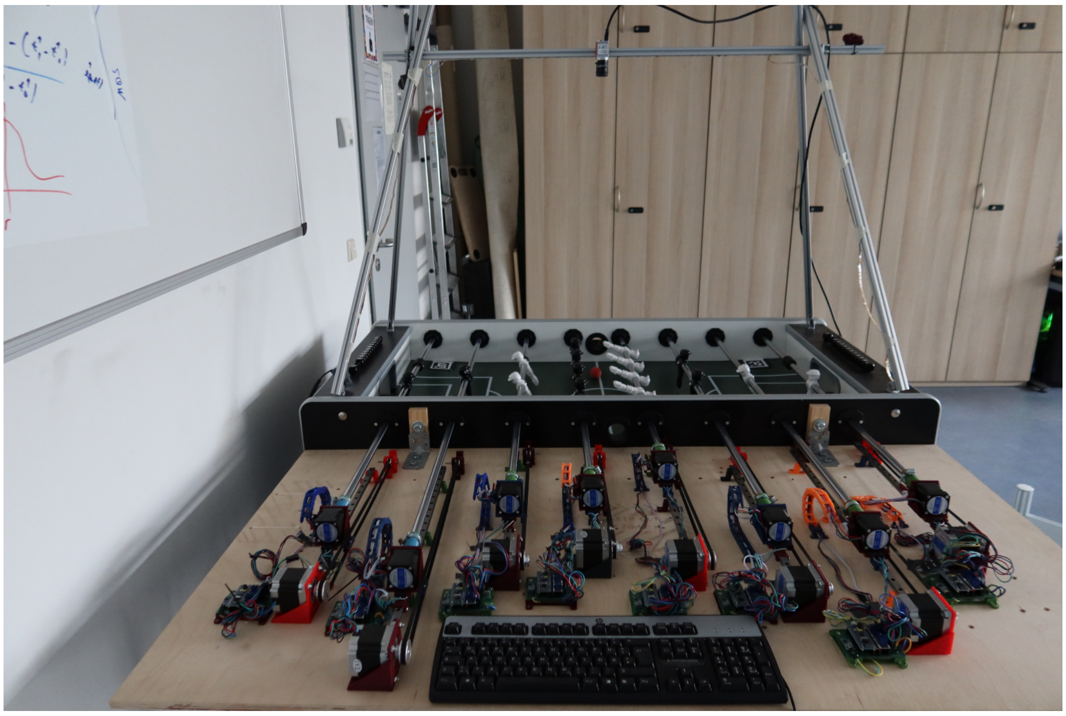

Figure 2.1: Automated Foosball Table

camera fixed above the table, looking down. It transmits 120 frames per second to the controlling computer, allowing reactive play. The whole project is capable of playing against itself, although at this point the ball has to be inserted manually. A further point that is missing so far is feedback for rod position and rotation, at the time of writing the computer has no knowledge about this beyond the commands that it sent. Both of these issues will be solved as part of further projects.
This thesis intends to create a foundation for research into Computational Intelligence methods of controlling this table. As the hardware is not yet ready for development, these first steps are done solely within a simulation, with the goal of transferring gathered knowledge to the hardware in the future.

2.2.2 Foosball Table Simulator

To be able to design algorithms to control the Foosball Table, a simulation was developed alongside the hardware to approximate the game while using the same player interface. It was decided to simplify the simulation by dropping the third dimension and simulating the few non-flat areas on the table using static forces, a very different approach to what [6] and [19] used. This simpler approach allows focusing on whether or how well an algorithm works on the general problem, while simulating many games for comparison with comparatively little compute time. [6] and [19] focused on transferability between simulation and hardware, while this thesis focuses on finding algorithms that work on the broad problem of playing an entire game of Foosball. The

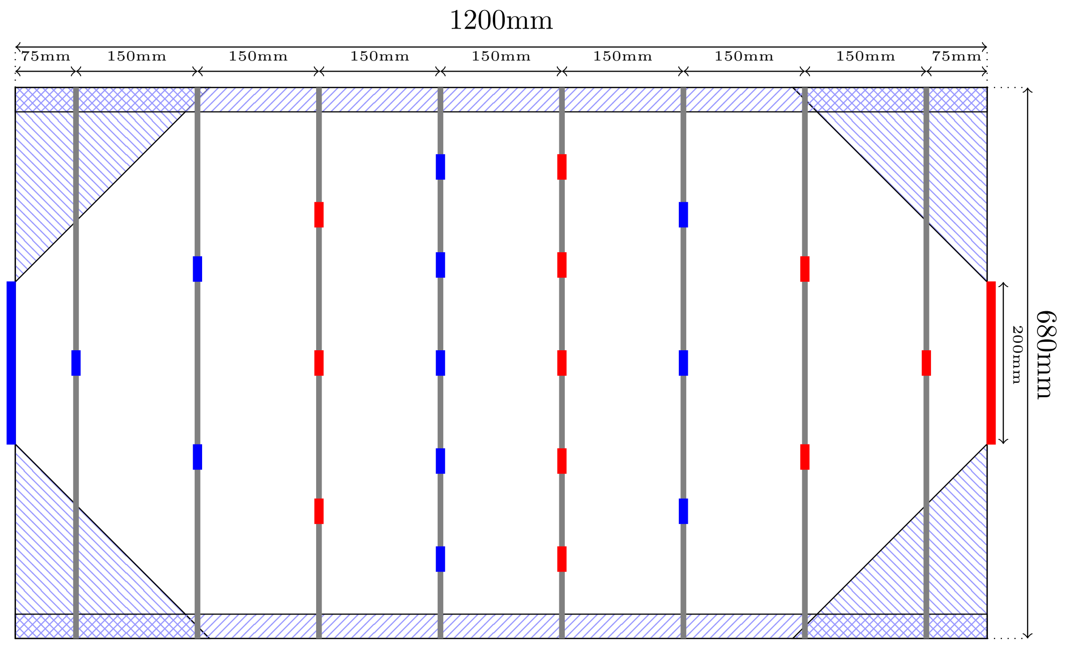

Figure 2.2: Top-down view onto the playing field of the simulation

Foosball Table is approximated as a two-dimensional rectangle, surrounded by solid border walls. Their only openings are the goals on each side. The small slant in the corners and the long edges of the foosball table are recreated using a static force applied to the ball in these areas. The basic structure and measurements are displayed in Figure 2.2. The four rods that make up one player are named Goalkeeper, Defense, Midfield and Attack. The hatched areas on the figure show the slanted areas of the table, which apply a small force to the ball to keep it away from the walls.
Representing the ball in the simulation is a circle of the same diameter. All game interactions happen around its position and rigid-body collisions with the surrounding walls or the players.
Players are simulated as boxes by calculating their intersection area at the

height of the ball. Collisions are handled by Box2d [5]. Vertical force, for example by a player down onto the ball, is not considered within the simulation. This prevents some special game tactics that are used in the actual game, but speeds up the simulation significantly.
The simulation runs at 120 updates per second and adds nearly no overhead to the player agent calculations. A game starts with the ball getting placed in the middle of the field and receiving a push in a random direction. This is a deviation from the actual Foosball game, which drops the ball in from one side, rolling between the midfield players of each side until it reaches one of them. This is because the simulation does not contain the slight unevenness of the table which leads to a random starting position, so a simple replacement was picked to provide a simulacrum of this randomness. A further difference from the actual game is that collisions between the rectangles representing the players with the ball are one-way only. In the physical game, the ball is often dampened by hitting a player and rotating the rod away from itself. This is not represented in the simulation.

The Foosball Table game has no clearly defined set of actions or states that would make tree searches or similar algorithms feasible. The chaotic and fast nature of the game means the same action sequence done just with a slight delay may lead to a vastly different outcome.
Another notable quirk of the game is that inaction of one player stalls the performance of the other player. A static or non-reactive agent leaves over half of the field where the ball would get stuck if it were to stop there. That means that in a way, both players have to cooperate to be able to play against each other. Although it is impossible to win without involving the opponent, as you have to get past their players, it is very possible to keep the ball indefinitely without any chance at counter-play. For this reason, the simulation includes a reset function which resets the ball to the starting position in the middle of the field if the ball has stopped moving dynamically for a predefined period of time.

The basic structure of an AI for the foosball table simulation can contain complex method structures and store variables as needed. The class contains the requested rod configurations. The simulation framework will attempt to move the rods into the requested positions, but most configurations are not reachable within a single time step. The simulation will move the rods towards the configuration. This can happen synchronously within the simulation or asynchronously as it would happen on the actual hardware. To reduce the influence

of the hardware used and load factors for experiments, only the synchronized mode is used. That means the AI calculations happen each simulation step right before the requested rod configuration is needed for the controller.
Each agent can change its target action each step. The ball can change direction and speed quickly, so this allows the agent to react in time to block a ball from reaching its goal. Of course, in hardware the decision time of an agent is constrained by computer speed instead of locked to any actual update rate, but within the simulation the agent can take as much time as it needs. To ensure transferability of the methods used in this thesis, the execution time is measured and compared to the 8.3 milliseconds limit for decisions on the hardware. The noise and delay of the hardware is also not considered in the simulation. To develop and evaluate agents without introducing further noise and difficulty, any data that the agent receives is correct.
Tuning the simulation to correspond as much as possible to the values of the actual Foosball table should be done before any attempt to transfer complex agents between them is done.

2.2.3 Simple Algorithms for Comparison

This section introduces two basic algorithms that were implemented alongside the Foosball Table and its simulation, to test its functionality and provide a baseline for future agents.

Simple Block

The simplest way of providing acceptable levels of play is just for each rod to find the player closest to the vertical position of the ball and moving it to that position (lines 2-7). This agent is called Simple Block (SB) and shown in Algorithm 1 and Figure 2.3. According to the horizontal position, the rod gets rotated so that players are either in the way to block it, moving forward to shoot or raised out of the way of the ball (lines 8-14) according to two thresholds *rottmin* and *rottmax*. This behavior has a few flaws: It does not take into account when a foosball rod cannot move further up or down, and still attempts to do so if the closest player to the ball would have to move in that direction to block it. This leaves openings in its defense. The simplistic rotation behavior also leads to situations where it shoots backwards when it has already reached shooting position, so the player is moved forward, but the ball

```
Input: $\overrightarrow{ball}$
Data: $rods$, $rottmin$, $rottmax$
Result: $translations$, $rotations$
for $r \in (0, rods.size)$ do
    $distances \leftarrow \perp$;
    for $p \in (0, size(rod.players))$ do
        $distances[p] \leftarrow \text{abs}(\overrightarrow{ball}.y - rods[r].players[p])$;
    end
    $min\_index \leftarrow \text{argmin}(distances)$;
    $translations[r] \leftarrow rods[r].players[min\_index] - \overrightarrow{ball}.y$;
    if $ball.x < rods[r].x - rottmin$ then
        $rotations[r] \leftarrow AVOID$;
    else if $ball.x < rods[r].x + rottmax$ then
        $rotations[r] \leftarrow SHOOT$;
    else
        $rotations[r] \leftarrow BLOCK$;
    end
end
```

Algorithm 1: Simple Block

is behind it. As soon as the ball moves far enough behind it, it moves back to rotate out of the way and shoots the ball in the wrong direction. These flaws would be easy to fix, but were left in to keep to the principle of simplicity for these algorithms. The focus here is not creating the optimal algorithm for foosball table play, but to provide a reference point for more intelligent approaches to the problem and an enemy to learn from. Moreover, despite these flaws, the performance of this algorithm is good, which demonstrates the importance of blocking within the Foosball Table game.

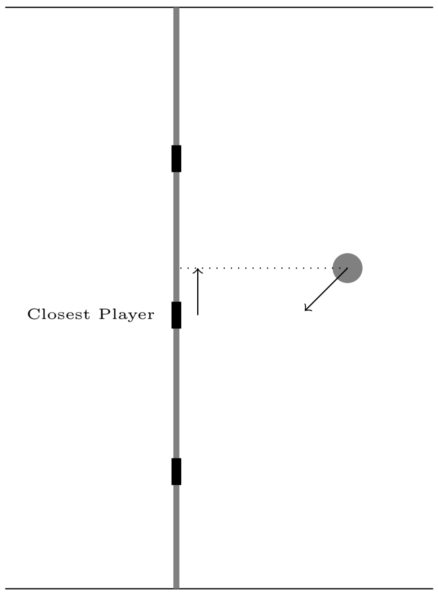

Figure 2.3: Basic principle of Simple Block

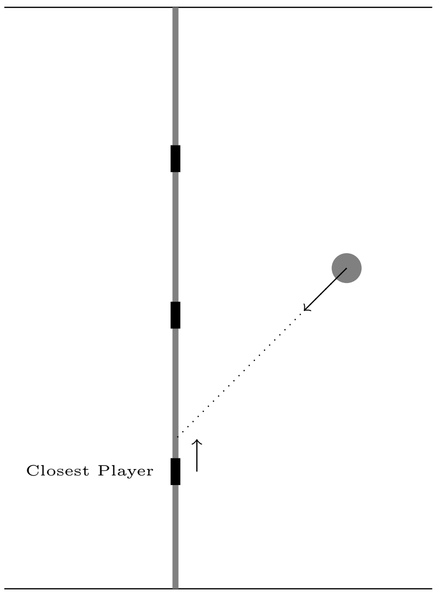

Figure 2.4: Basic principle of Interpolated Block

Interpolated Block

An improvement to SB is to add linear interpolation of the ball's path to it. This algorithm is named Interpolated Block (IB) and shown in **??** Algorithm 2. Excluding the corners and borders of the Foosball Table with their slight slant, the ball moves in a straight line until it hits something. This means interpolating the positions where it will cross the rod's positions (line 3) improves the reaction time of the agent. SB has to follow the current position of a diagonal shot, so it is limited by the speed the rod moves towards the ball's current height. Instead, IB predicts where the ball will cross the rod's path and moves a player into the path of the ball in anticipation. The rotation behavior (lines

```
Input: $\overrightarrow{ball}$, $\overrightarrow{ballvel}$
Data: $rods$, $rottmin$, $rottmax$
Result: $translations$, $rotations$
for $r \in (0, 3)$ do
    $distances \leftarrow \perp$;
    $intersect\_y \leftarrow \text{predict\_path}(\overrightarrow{ball}, \overrightarrow{ballvel}, rods[r].x)$;
    for $i \in (0, size(rods[r].players))$ do
        $distances[i] \leftarrow intersect\_y - rods[r].players[i]$;
    end
    $min\_index \leftarrow \text{argmin}(distances)$;
    $translations[r] \leftarrow rods[r].players[min\_index] - intersect\_y$;
    if $ballx < rods[r].x - rottmin$ then
        $rotations[r] \leftarrow AVOID$;
    else if $ballx < rods[r].x + rottmax$ then
        $rotations[r] \leftarrow SHOOT$;
    else
        $rotations[r] \leftarrow BLOCK$;
    end
end
```

Algorithm 2: Interpolated Block

8-14) is the same as in SB.
This change causes a surprisingly large improvement in performance for the algorithm that otherwise differs very little from the previous one. A direct comparison between SB and IB shows that IB wins very nearly every single game, with an average score of eight to zero within 120 seconds. It manages to keep the ball on the side of SB on the table more than on its own and thus keeps it on the defensive. But Figure 2.5 shows that it does not dominate it. The ball is still regularly within the middle or on the side of IB. This is because it is only better in defense, not offense. It just shoots the ball forwards when it can, nothing else. This simple behavior also leads to the horizontal lines visible in the heatmap. Both algorithms tend to get stuck shooting the ball forward into the other player, who shoots it back. This keeps on repeating until the game resets the ball position.
Figure 2.5 also shows that the ball stays as the bottom of the field more than at the top, which is surprising. Switching the players from left to right and from right to left mirrors the heatmap horizontally, but not vertically. That means this bias for the lower side of the field is inherent in the game, and the reason for it is unclear for now.

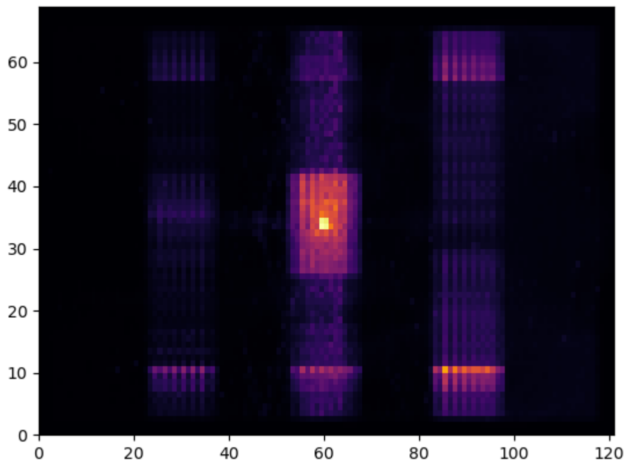

Figure 2.5: Heatmap showing ball positions within 1000 games between IB (left) and SB (right)

2.2.4 Evaluation

Evaluating the performance of the agents is another problem. Since the Foosball Table simulation is custom-made, there is no human-level performance threshold that could be used for comparisons, like [15] used for Atari games. As a two-player game where one player's performance relies on actions of the other player, there is no simple way of grading performance. Furthermore, it is unlikely that early versions of agents will reach this performance level. Instead, performance is compared to the algorithms introduced in Section 2.2.3, which serve as a baseline for comparison. Even beyond merely winning, there are indicators within the game that can be evaluated to find out how much better or worse an algorithm performs. This section introduces a selection of these indicators.

Win Rate

The most straightforward way of evaluating an agent's performance is putting it against a static enemy, for example one of the simple algorithms. After running n games, we can approximate the chance for the agent to win P_{win} using the number of wins w

$$P_{win} \approx \frac{w}{n} \tag{2.1}$$

and use it to compare its performance with its peers.
While this is simple, it leaves out many aspects of the game that might also be relevant. It is in theory possible for an agent to be excellent at defending itself, but equally bad at attacking. A simple example for this would be one of the reference algorithms SB and IB described previously without any rotation input. If they always blocked, they would still be good at defending, but terrible at attacking. They might still lose or at least never win, which would lead to a low score in this measurement and thus leave out its strength. Other metrics are required to show more details about strengths or weaknesses.

Manual Evaluation

It is possible to evaluate an agent's play style subjectively by watching it play. Obviously, this is not feasible for large-scale evaluation, but it is very helpful to find details about choices and tradeoffs that lead to the overall performance.

Average Goal Speed

Average Goal Speed (AGS) is a metric aimed at combining how long it takes for an agent to score a goal with how long it takes to receive one. To reward scoring goals in quick succession, but also receiving goals only slowly, the time values of goals scored $t \in T_+$ and goals taken $t \in T_-$ are subtracted from the maximum duration of a game t_{max}:

$$\mathrm{ags}(g) = \sum_{t \in T_+} (t_{max} - t) - \sum_{t \in T_-} (t_{max} - t) \tag{2.2}$$

Positive values mean the agent won on average, while negative values mean it lost on average.

Dominance

A simple way to ascertain which player has the upper hand in a Foosball Table game is evaluating which side of the table the ball is staying at most of the time. An average of the horizontal component of the ball position can show even slight advantages. This is very relevant for the overall result because it improves the chances for the player to score. Shots that go across the entire field are possible and do happen, but most of the time players score in the vicinity of the goal. Fewer obstacles in the way and fewer ways for the opponent to defend itself improve the chances considerably, and thus a player with a higher dominance is more likely to score.

2.3 Basic Concepts

This chapter describes the setup and the basic principles that apply to both approaches presented in this thesis.

2.3.1 Game AI

Computer games represent a very broad field of programs that is perfectly suited to develop the technologies required to emulate intelligence. There is a large incentive to develop ever-improving computer enemies for players to

face, but with a requirement to keep the calculations within the capabilities of a modern computer. At the same time, beyond attracting the ire of the players of these games, there is no big risk involved in deploying experimental techniques as there might be in other fields, for example in self-driving cars. Route-planning and decision-making processes that are developed for and tested in racing games or simulations of traffic systems may provide useful data that can later be used to develop systems for actual cars.
Reaching human-level performance in popular games has attracted a lot of attention. The field of Artificial Intelligence has come a long way since Deep Blue[4] first beat the reigning chess world champion in 1997. This was done using a parallel approach to searching the hundreds of millions of configurations of the chess tree that could arise from the current state and picking the best one. The algorithm was assisted with a large database of professional chess games for the machine to use a part of the experience of professional chess players. A large amount of manual tuning and work went into it reaching this major goal, which shined a spotlight on the advances in the area.
Since then, approaches to play board games like chess have become simpler and more powerful. AlphaGo[21] has reached superhuman performance in the game of Go, which is regarded as a very difficult step to take due to Go's search tree complexity. They got around this using a combination of supervised Deep Learning and Monte-Carlo Tree Search. AlphaGo Zero[22] further improved upon this and moved to unsupervised learning, playing millions of games against itself to learn the game.
AlphaZero[29] has generalized this algorithm from Go to other games like Chess and Shogi. It uses the same combination of Monte-Carlo-Tree-Search and Deep Learning to achieve superhuman performance without supervision on combinatorial games. These are games with one or two players, no chance factors that influence the game, perfect information for each player, a turn-based execution with finite action and state spaces and a finite length with a win, loss or draw at the end[29, p. 392]. Applying this to games with infinite state or action spaces, where an explicit tree search is not even feasible anymore, comes with its own set of challenges. [15] proposed the Deep Q-Learning architecture, which exceeds human-level performance in many Atari games. They achieved this by approximating the Q-Function using Deep Learning.

2.3.2 Markov Decision Process

A mathematical framework for processes involving uncertainty is known as a Markov Decision Process (MDP). [25, p. 443-444] differentiates between two types of uncertainty: Uncertainty in the result of an action and uncertainty in perception. MDPs only deal with the former, they assume a perfect information state. Each agent knows the full state of the game. This also applies to the Foosball Table game, where both agents know the state of the game fully and the only uncertainty is in the results of actions.
A MDP is a 4-tuple $S, A, p_a(s', s), r_a(s', s)$ where S is the set of possible states, A is the set of actions that can be chosen, $P_a(s', s)$ is the probability of reaching state $s' \in S$ after choosing action $a \in A$ in state $s \in S$ and $r_a(s', s)$ is the reward for reaching state s' from state s with the action a. The state and action sets can be infinite. MDPs form the basis for mathematical description of Reinforcement Learning problems. Each agent in an MDP has full knowledge of the game state.
In the Foosball Table case, the state space is made up by the possible configurations of ball and rod positions and velocities. The action space consists of rotation and translation actions for each rod. Both spaces are infinite, but can be reduced significantly using discretization.

2.3.3 Evolutionary Algorithms

> **Definition 1** (Optimization problem). An optimization problem is a pair (Ω, f) consisting of a (search) space Ω of potential solutions and an evaluation function $f : \Omega \to \mathbb{R}$ that assigns a quality assessment $f(\omega)$ to each candidate solution $\omega \in \Omega$. An element $\omega \in \Omega$ is called an (exact) solution of the optimization problem (Ω, f) if and only if it is a global maximum of f, that is, if $\forall \omega' \in \Omega : f(\omega') \leq f(\omega)$. ([14, p. 231])

A variety of problems can be considered optimization problems and solved using approaches from four categories [14, p. 233-234]: Some can be solved analytically, some by completely exploring the search space, if it is small enough. Others can be solved using a blind random search, by picking random solutions and keeping the optimum. The last category is a guided random search, which is related to the random search, but uses further information about the search space and the evaluation function to “guide” the search towards an optimum.

Algorithms which iteratively work towards solving an optimization problem are called metaheuristics[14, p. 225]. They are typically used on problems which cannot be solved numerically in reasonable time. Approximate solutions are often “good enough” and a variety of methods exist to find them.

One such metaheuristic in the guided random search category is the Evolutionary Algorithm (EA). EAs have their basis in nature. A living organism is defined by its chromosomes, which consist of genes. Different configurations of genes cause changes that can improve or worsen its probabilities of survival in the surrounding environment, and thus its chances of procreating. Advantageous configurations have a higher chance of procreating and thus, over time, appear more and more in a population. Genes are inherited from parents or changed by a random process called mutation. Over long periods of time, complex organisms appear, perfectly adapted to their environment.

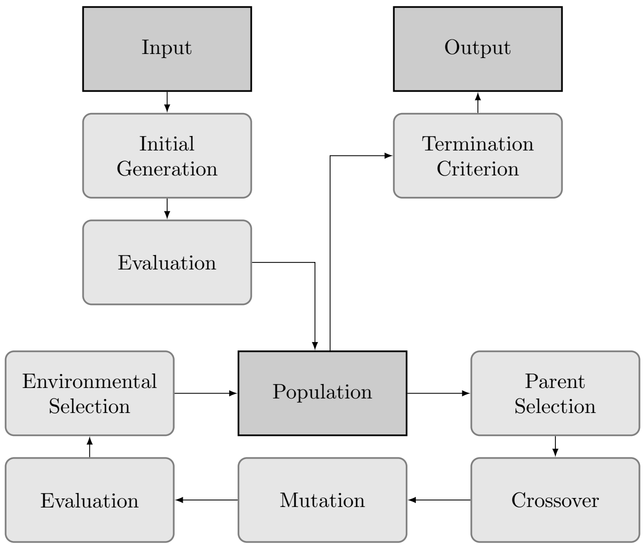

Figure 2.6: General structure of an Evolutionary Algorithm

This same principle is used to solve computational problems. The basic building blocks of an EA according to [14, p. 237] are a solution encoding, an initializer function for the population, a fitness function, a selection method, genetic operators, a termination criterion, and values for various hyperparameters. A solution has to be encoded in a way that allows genetic operators to work on it. Initially, the population is created by the initializer function and a fitness function is used to evaluate each individual solution. The genetic operators, usually crossover and mutation, are used to evolve the population over time. By selecting individuals with a higher fitness, a selection pressure is applied to the population which improves the individual solutions over time until a solution reaches a predefined threshold. Then the calculation ends and the problem is solved. One possible structure of such an algorithm is shown in Figure 2.6.

A Rolling Horizon Evolutionary Algorithm (RHEA) [18] uses this principle to play a game, using a simulation towards a limited time horizon of this game as fitness function. It generates and evolves chains of actions, before executing only the first action of the chain and starting the process again in the next time step.

2.3.4 Q-Learning

In Q-Learning [28], an agent moves through an MDP. For each step, it receives the state of the environment $s \in S$, picks an action $a \in A$ and moves into the follow-up state $s' \in S$. Accordingly, it receives the reward $r_a(s', s)$. The agent has to find the policy $\pi : S \to A$ that maximizes its discounted expected reward with discount factor γ

$$V^{\pi} \equiv r_a(s', s) + \gamma \sum_{s'} P_a(s', s) \cdot V^{\pi}(s') \tag{2.3}$$

To achieve this, it keeps track of Q-values that approximate the value of an action a from a state s and improve this approximation while playing.

$$Q^{\pi}(s, a) = r_s(a) + \gamma \sum_{s'} P_a(s', s) \cdot V^{\pi}(s') \tag{2.4}$$

The agent adjusts these Q-values in a look-up table during a sequence of episodes. In the nth episode, it observes the current state s_n, picks an action a_n, observes the new state s'_n, receives an immediate reward r_n and updates the Q-value of state s_n and action a_n with a learning factor α_n [28]

$$Q_n(s_n, a_n) = (1 - \alpha_n) Q_{n-1}(s_n, a_n) + \alpha_n \left(r_n + \gamma \max_{a' \in A} (Q_{n-1}(s_n, a') \right) \tag{2.5}$$

[28] showed that the Q-values reach optimal values for $n \to \infty$, which leads to one of the optimal policies when the agent always picks the action with the highest Q-value for a finite MDP.

Q-Learning serves as the basis for further algorithms that increase its utility by replacing the Q-function, which can quickly become infeasible with large state and action spaces, with approximations. One such example is Deep Q-Learning, which replaces the Q-function of regular Q-Learning with a Deep-Learning neural network. This network takes in the current state s and outputs a Q-value for each action $a_n \in A$.

3 Evolutionary Algorithms

This chapter introduces two EAs capable of playing the Foosball Table game, based on two different representations of an individual. A single-step EA directly on the simulation's input values and a RHEA based on custom action sequences.

3.1 Method

3.1.1 Evolutionary Algorithm

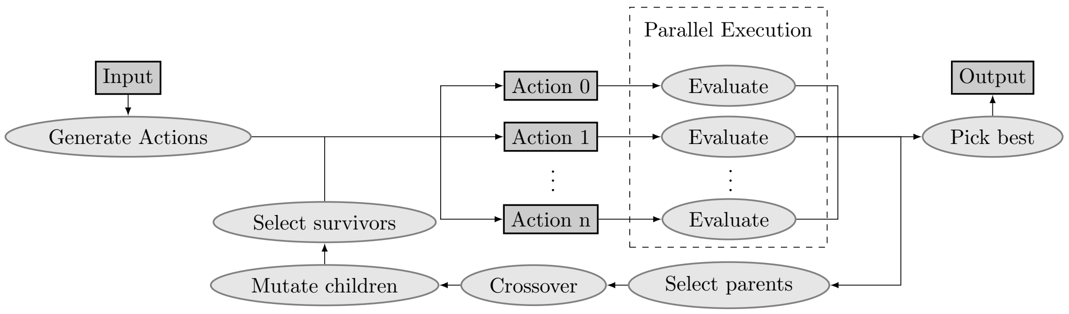

Figure 3.1: Basic structure of the Evolutionary Algorithm agent

The EA for the Foosball Table uses the simulation input value structure of a translation and a rotation value for each rod to encode an individual and uses genetic operators to find a local optimum of the value function within a tight time limit. It regenerates a list of possible solutions randomly each time step and evaluates them using the simulation. For this, the enemy is approximated using its current position within a special static agent that does nothing but keep this position. The evaluation simulation is run forward for a specific time, after which the ball position $\overrightarrow{b}$, the actual own and enemy scores lg_{game} and

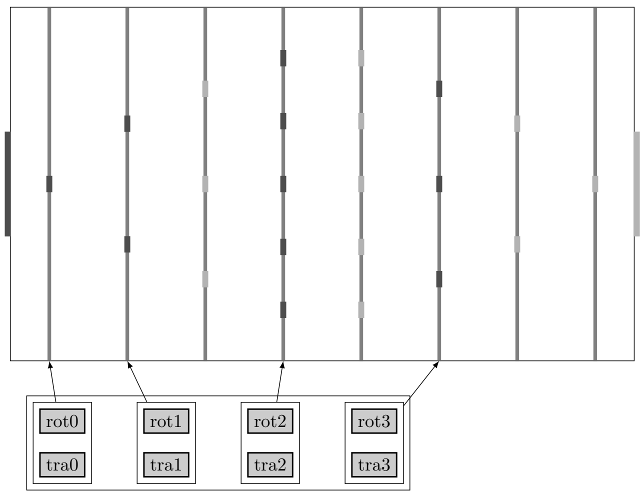

Figure 3.2: Individual of the Evolutionary Algorithm

rg_{game} and the simulated scores lgl_{sim} and rgl_{sim} are taken and evaluated using a simple fitness function:

$$v_i = \text{dist}(\overrightarrow{b}) - 5 \cdot (lg_{sim} - lg_{game}) + 5 \cdot (rg_{sim} - rg_{game}) \tag{3.1}$$

with

$$\text{dist}(\overrightarrow{v}) = (v.x - 1.2)^2 + (v.y - 0.34)^2 \tag{3.2}$$

where $(1.2, 0.34)$ is the position of the enemy goal. This is the squared distance of the ball to the enemy goal. The EAs will minimize this to attempt to get the ball into the enemy goal. Because the ball gets reset to the middle as soon as a goal is scored, which would lead to a bad fitness for solutions which scored, the difference in score from the actual game and the simulated result is also considered to counteract this effect.

This takes the long-term planning problem out of the hand of the agent, similar to the pre-planning step for the PTSP game used by [18], which pre-selects the order of waypoints, and leaves only the short-term route planning along these waypoints to the agents. The EA for the Foosball Table simulation works similarly afterwards. Instead of moving a ship through a path of waypoints, it attempts to move the ball towards a single destination, the enemy goal. No maze has to be navigated, the ball has to get past the moving obstacles that the enemy (and own) players provide. Instead of accelerating and rotating a ship, the ball has to be shot in a path that moves it past these obstacles to reach the enemy goal in a way that makes it impossible or unlikely to be intercepted.

Individuals are encoded using eight values as shown in Figure 3.2, two for each rod, representing the requested translation and rotation the agent wants the rods to move towards. They are generated uniformly along the valid translation value range for each rod, while the rotation value is chosen from the same three discrete positions used by SB and IB (Section 2.2.3). This was done because they provide good results with the two comparison algorithms, so this may improve the fitness of the starting population slightly. Parent selection is done using tournaments of size three. Elitism is used for environmental selection to ensure fast convergence, by cutting the worst individuals from the population until the target population size is reached. This value was picked because it was used successfully on the simpler, but comparable PTSP by [18]. Because the individual representation here consists of floating-point numbers with limited range, crossover is done using the bounded Simulated Binary Crossover (SBX) [7] operator and mutation is done with Polynomial Mutation. This allows the individuals to use rotation values beyond the initial discrete values. The operating structure is also shown in Figure 3.1.

Macro-actions are implemented similarly to [18]. Because actions of the Foosball Table EA agent are not cumulative, but absolute, repeating the same action over the duration of a macro-action is equivalent to setting the target rod configuration initially and doing nothing for the rest of the time.

It is possible to argue that this algorithm already fits the description of RHEA because it does look forward to a rolling horizon, to the time limit set by the simulation used as fitness function. It executes only a small step of its plan to move towards the target rod configuration within one time step before reconsidering. However, it cannot keep improving a plan over multiple time steps as it has to start from scratch each time again, without a sequence of actions that is usually used for RHEA. For this reason, a different approach that is closer to the concept of RHEA is implemented, and these two approaches will be compared in this chapter.

3.1.2 Random Search

“Devolving” the EA agent into a Random Search agent is as simple as setting the number of generations the algorithm will generate to zero. In that case,

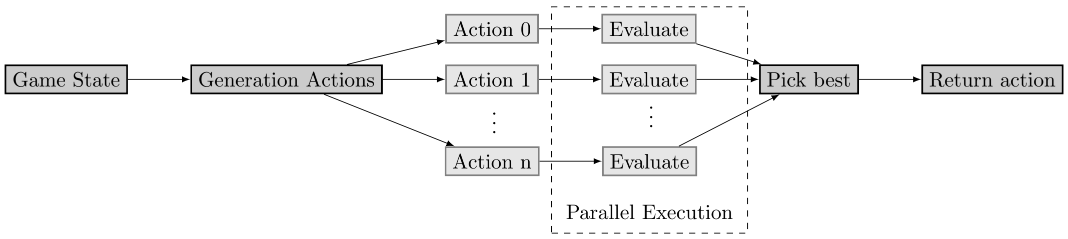

Figure 3.3: Basic structure of the Random Search agent

it does only the initial generation of individuals, then evaluates them in the same way and picks the best one it found. Then the EA agent operates as shown in Figure 3.3. This agent is included for comparison with the others, to evaluate how much of an advantage they bring over this basic approach. A larger population size of 40 was picked to provide a broader sample of the search space, and a look-ahead time of 0.25 seconds is used to evaluate the individuals.

3.1.3 Rolling Horizon Evolutionary Algorithm

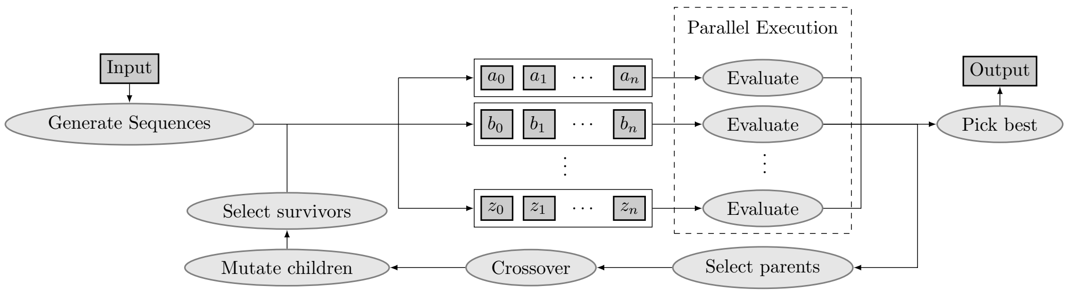

Figure 3.4: Basic structure of the Rolling Horizon Evolutionary Algorithm

The basic structure of the RHEA agent (shown in Figure 3.4) is similar to the EA agent. Only the input encoding was changed. Instead of the eight floats that represent a single target position for the rod to move towards, relative and discrete values are picked for each step. Rotation is done the same way as in the initialization step of the EA agent, with three discrete values: Avoid, Block and Shoot. Translation is done using 21 relative values in discrete steps. The reason for the high number of options here is that a high reaction speed is required for good performance in the Foosball Table game, so the RHEA agent needs the possibility to move quickly within a single time step.

This input parameter configuration reduces the search space and avoids a lot of the infeasible configurations of inputs which the rods cannot follow quickly enough. It also enables the creation of action sequences that are required for the implementation of a RHEA agent. Unlike the EA agent, the genetic operators act on the sequence of actions, not on a single target value. The crossover operator is a simple uniform crossover, same as [18] used for the PTSP. The mutation operator modifies only one part of an action: It picks a new rod the action influences, it changes the translation value by adding a random value picked from a normal distribution to it, or it picks a new rotation value. Elitism is used again for environmental selection in the same way as it was for the EA agent. Because low diversity in the results was found during development, the ability to add a number of randomly-generated individuals each generation was added, called the diversity rate.
The evaluation is done in the same way as for the EA agent and uses the same fitness function. The length of an action sequence corresponds with the simulation look-ahead time so that within one evaluation run the simulation executes the entire sequence of actions. When one action is executed, the first action in all individuals gets removed and a random new action gets added to the end, to keep the sequence at the same length.

3.2 Hyperparameter Experiments

To determine the optimal values for these hyperparameters, a number of experiments have to be run. For each parameter, a range was determined which can be evaluated within reasonable time and picked a preliminary default value. Then the performance of the agent is compared through 100 games, each with all hyperparameters on their preliminary default, except one, which gets a variety of values. The performance then gets evaluated in two directions: Which value leads to the highest-performing agent, and which value does so without requiring too much computation to fit within the macro-action size. This is important because an agent which performs very well in the simulation, but requires too much time, cannot be transferred to the physical hardware.
One problem with this time limit is that it depends on the hardware used. Each generation in either algorithm gets evaluated using multiple threads, which means that a CPU with a higher core count will have a higher headroom to use larger population sizes without requiring more time. To compare

the time required for each configuration, the simulation is run with the different parameters on a high-end desktop CPU (AMD Ryzen 5950X).

3.2.1 EA Agent

Table 3.1: EA Hyperparameters

Hyperparameter	Range	Experiment Default
Population Size	10-50	20
Number of Children	10-50	20
Number of Generations	1-4	2
Look-Ahead Time	0.05-0.5s	0.25s
Macro-Action Size	1-4	1
Mutation Rate	0-80%	50%

The range and initial values for the EA hyperparameters are shown in Table 3.1. The preliminary defaults were picked from early experiments while developing the method, after which they can be replaced with better values picked using the results from the experiments.

For the average goal speed shown in Figure 3.5 only the macro action size from 2 upwards and the simulation time show a significant difference. A higher macro action size has a negative effect on the average goal speed of the EA agent, while the other parameters showed no significant differences. Values of 3 and 4 perform significantly worse, with 4 showing a negative average goal speed, which means that it loses more games than it wins. A higher simulation time also has a slight negative effect on the average goal speed of the agent, meaning it takes longer to score goals with longer simulations. Because the execution time of the agent is not limited, this shows an advantage in shorter look-ahead times, possibly because it incentivizes more aggressive play to get results within the very short timeframe.

The average dominance of the experiments, shown in Figure 3.6, also does not show much difference for many of the separate experiments. This is because the furthest area that the player can influence is that of the attack rod, which is at 825 mm out of 1200 mm overall field length. This means that a dominance much further than $\frac{825\ mm}{1200\ mm} = 0.6875$ is not possible, as the ball has to move back to this position so that the agent can influence it. The high value of close

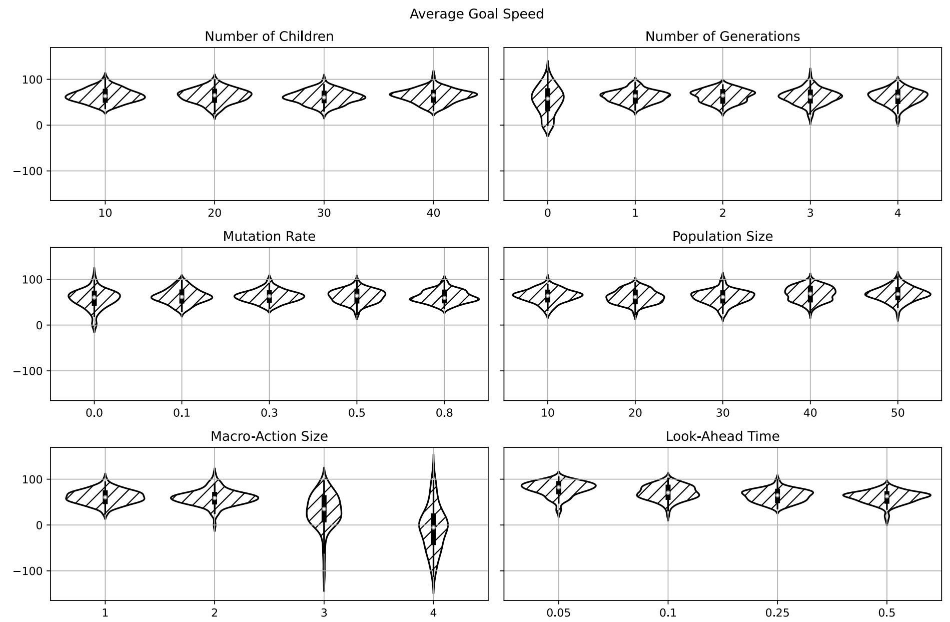

Figure 3.5: Average Goal Speed results of the EA hyperparameter experiments against IB

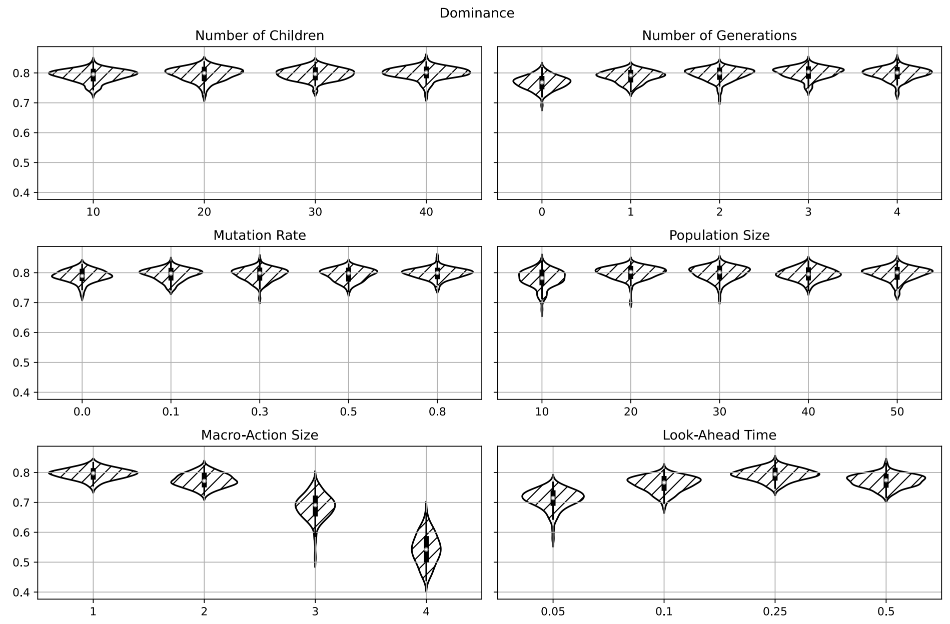

Figure 3.6: Dominance results of the EA hyperparameter experiments against IB

to 0.8 on average means that the enemy player is always on the defensive, unable to get the ball even beyond the attack row of the agent. Similar to the average goal speed, step size shows a negative influence on the dominance.
The change in the number of generations from 0 to 1, 1 to 2 and 2 to 3 show slight improvements in the dominance results, as does the change in population from 10 to 20. A very interesting effect is seen on the dominance results of the different simulation times. A lower simulation time shows an improvement in average goal speed, but a decrease in dominance. This suggests that these two metrics stand for different play styles. A higher dominance shows a good defensive, which keeps the ball furthest away from the own goal for as much time as possible. It does however not necessarily lead to scoring goals. A higher average goal speed does not contain many details about the defensive capabilities of an agent beyond not receiving many, but only about the ability to score goals. The very short simulation time of 0.05 seconds leads to a worse defensive, but a more decisive win.

Figure 3.7 shows the average goals the agent scored within 120 seconds.

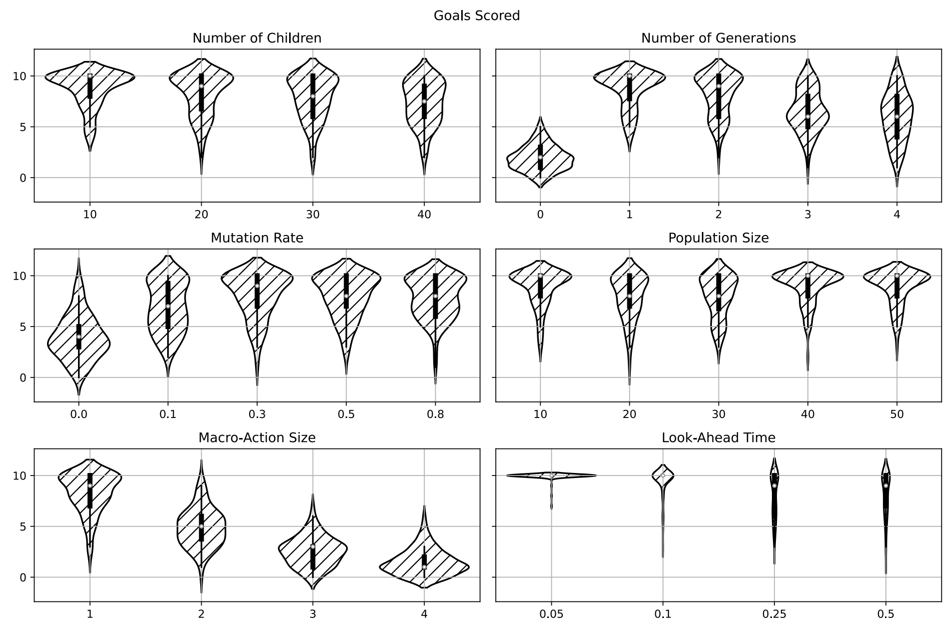

Figure 3.7: Goals scored of EA hyperparameter experiments against IB

Larger child counts seem to have a slightly negative effect on the number of goals scored. No generations beyond the initial random one (in essence random search) shows a low number of goals scored, but values higher than 1 seem to

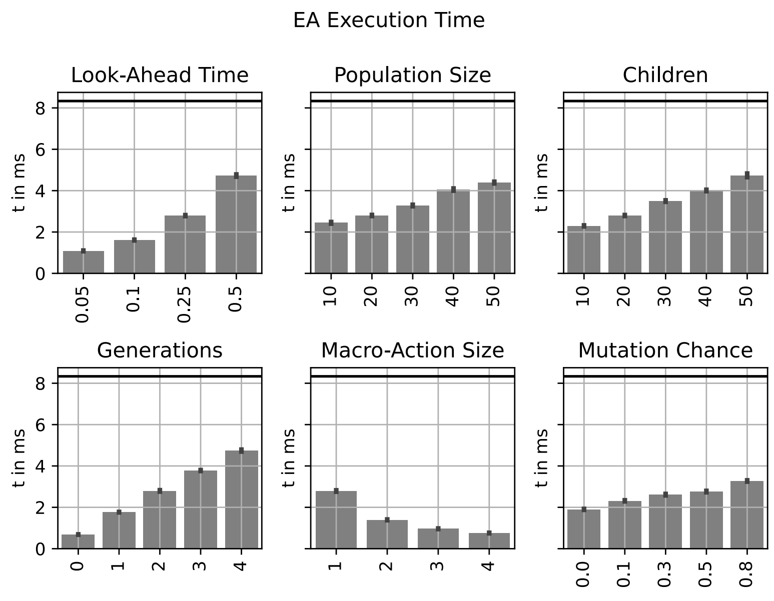

Figure 3.8: Execution times of the EA agent

degrade the number of goals scored slightly. This is confirmed with a Mann-Whitney U Test, showing p-values of less than 10^{-34} for the values of 0 and 1, 0.03 for 1 and 2 and $9.8 \cdot 10^{-7}$ for 2 and 3. Only 3 and 4 are not significantly different anymore, with a p-value of 0.09. High mutation rates of 0.3 upwards show the same results, but mutation rate below that reduce the goals scored. Only the change in population size from 10 to 20 shows a significant change, with a p-value of 0.0006. Unsurprisingly, increasing the macro-action size reduces the number of goals scored on average, similar to the other metrics.

Figure 3.8 shows that the execution time of the EA agent on the test system is well below the threshold of 8.3ms. No configuration exceeds even 5ms, indicating that the algorithm still has a lot of headroom for larger populations or more generations. But at least when comparing it to the IB, none of these parameters show an upwards trend in any metric. The algorithm seems to have reached a play style that is close to the optimum possible with its limited look-ahead capabilities. To improve this further, the method would have to be changed.

The combination of the best results of the hyperparameter experiments is shown in Table 3.2. Using these values leads to a very fast and aggressive strategy by the agent that sacrifices some defensive capability for speed. This

Table 3.2: Final EA Hyperparameters

Hyperparameter	Value
Population Size	20
Number of Children	10
Number of Generations	1
Look-Ahead Time	0.05s
Macro-Action Size	1
Mutation Rate	30%

fast style of attacking can overcome the strong defense of the IB, while it can still defend well-enough that the ball does not get past all of its rods. However, this may only be possible because Algorithm 2 does not have a strong attack, focusing mainly on defending. Whether this holds against other, more capable agents remains to be seen.

This difference in play style can also be seen in the heatmaps shown in Figure 3.9, where in the more successful aggressive style on the left heatmap the ball spends less time close to the enemy goal. On the right, the more defensive style keeps the ball in front of its attack rod most of the time, but scores fewer goals on average. These differences can also be seen in the two videos in Figure 3.10. These run at half speed to make them easier to follow. Notable in particular is how the random target values the EA agent generates each time step lead to the rods staying mostly in the middle, which causes it to miss blocks that should have been easy because the ball went along the top or bottom of the table. The number of players, especially on the midfield rod, and the placement of the goals in the middle mean that its strategy still works out.

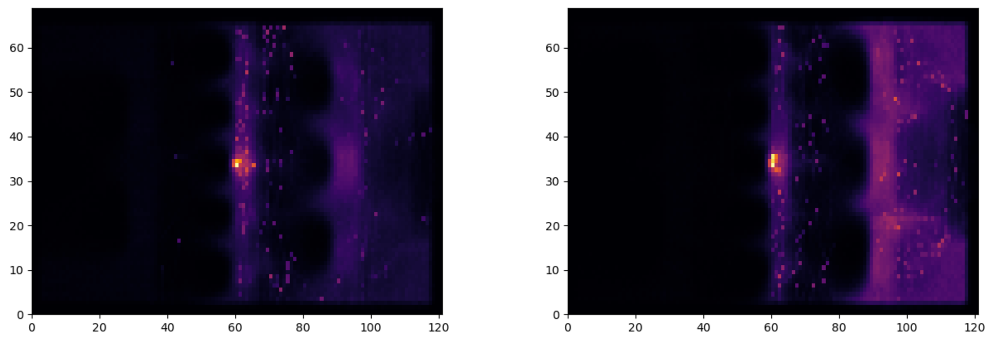

Figure 3.9: Heatmap showing ball positions over 200 games between the EA agent (left) and IB (right). The left heatmap is with the chosen parameters, the right one uses the same, except a simulation time of 0.25 seconds.

0.05s Look-Ahead Time 0.25s Look-Ahead Time

Figure 3.10: Videos of a sample game of the EA agent variants against IB at half speed

3.2.2 RHEA Agent

Table 3.3: RHEA Hyperparameters

Hyperparameter	Range	Experiment Default
Population Size	10-50	20
Number of Children	10-50	20
Number of Generations	1-4	4
Look-Ahead Time	0.05-0.5s	0.25s
Macro-Action Size	1-4	1
Mutation Rate	0-80%	50%
Diversity Rate	0-50%	10%

The hyperparameters and experiment ranges are shown in Table 3.3. The diversity rate is measured in percentage of the population size. Figure 3.11 shows little differences between the various experiment runs. The only differences that are significant enough to pass a Mann-Whitney U Test are the step from a diversity value of 0 to 0.1 with a p-value of 0.006 and the macro-action sizes with p-values of $2 \cdot 10^{-6}$, $9 \cdot 10^{-16}$ and $6 \cdot 10^{-7}$ for the steps 1 to 2, 2 to 3 and 3 to 4. No diversity per generation has a slight negative effect on the average goal speed, while increasing the macro-action size has a significant negative effect. To find further differences between the experimental runs, the dominance results are used.

Figure 3.12 shows the dominance results for the various values of hyperparameters. The most influential parameters are the macro-action size, look-ahead time, mutation rate and population size.
Increasing the macro-action size shows a significant reduction in performance, similar to the previous metric. Simulation look-ahead times show a significant improvement from 0.05 to 0.1 seconds and a slight improvement in dominance from 0.1 to 0.25 seconds (confirmed with Mann-Whitney U Test, p-value of 0.001). The dominance drops again when increasing it further to 0.5 seconds. A larger value means the static opponent within the evaluation simulations differs too much from the actual enemy, which degrades the value of the prediction.
The mutation rate shows a significant increase in dominance up to 0.5, which suggests that this is the optimal value. This is consistent with the results of the PTSP experiment by [18], although no drop in the value is observed for a mutation rate higher than 0.5 in this case.

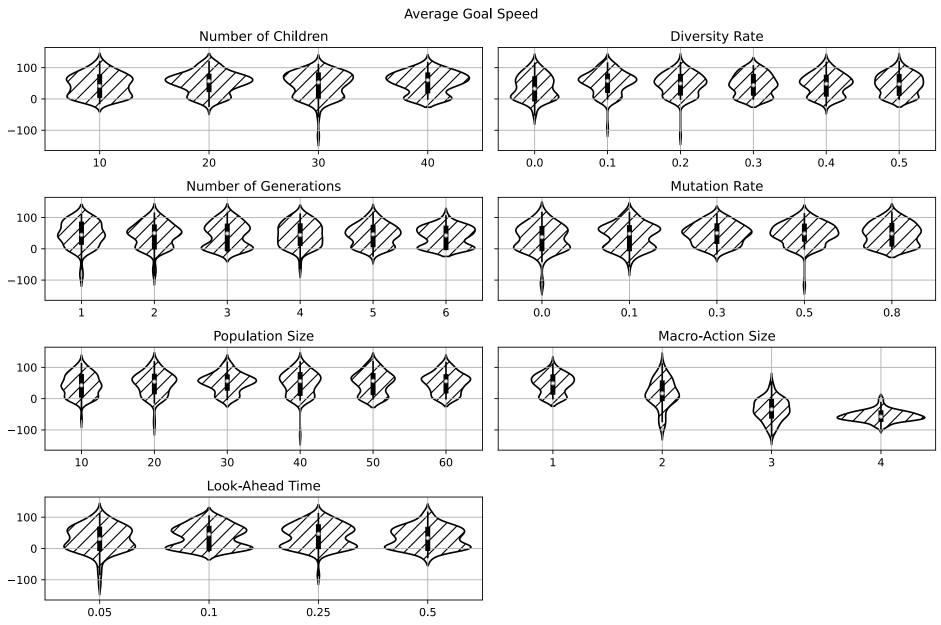

Figure 3.11: Average Goal Speed values of the RHEA hyperparameter experiments against IB

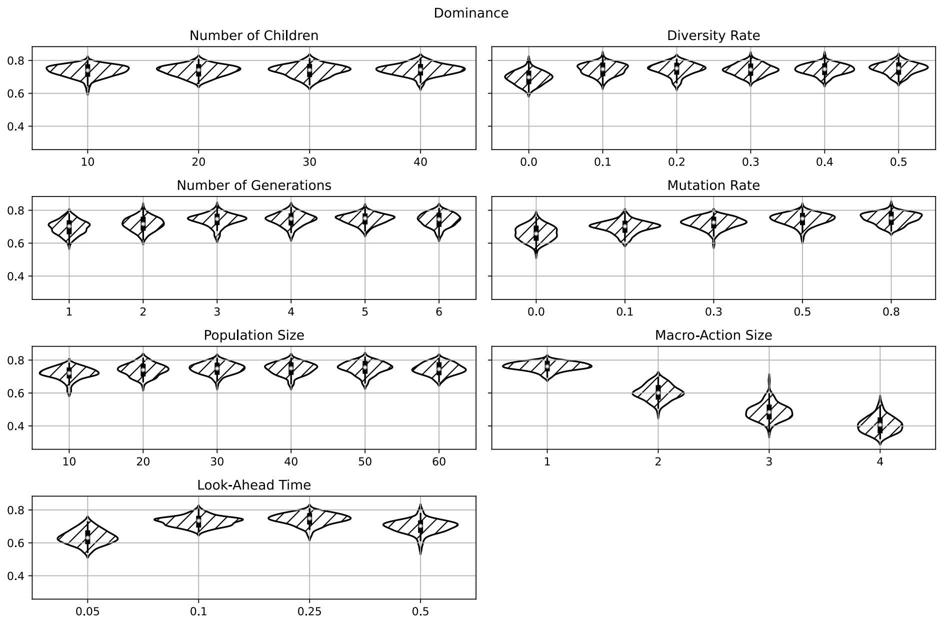

Figure 3.12: Dominance results of the RHEA hyperparameter experiments against IB

Increasing the number of generations shows an increase in dominance from 1 to 2 with p-value 0.0007 and from 2 to 3 with p-value $2 \cdot 10^{-6}$, after which it plateaus. The solutions found do not get better afterwards, likely limited by a different parameter or the algorithm design.
The diversity rate shows no difference between 0.1 and 0.5, but a drop in dominance for 0.0. The smallest of the values with good performance should be chosen because larger values cause a small increase of the execution time.
The number of children has no significant influence on the dominance result, so the minimal value will be used for the final agent. This suggests that good solutions are found through mutation and the initial random search alongside the diversity solutions for the final result, rather than through crossover.

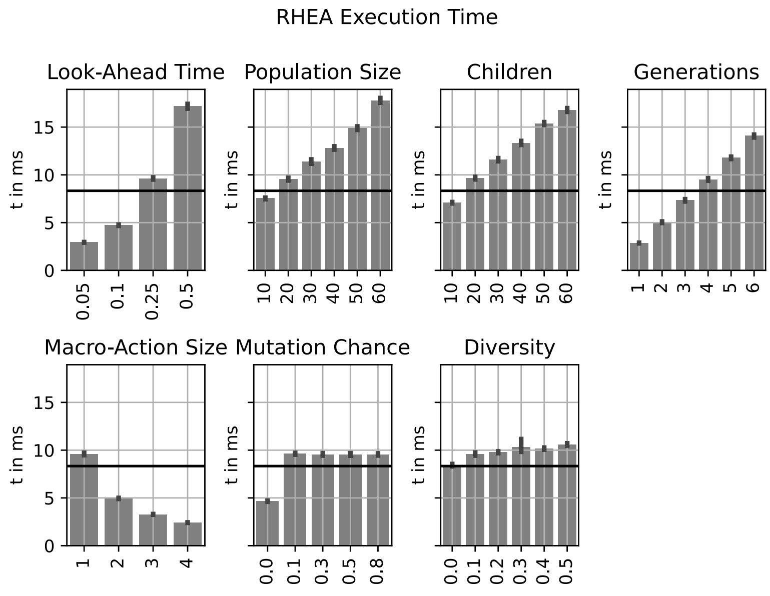

Figure 3.13: Execution times of the RHEA agent

Figure 3.13 shows that the RHEA agent is significantly more compute-intensive than the EA agent. The preliminary combination of hyperparameters picked, shown in Table 3.3, already takes longer to compute than the 8.3 millisecond limit, so this has to be considered when selecting parameter values for the final agent. The best value to get below the threshold of 8.3 milliseconds is the default value of 4 generations, because more than 3 showed no significant improvement anymore.

The parameters chosen for the final RHEA agent are shown in Table 3.4. This leads to good play, certainly capable of beating IB, but not as quickly as the

Table 3.4: Final RHEA Hyperparameters

Hyperparameter	Value
Population Size	20
Number of Children	10
Number of Generations	3
Look-Ahead Time	0.25s
Macro-Action Size	1
Mutation Rate	50%
Diversity Rate	10%

EA agent. Its play style is more comparable to the more defensive variant of EA with a look-ahead time of 0.25 seconds. Figure 3.14 shows a sample game of the agent at half speed. Notably, it does not have the same behavior as the EA agent, which keeps its rods mostly in the middle, because of its different action encoding.

Figure 3.14: Video of a sample game of the RHEA agent against IB at half speed

3.3 Discussion

The RHEA agent did not perform as well as expected. It does beat IB reliably and is able to play well, but it is both more expensive to run and performs worse than the EA agent. This shows that the difference in input configuration necessary to enable rolling-horizon planning either limits the possible movement range and reaction time of the agent too much or leads to a search space that is too large to reliably find good solutions within the time limit. The input encoding of the EA agent is more limited as it only allows a direct movement of the player rods towards a target configuration, while the RHEA agent allows more complex movements, for example back and forth. However, because the EA agent reconsiders its targeted rod configuration each step, it

can quickly reconsider and pick a more optimal position. This seems to perform better for the Foosball Table simulation.
The hyperparameter experiment results in Figure 3.12 show improvements when increasing population size or the number of generations, but also run into the maximum dominance possible when comparing with IB, so further improvements cannot be seen in this graph. The reduction in simulation look-ahead time that lead to a more aggressive play style for the EA agent did not have the same result for the RHEA agent. The various hyperparameters did not seem to have much of an effect on the ability to score goals, which means the offensive gameplay is limited by the action sequence abstraction of the RHEA agent.
The EA agent performed about as well as possible against the reference agent, beating it in every metric. This shows that for further development, a different reference agent is required. A notable, interesting result is that a shorter look-ahead time of the internal simulation lead to a better performing agent, as it forced the agent into a more aggressive play style. This difference in the fitness function shows that the implementation of a long-term planner that guides these short-term methods with an overall strategy would likely benefit them greatly.

4 Deep Q-Learning

This chapter contains a description and evaluation of a DQN agent for the Foosball Table simulation, of the problems that arose during development and the attempts to solve them.

4.1 Algorithm Description

4.1.1 Basic Algorithm Structure

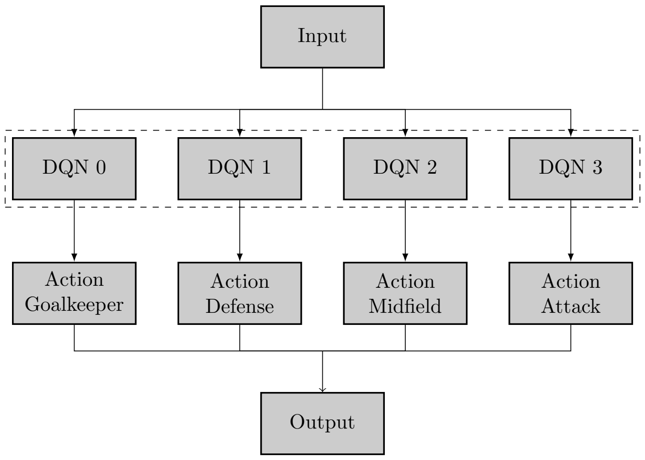

Figure 4.1: Basic structure of the DQN agent

Because the simulation allows direct integration of agents into the framework, with access to specific input data, there is no requirement to work on images similar to what [15] used for Atari games. The initial plan is to leave out the complex convolution layers and focus solely on small fully connected layers. This would simplify the agent and allow fast operation even on cheaper hardware.
Because controlling all rods from a central DQN would lead to too many actions, the decision was made to implement the Foosball Table DQN agent as a

Multi-Agent System (MAS). [8] has shown that that cooperative behavior can emerge from the combination of multiple DQN agents working separate parts of the same game.
Figure 4.1 shows the basic setup. The same set of input data is fed into four separate DQNs, each responsible for one rod. Each selects an action it considers optimal, and this action is then applied to the corresponding rod, influencing its translation and rotation.
The amount of input data, especially if it only consists of a single simulation time step, may not be sufficient to learn within the random environment of the simulation. [15] has used multiple input frames to allow the network to extract time-specific data, like movement and acceleration. This is also relevant for the Foosball Table agent. To examine this, the algorithm can run with one or multiple input frames and with 1-dimensional convolution layers on time data from a larger quantity of input frames.
One Q-Learning episode lasts from when the ball is placed in the middle until a goal is scored or the game ends due to the time limit.

The DQN agent implements Dueling Double Deep Q-Networks with or without convolution, Prioritized Experience Replay [20], an Epsilon Decay training algorithm, Bootstrapping, separate training of each rod (Solo) and a custom training algorithm. These are explained in further detail in the following sections.

4.1.2 Action Space

Deep Q-Learning requires a discrete number of actions, but the Foosball table simulation does not have a limited number of actions. Instead, the agent provides the translation and rotation values of that the player's rods should attempt to reach. This allows multiple ways of discretizing this action space, for example absolute (move to position x) or relative translation actions (move up). Absolute translation actions further require reducing the numerically infinite space between minimum and maximum rod position. A resolution of one centimeter was picked as a starting point, as this limits the size of the set of discrete positions and allows both absolute and relative actions. At a later point, lowering the resolution could be attempted, at least with relative translation actions.

Rotation can be discretized into very few static positions because the simulation is not complex enough to allow the three-dimensional fine-control a player might have with an actual Foosball table. One position to avoid the ball entirely, one to block it and one to move forwards to shoot should be enough to allow the agent to play while keeping the number of actions low-enough for Deep Q-Learning. This is the same configuration used for SB and IB (Section 2.2.3), with which it has shown to be good enough for an acceptable level of play.

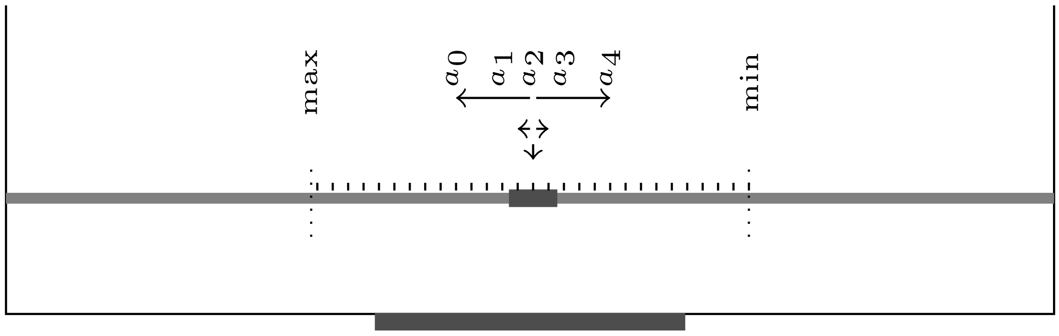

Figure 4.2: Goalkeeper rod with discretized relative translation actions (a_0 to a_4)

To allow both fast and precise translation, relative translation actions result in five different actions: Move up fast, move up slow, stay, move down slow and move down fast (shown in Figure 4.2). Combined with the three rotation actions, this results in 15 different actions.
Absolute translation actions result in a different number of actions for each rod, depending on the movement range of the rod.
Using a single DQN for all rods would increase the number of actions far beyond what is used by [15]. Especially in the Foosball Table environment, where in this case a vast majority of actions would have no effect on the ball, this would lead to a large redundancy in player actions. This would be confusing for the network and increase the training difficulty significantly. Thus, a player consists of four networks, each controlling a single rod. These are trained together to allow development of cooperative behavior, working together with the other rods.

4.1.3 State Space

The most relevant data for a player is the state of the ball, its position and velocity within the table's coordinate system. The velocity was added to allow

the possibility of using a single data point as input, similar to the EA and RHEA agents described previously. Data about the rods constitutes the rest of the values, where on the table it is located and what the current position and orientation of its players are.
During testing, it was also noticed that the neural network struggles connecting the effects of its actions with results because it takes many simulation steps before the requested rod state can be reached by the PID controller. To counteract this, the current requested position and rotation is also added to the input data to provide immediate feedback to the networks when an action is selected.
Put together, the state consists of two vectors defining ball position and velocity and five scalars for each of the rods, defining translation target, rotation target, angle, the static horizontal position of the rod and its vertical offset. This adds up to 44 values for the state vector.

4.1.4 Neural Network Framework

To train and run neural networks, PyTorch [16] was chosen as a framework because it can work directly with the simulation using its C++ API. Training directly in the simulation did not work due to instability within the C++ framework, so instead training data is extracted from the simulation and training happens within a separate Python script. The network is then exported and accessed by the simulation player. This simplifies debugging and evaluating training data because the large number of data science libraries available in Python can be used directly.

4.1.5 Deep Q-Learning

The only requirement for a Neural Network to work as a DQN is that it takes the game state as input and has an output for each possible action. The internal structure can vary wildly depending on the specific problem. Popular examples of DQNs like [15] use 2D-convolution layers to take game images as input and run their output through final dense layers. This does not correspond directly to the game because the simulation provides its state as several floating-point values, not as an image.
Because the simulation does not provide images as input, the two-dimensional convolution layers are left out and only the dense layers are required to learn the

game. The number and width of fully connected layers is a multidimensional hyperparameter.

[11] proposed an improvement to Deep Q-Learning Methods to solve an overestimation bias found in some experiments with Deep Q-Learning. These issues were discussed by [26]: reward overestimation occurs because of the maximization step on estimated values inherent in q-learning. The agent picks the action with the highest q-value and exacerbates the influence of noise or error within the q-value estimations. Over time, this can lead to unrealistic q-values that far exceed the final rewards. [26] presents two worst-case scenarios and how this can lead to learning failure or bad performance. [10] proposes using two Q-function estimators trained on separate experience sets. Instead of using their own estimation for q-learning, they use the other estimation within the maximization step. [11] then demonstrates how this can be integrated into the Deep Q-Learning method by [15] by using the sparsely updated target network as a second estimator. The Q-Learning target

$$Y_t = R_{t+1} + \gamma Q(S_{t+1}, \underset{a}{\operatorname{argmax}} \{Q(S_{t+1}, a, \theta_t)\}, \theta_t) \tag{4.1}$$

is replaced with

$$Y_t = R_{t+1} + \gamma Q(S_{t+1}, \underset{a}{\operatorname{argmax}} \{Q(S_{t+1}, a, \theta_t)\}, \theta_t^-) \tag{4.2}$$

where θ_t are the trained network weights and θ_t^- are the static target network weights. They show that this improves performance on select Atari games in comparison with the regular method. This overestimation issue was also found when training the Foosball Table agent, so Double Q-Learning was implemented to counteract this.

[27] introduced a new network architecture for deep learning. They separated the fully connected layers into two network streams with the same input data. An advantage network with one output for each action, just like in a regular DQN [15], and a value network with only one output. In the end, the values are combined to represent the final Q values using

$$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + \left(A(s, a; \theta, \alpha) - \frac{1}{|A|} \sum_{a'} A(s, a'; \theta, \alpha) \right) \tag{4.3}$$

where s is the current state, a is the action, θ are the parameters of the convolutional layers of the DQN and α and β are the parameters of the advantage and value streams [27].

This step is done within the DQN and requires no additional modifications in the surrounding architecture. The idea behind this is that in many games there are states where the specific action an agent takes do not matter much, if at all, only the overall value of the state is relevant. This architecture allows the network to learn an overall value of a state separately from the separate action values. [27] found that this improves the efficiency of learning the state-value function and increases the robustness of the network against noise, which otherwise leads to rapid switching between actions of nearly identical value.

The network architecture used for the Foosball Table DQN agent includes both of these extensions. It consists of a number of fully connected neuron layers using the activation function ReLU. Batch normalization is done between them to accelerate training [2]. Adam is used as optimizer and Mean Squared Error as loss function.

4.1.6 Experience Replay

To combat the inherent instability in DQNs, [15] used a sliding-window experience buffer storing a limited number of state changes and rewards the agent witnessed. From this, training batches can be chosen randomly, avoiding biases inherent in sequential game episodes. It also allows a single state transition to be learned from multiple times.
This was further improved with Prioritized Replay, proposed by [20], which prioritizes situations with large temporal difference error. At first, experiences receive the maximum priority, to ensure they are picked at least once for training, and afterwards they receive a priority based on the difference in value from what the DQN predicted. Transitions are picked from the memory buffer with probability

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha} \tag{4.4}$$

where p_i is the priority of transition i and α determines how much influence the prioritization has [20].

4.1.7 Epsilon Decay

Epsilon Greedy is the simplest policy that adds exploration to the training of an agent. The idea is to set $0 \leq \epsilon \leq 1$ and using this as the probability to pick a random value, otherwise whatever action the agent picked is chosen. This forces the agent to experience situations it otherwise would have never reached and to learn from them. A broad selection of experiences is required to get an overview over which actions have positive and which have negative influence in specific situations.
Epsilon Decay improves on this by setting a large epsilon early on in training and decaying this value over time, until it reaches zero or a preset minimum value.

4.1.8 Bootstrapping

An inherent problem of Deep Q-Learning is jumping from a newly initialized network with random output values to the first success. Especially in highly random games like the Foosball Table, this poses a problem. The bootstrapping method is similar to the teacher policy used by [23], where for parts of the training of an air hockey striking agent, a basic policy guiding the DQN towards striking the puck instead of trying random actions that may not even get close to the puck.
For the Foosball Table game, the same algorithm presented earlier in Section 2.2.3 that is also used for evaluating the performance of agents is used to choose actions in early stages of training together with the random actions chosen by the epsilon greedy policy to guide the networks towards actions that influence the ball.

4.1.9 Solo

A unique problem for the Foosball Table player agent is the split into four DQNs. Each of them relies on the other three to perform well-enough, otherwise neither of them will succeed and reach highly rewarded states. Especially at the beginning of training, this is an issue, as the network has to filter not only through its own mostly random actions, but also those of the other three rods. This presents an obstacle that all four networks have to surpass before they can perform well.

One possible way to get around this is to initially train each network separately. For this, only one rod gets controlled by the DQN, and all others by a simple heuristic. This quadruples the number of games that have to be simulated for training data, but does not change the time this takes in any significant way. This is because the simulation time mostly consists of waiting for the neural network to finish. Each game step only has a single network forward calculation instead of four, and thus only takes about a quarter of the time.

4.1.10 Multi-Frame Input and Convolution

[15] not only used data from one game frame, but from a sequence of four, to add time data to the network input. While the Foosball Table agent does not use image data, time data is very relevant to be able to predict where the ball is going. The ball velocity is part of the input data, but by itself it is not enough to predict the ball position more than a few steps ahead. It might bounce off a wall or a player and change direction rapidly. Rods are completely impossible to predict without time data, as their velocity is not part of the input data. One way to solve this is adding multiple input frames to the data and moving those into one-dimensional convolution layers. These allow the network to do time-based convolution on the input sequences and extract velocities or other relevant data. This allows prediction of player positions and thus improves prediction of ball positions.
Two ways were implemented to add time data to the DQN agent: Multi-Frame and Convolution. Multi-Frame just means adding more input frames into the input layer of the network. This is simple and may work with a small number of frames, like the four frames used by [15]. Convolution adds four 1-dimensional convolution layers to the network. The input data is changed into a number of time series, each input value a separate channel into the first convolution layer. The convolution layers have kernel sizes of 21, 11, 5 and 3 and output channel sizes of 32, 64, 128 and 128. The result of the last layer gets fed into the regular linear part of the network. A batch norm layer is placed after each convolution layer to accelerate training [13] and ReLU is used as activation function.

4.1.11 Training Opponent

As the Foosball Table game relies heavily on an opponent that plays well enough that the ball does not get stuck regularly in its zone of influence, the choice of enemy may be essential for learning progress. A common strategy used for DQN agents is training them against themselves[22]. This approach works well in combinatorial games, but may prove troublesome in the Foosball Table simulation because the agent has to overcome the initial skill gap until it plays well-enough to enable learning more complex strategies. It may just get stuck doing random operations that do nothing while the ball stays outside its reach. However, it may also work because initial "random flailing" of the rods does lead to a dynamic game, from which the agent could learn what works and what does not.
To investigate this, three modes of training are implemented: Letting the agent play solely against itself, only playing against SB and playing half of the games against itself and half against SB.

4.1.12 Handicap

Assuming the trained agent stays below the performance of another algorithm, this can be used to find an approximate performance value. How much do you have to handicap the better-performing agent until its win rate drops below 50%? The first step towards such an algorithm is choosing a way to handicap an agent. One possibility is introducing noise into the input data it uses to evaluate the game state. This could be scaled up or down easily to increase or decrease the agent's performance. However, just choosing a simple noise pattern would introduce a temporal jitter into any agent's output, if this agent assumes valid input data. Also, different noise intensities would have to be chosen for each input value to correspond to what it represents, for example the ball position has a very different value range from a rod rotation angle.
A different approach to handicap an agent is delaying the input values. Simply replacing its current input value with one from n frames ago would handicap it in a way that makes it more difficult or impossible to deal with fast-moving situations, but still allows the agent to work with slower changes in game state. This introduces a predictability into the handicap algorithm that is preferable to the chaos of the noise approach. Additionally, it does not require any value tuning, only a simple integer value about how old of an input frame the agent receives.

A binary search can then be done to find the handicap value corresponding with an agent. A higher number of games for each step of the binary search improves the stability of the metric because doing a step in the wrong direction early on has a large influence on the overall result.
However, this metric has shown to be too susceptible to noise to be used for comparison. Very high game counts may compensate for this, but the time required to use it for evaluation would go beyond what is reasonable. The agent developed for this may still prove useful because it can scale its difficulty down if required, which could help with training the DQN agent.

4.1.13 Reward Function

The default reward function used for the DQN agent rewards reducing the distance to the enemy goal. Because of the many time steps required between scoring goals in the Foosball Table game, a sparse reward function that only rewards goals would lead to a very steep difficulty curve to actually scoring a goal. Instead, the distance to the enemy goal is minimized, so that shooting the ball forwards is rewarded even if it does not score a goal. Both reward functions are implemented and will be evaluated in the hyperparameter tuning experiments. They are referred to as “Distance” (Equation (4.5)) and “Goals” (Equation (4.6)).

$$r_{distance}\left(\overrightarrow{ball}\right) = \sqrt{\left(\overrightarrow{ball}.x - 1.2\right)^2 + \left(\overrightarrow{ball}.y - 0.34\right)^2} \tag{4.5}$$

$$r_{goals}\left(\overrightarrow{ball}\right) = \begin{cases} 1 & \text{if goal scored,} \\ -1 & \text{if goal received,} \\ 0 & \text{otherwise,} \end{cases} \tag{4.6}$$

4.1.14 Training algorithm

The training algorithm (Algorithm 3) takes in a target number of game steps to train, alongside a batch of options to define network architecture and other hyperparameters. It then runs through the same sequence of steps until this target number is exceeded. It generates input data by running the agent against SB, initially with a large handicap value to reduce its performance and

```
Data: $training\_steps$, $game\_count$, $game\_parameters$, $batch\_count$,
      $training\_parameters$
Result: $dqn$
1  $handicaps \leftarrow [0, 3, 7, 36, 69]$;
2  $cur \leftarrow 0$;
3  $buffer \leftarrow \perp$;
4  $dqn \leftarrow$ initialize();
5  while $cur < training\_steps$ do
6      $decay \leftarrow 1.0 - \frac{cur}{training\_steps} \cdot 10$;
7      $handicap \leftarrow handicaps[\max(0, decay \cdot (handicaps.\text{size} - 1))]$;
8      $\epsilon \leftarrow \max(0.1, decay)$;
9      $data \leftarrow$ run_games($\epsilon, game\_count, game\_parameters$);
10     $buffer$.append($data$);
11     while $buffer.size < 1000000$ do
12         $data \leftarrow$ run_games($\epsilon, game\_count, game\_parameters$);
13         $buffer$.append($data$);
14     end
15     $buffer$.append($data$);
16     $batch \leftarrow buffer$.select_minibatch();
17     $frames\_done \leftarrow$
         train($dqn, batch, train\_count, training\_parameters$);
18     $cur \leftarrow cur + frames\_done$;
19 end
20 evaluate($dqn$);
```

Algorithm 3: Training Algorithm

ease initial training (lines 6-14). This handicap value gets reduced to 0 over the first 10% of training steps, to initially explore the search space against a weaker opponent. Afterwards, the rest of training is dedicated to moving the DQNs approximation closer to the actual Q-values while training on data gathered with the actual policy. The probability ϵ, with which the agent picks a random action, gets reduced simultaneously with the handicap value to a minimum of 0.1. The list of handicap values (line 1) is static and was picked by evaluating the performance of SB against its handicapped self and choosing a number of approximately linearly distributed values. They are only relevant in the Opponent Mode experiments.
All the game steps from this are stored by the algorithm and transferred to the training buffer. The training buffer is initially filled up to the maximum size, and afterwards older data is deleted when adding to it.
Training (line 17) is done on minibatches, selected from the replay buffer (line 16). Each of the four networks has its own replay buffer and gets trained on separate data, although taken from the same games. This was done to allow separate reward functions for each rod, but ultimately not used for the experiments. Finally, when the algorithm has run through the predefined amount of input frames, it stops and evaluates the result (line 20).

4.2 Experiments

4.2.1 Hyperparameters

Table 4.1: Training Hyperparameters

Hyperparameter	Value
Learning rate	0.0001
ϵ range	0.1-0.9
γ	0.99
Experience buffer size	1000000
Total training steps	2000000
Batch size	512
Target network update frequency	512
Max episode length	30s
Training session steps	256

The DQN agent has many possible hyperparameters and more combinations of values than can be tested within the scope of this thesis. To find a working configuration, a similar hyperparameter experiment to the previous EA chapter is run, with a selected number of parameters. A number of parameters are not changed throughout the experiments. These are shown in Table 4.1. Their values were picked while developing the algorithm. A high discounting factor γ was picked because each action of the game has only a small influence on the result and rewards may be far in the future. A buffer size of one million time steps was picked because it was used in [15]. A batch of 512 entries are picked from the replay buffer for each training step. The target network is updated after each batch. 256 training steps are done before new data from 16 games is added to the buffer. These games last up until a goal is scored or the maximum episode length of 30 seconds is reached.
Preliminary tests have shown the networks struggle to learn the game, so a variety of configurations is tested for convergence the resulting agent's performance against SB. The most important parameters are the layout of the neural networks and the number of input frames. Both of these parameters alone have ranges too vast to explore fully, so a small selection of values is chosen from successful values in related work and combined with handpicked values around them. A default value is chosen and for each experiment only one hyperparameter is varied between experiment runs. The experiments are run for two million input frames and then evaluated and checked for convergence.

Table 4.2: DQN Experiment Hyperparameters

Hyperparameter	Range	Experiment Default
Neuron Layout	see Table 4.3	500-400-300-200-100
Absolute translation actions	False, True	False
Bootstrapping	False, True	False
Multi-Frame/Convolution	1, 4, 30, 60, 120	120
Opponent Mode	SB, Half, Self	Self
Reward Function	Distance (0), Goals (1)	Distance (0)

Table 4.2 shows the configurations that were tried within the hyperparameter experiments. Neuron layouts only set the fully connected layers and were picked from experiments and from values used in related DQN papers. The multi-frame and convolution values were picked to evaluate a broad range of

Table 4.3: DQN neuron layer values for the hyperparameter experiments

Index	Name	Linear Neuron Count
1	512	512
2	1024	1024
3	512-512	1024
4	500-400-300-200-100	1500
5	512-512-512	1536
6	1024-1024	2048
7	512-512-512-512	2048
8	1024-1024-1024	3072
9	1024-1024-1024-1024	4096

the possible values. The values 1 and 4 are with convolution disabled while the others make use of it.

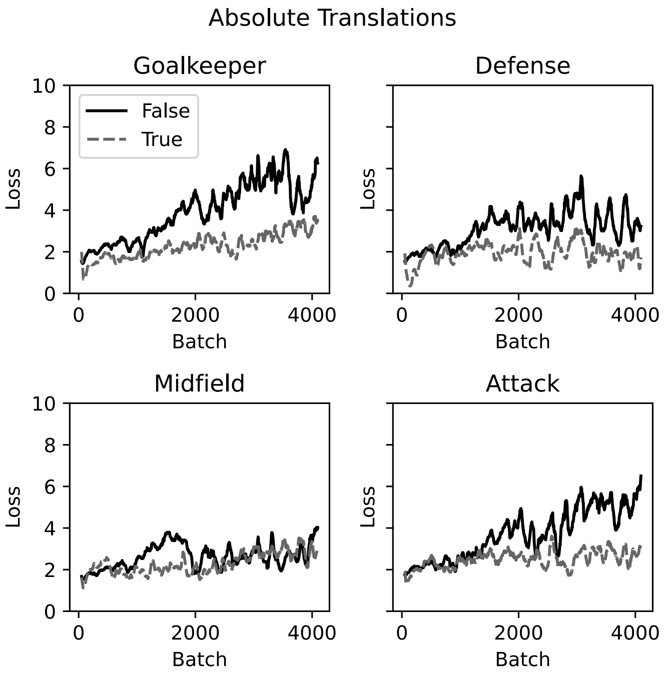

Figure 4.3: Training loss of the relative or absolute experiments

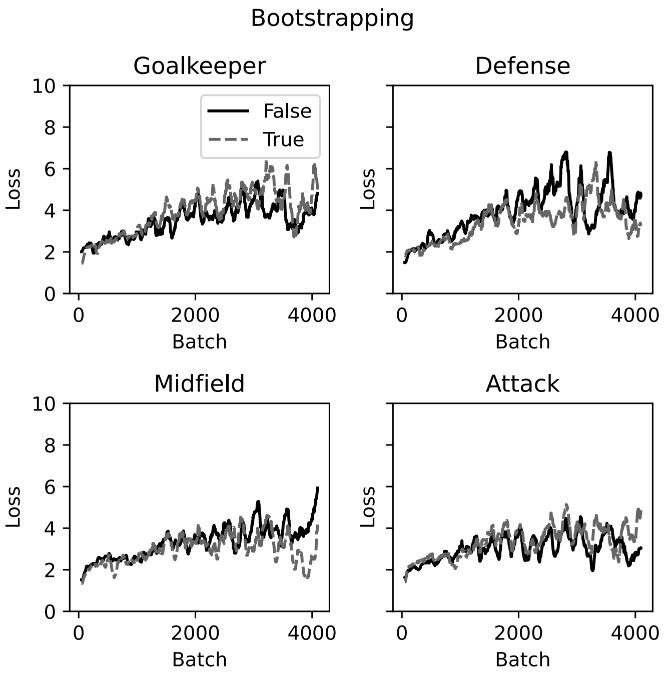

Figure 4.4: Training loss of the bootstrapping experiments

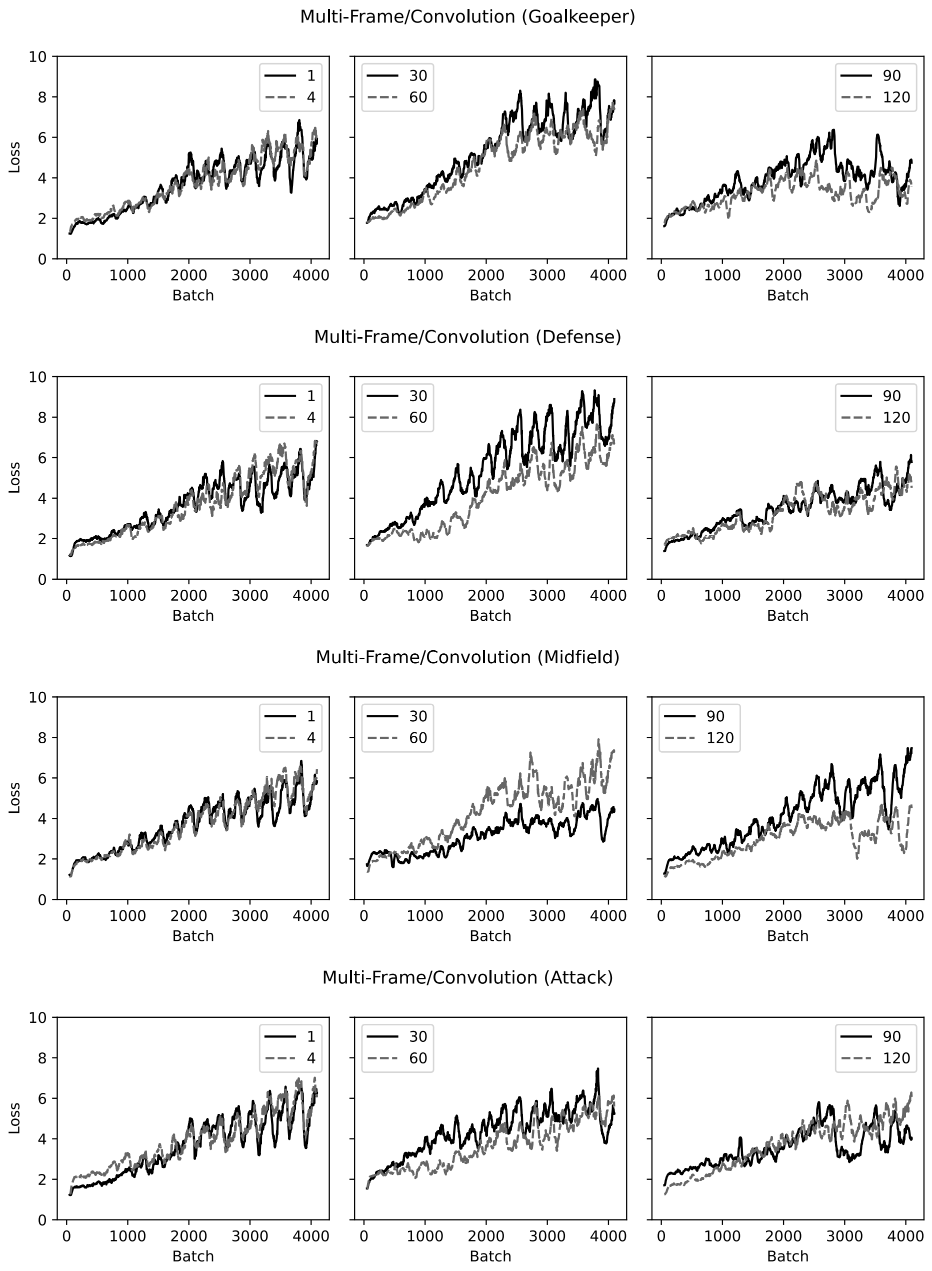

Figure 4.5: Training loss of the multi-frame/convolution experiments

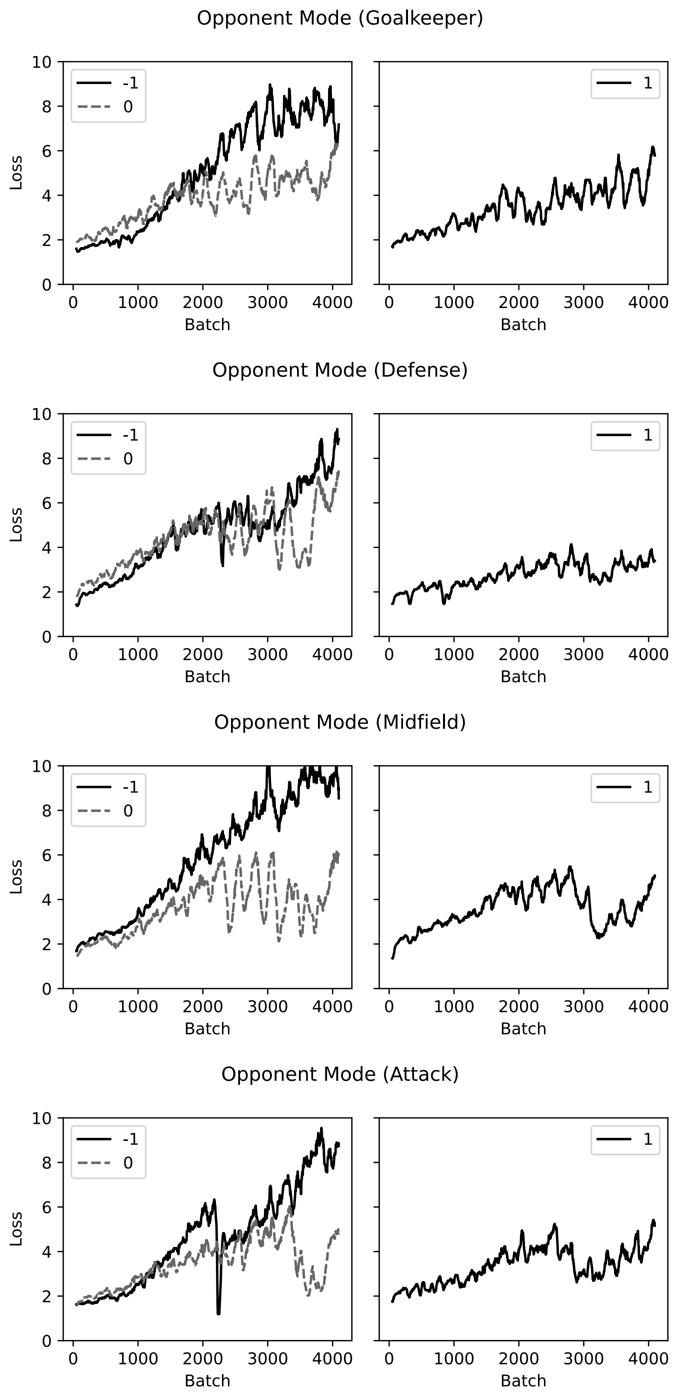

Figure 4.6: Training loss of the opponent mode experiments

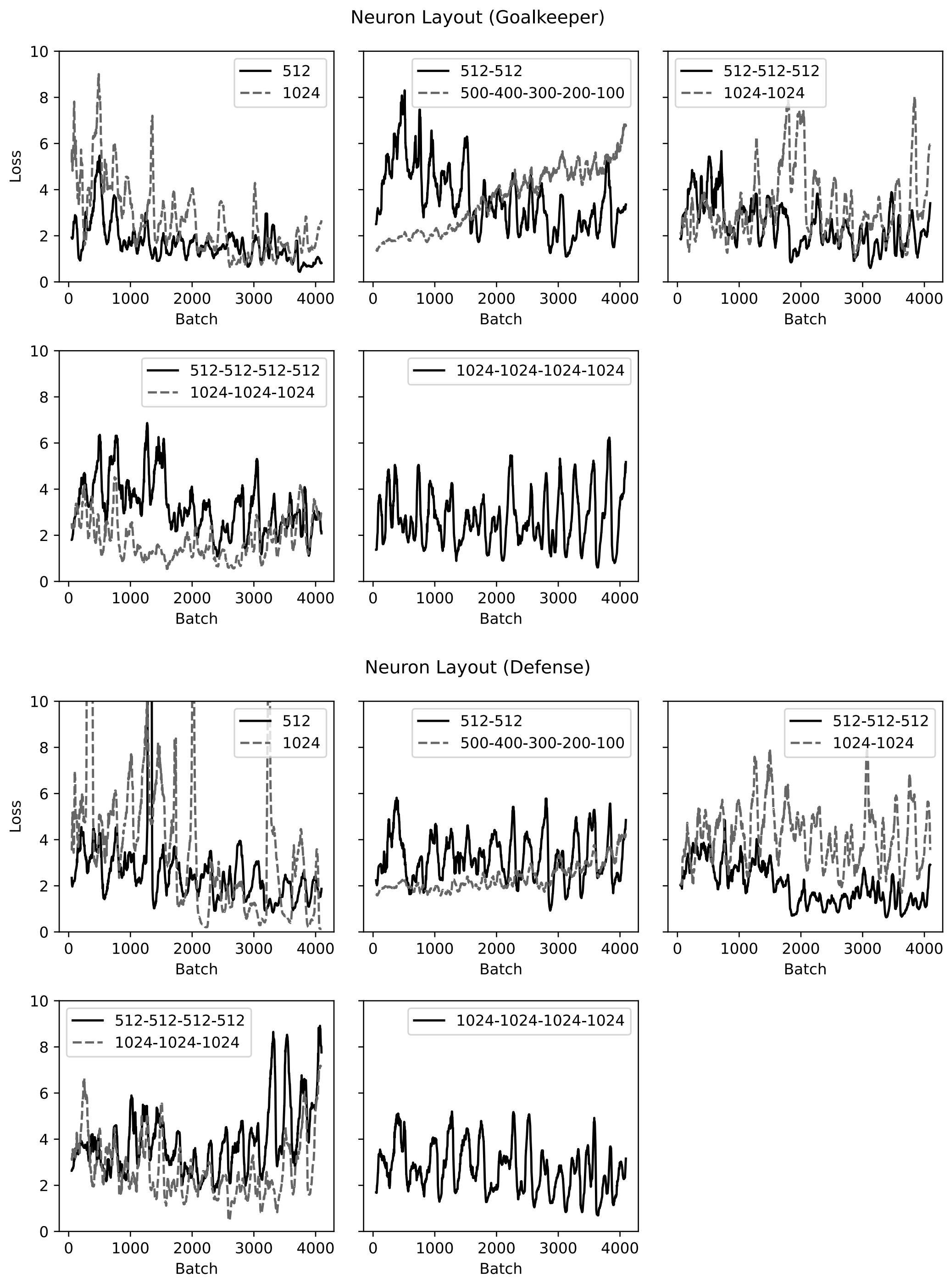

Figure 4.7: Training loss of the neuron layer experiments (Part 1)

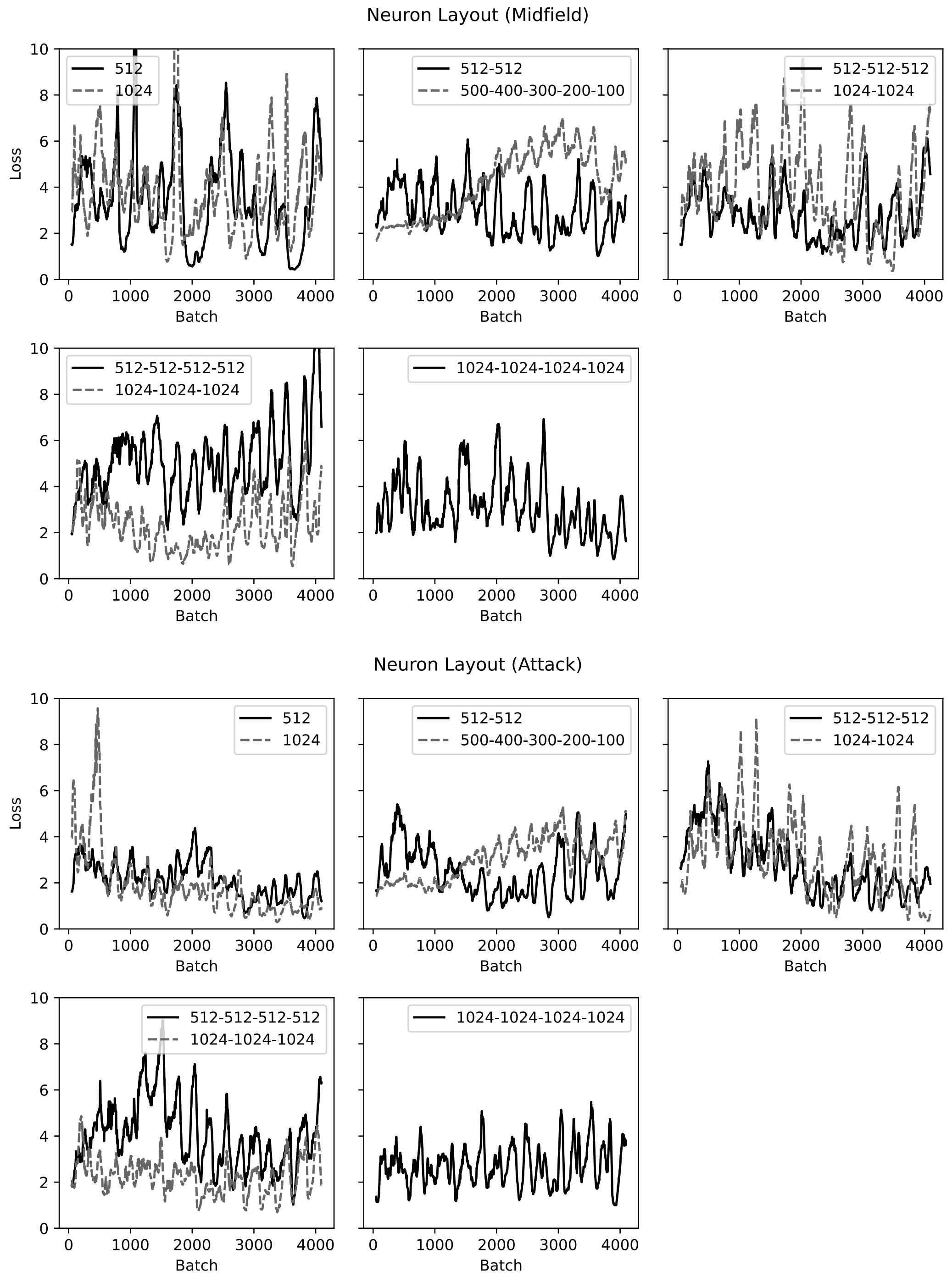

Figure 4.8: Training loss of the neuron layer experiments (Part 2)

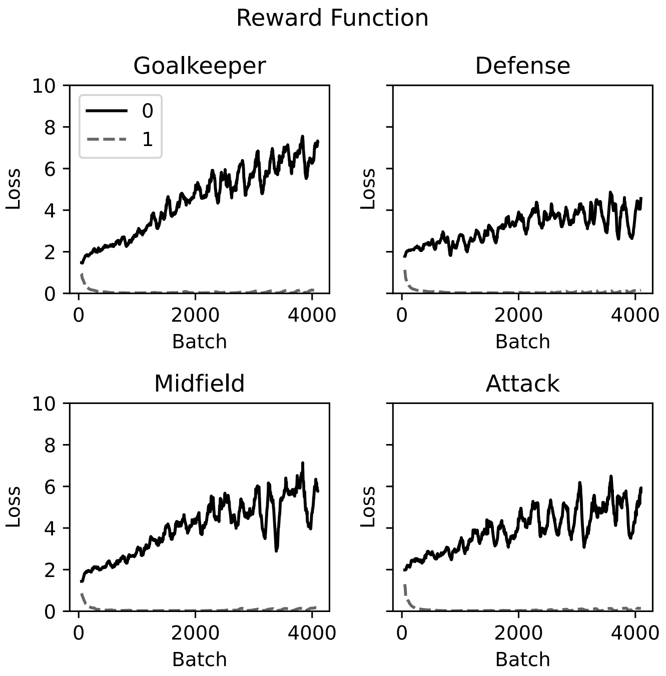

Figure 4.9: Training loss of the reward function experiments

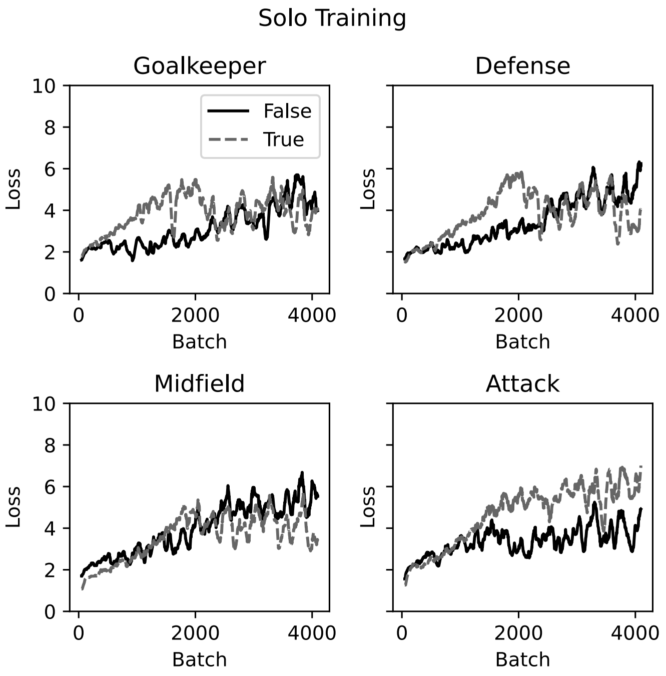

Figure 4.10: Training loss of the Solo experiments

Figures 4.3 to 4.10 show the neural networks struggling to learn in nearly all configurations. The only experiment that converged is the alternative scoring function. Divergence does not necessarily mean bad performance, especially with the regular dense scoring function, where values can be off, but as long as they are off by similar amounts the Q-value order might still be correct or close enough. However, the consistent and increasing error values during training are not promising. Absolute translation action diverged less than relative actions (Figure 4.3). It seems that even with more data, in the multi-frame/convolution experiment (Figure 4.5) the DQNs are unable to find a consistent pattern of Q-values that leads to a good strategy. Playing only against itself, against reference SB or both does not seem to make a difference for this either (Figure 4.6). Training only on games against SB has higher loss over time, but all three experiments diverge. Training the networks separately with SB controlling the other rods (Figure 4.10) does not improve the convergence. Neither does training the networks on SB behavior initially (Figure 4.4). That means playing better during training does not make it easier to learn, at least not enough to avoid the divergence.

The various neuron configurations in Figures 4.7 and 4.8 show a few differences, especially between the default "triangle" configuration 500-400-300-200-100 and those with multiple layers of the same size. The "Triangle" configuration diverges further than the others. It shows an upwards trend of worse and worse loss over time in most experiments, while the same-size layers stay at their level of divergence.
The only experiment that converged was with the alternative reward function "Goals" (1 in Figure 4.9). This sparse reward function makes it easier for the networks to find a consistent configuration of Q-values than rewarding reduced distance to the opponent's goal. The evaluation will show whether it found a good solution.

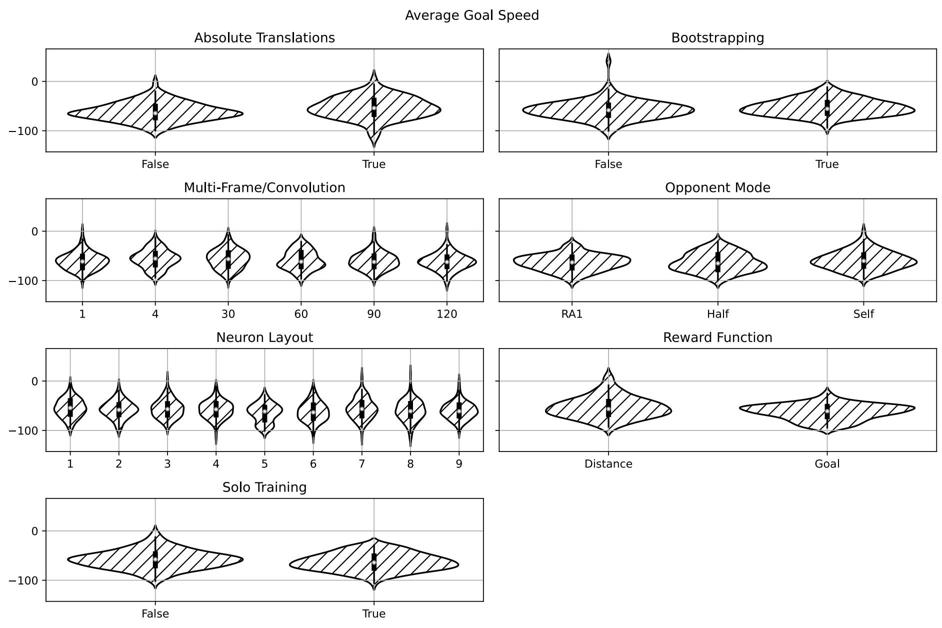

Figure 4.11: Average Goal Speed results of the DQN hyperparameter experiments against SB

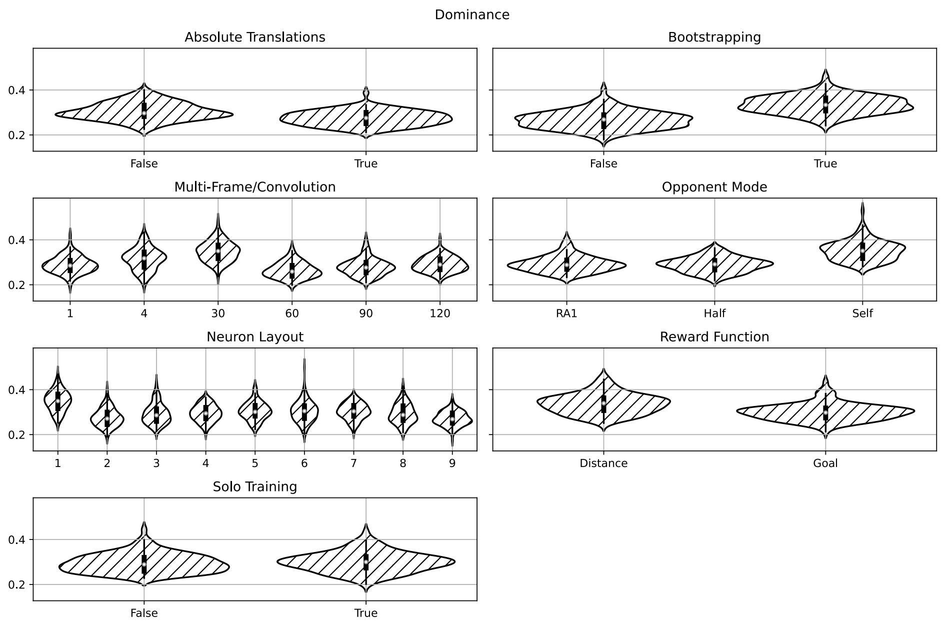

Figure 4.12: Dominance results of the DQN hyperparameter experiments against SB

Figures 4.11 and 4.12 show the average goal speed and dominance results for the experiments. The neuron configurations are marked using the indices from Table 4.3. Neither metric shows much variation for the DQN agents. All trained agents perform poorly, even when compared only to the weaker reference algorithm SB. These do show small variations in the graphs, like the dominance with or without bootstrapping enabled, the improvement in dominance without convolution for one or four frames up to convolution with 30 input frames and the subsequent drop with 60 frames. These differences are more likely to be the training algorithm finding a slightly more advantageous random behavior to adopt than an actual improvement due to the hyperparameter. Notably, the only converging experiment with the alternative scoring function also performs badly, which suggests it got stuck in a bad local optimum and did not perform any better than its diverging counterparts. In fact, both metrics suggest it performed slightly worse, but this may just be variance between the experiments.

4.3 Discussion

Using DQN to control an entire side of the Foosball Table did not work out as intended. The only configuration that converged was with the “Goal” reward function, but it did not find a Q-value configuration that leads to successful play. The other experiments did not converge and also performed no better than a random configuration. Watching them play just shows random movements, with no real concept behind it. The DQN agent in the configuration implementation does not work. Most likely, the dynamic physics-based environment of the Foosball Table and the small influence or no influence at all that an individual action has provides too much difficulty for the DQNs to distinguish and learn from.

5 Direct Comparison

This chapter contains an evaluation of the agents against each other. The initial plan was to compare the RHEA agent with the DQN agent here, but because the DQN implementation was not successful, instead the three agents of the RHEA chapter, RS, EA and RHEA, are compared with each other and with the two reference algorithms SB and IB.

Figure 5.1 shows the results of a direct comparison between all the previously

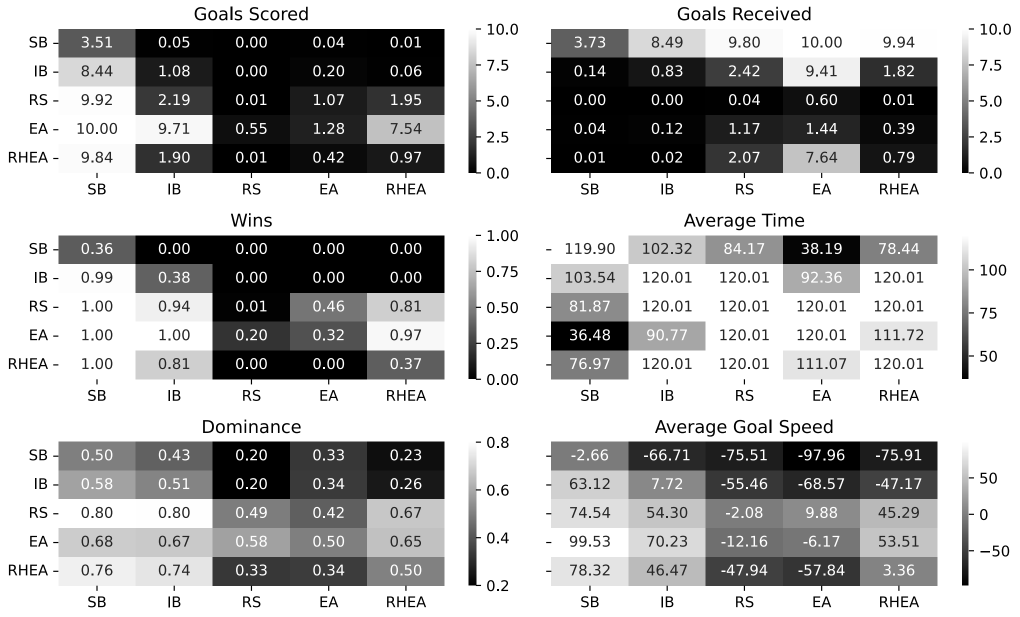

Figure 5.1: Heatmaps showing the performance over six metrics of the agent on the y-axis against the agent on the x-axis

mentioned agents. 100 games were played for every combination of two agents, terminating either after 120 seconds or 10 goals scored by one side.

SB has the worst results against the other agents. It scores the fewest goals against the others and takes the most. In fact, the only agent it reliably scores

against is itself. Against the other agents, it only receives goals, even against the only slightly better IB. The improvement in defensive capability of IB over SB is also very apparent in the heatmap showing the received goals. Whereas SB lost many games on goals received, IB manages to drag the games out to the time limit most of the time. Only against the EA agent does it lose decisively on average, but still manages to hold out more than twice as long as SB.
The RS algorithm performs very well, easily beating SB and IB. It performs very well even against the evolutionary algorithms, beating the RHEA agent in 81% of the games and the EA agent in 46% of the games. Against itself, it draws most of the time, which shows that it is better at defending than attacking. Surprisingly, in this case that means better at avoiding taking goals, not at pushing the ball onto the opponent's side of the field, as the RS agent has a worse dominance than EA, but a slightly higher chance of scoring. This is seen in the low, but positive average goal speed.
The EA agent performs best out of the five agents. Its focus on attacking compared to the RS agent is seen in the relatively high win rate against itself. The RS agent draws most of the time, while the EA agent wins 32% of the games against itself, meaning it drew 36% of the games. Interestingly, while it beats the simpler algorithms and the RHEA agent the most decisively, it loses more games than it wins against the more defensively oriented RS. This shows how adapting your play style to the opponent can have a large impact in the Foosball Table simulation, and likely also the physical game.
The RHEA agent performs better than SB and IB, but not as well as RS and EA. Its average goal speed values against the other agents show that it is about as good at scoring against SB and IB as the RS agent, although it wins less often against IB. It receives many more goals on average from the EA agent than from the RS agent, but manages to score more against EA than RS.

6 Conclusion and Future Work

This thesis examined the performance of a RHEA and a DQN agent within a highly dynamic two-dimensional simulation created to approximate Foosball Table games. While developing the RHEA agent, an EA agent based on a slightly different principle was also created and is evaluated alongside the others. Multiple configurations of hyperparameters were examined for their influence on the agent, and a final configuration was picked based on their performance and the time the agent requires. Because one future goal of this thesis is transferring the gathered knowledge onto a Foosball Table robot, each algorithm has to be able to run in real-time, within less than 8.3ms to work with the 120 frames per second camera on the actual hardware. All agents stay within this time limit, while the RHEA agent is the most computationally expensive.

Question 1 How well can a Rolling Horizon Evolutionary Algorithm algorithm play the game?

The RHEA concept did not perform as well as predicted, but it is capable of playing the game competitively. Abstracting the search space in a way that allows an algorithm to search it quickly and to plan ahead for an algorithm like RHEA, while not limiting the reaction speed, has been the largest problem for this. The highly dynamic environment of the Foosball Table simulation leads to a lower value of precise short-term planning as is done by RHEA because the plans it develops have a low chance of working in the way it intends. The simpler input strategy of the EA agent leads to significantly more success in the game, even with a very short look-ahead for the fitness function. An essential insight of this is that future work should account for the dynamic environment by focusing on rough long-term (few seconds at most) plans to outplay the opponent and use simple short-term planners to follow this plan. This is similar to the PTSP work in [18], where the order of waypoints to pass is decided beforehand by a global planner and the short-term algorithms are used to execute the plan. The focus there was on the short-term algorithms,

which were used as a basis to develop the EA and RHEA agents in this thesis. The long-term planning was neglected for this thesis, but the results showed that there may be potential in this direction that should be investigated in the future.
This dynamic environment also led to difficulty for the EAs to use their advantage in guiding the random search towards advantageous positions in the search space. Random Search already performed very well in the PTSP comparison [18], where the agents directly influenced the movement of a ship. On the Foosball Table, the ball has to be influenced through movement and rotation of the player rods. This adds a significant additional step to the search space that makes it hard to search efficiently.
This also corresponds with the findings by [9], which showed competitive performance, but similar or better results with a random search algorithm. The hyperparameter experiments have shown that higher look-ahead times for the fitness function simulation lead to lower performance of the algorithm because the gap between opponent behavior in the game and the simulation becomes too large. This is a problem for multiplayer games. A solution for this could be adding opponent modelling to the fitness function as proposed by [24], which trains a model of the opponent during play that improves the value of the simulation result.

Question 2 How well can a Deep Q-Learning algorithm play the game?

DQNs have been shown to be able to solve difficult problems with search spaces too large for more traditional methods like EA, like Atari games [15]. A similar approach was attempted as a part of this thesis, but ultimately unsuccessfully. The difficult dynamic environment of the Foosball Table has provided too much of an obstacle for the neural networks to grasp. No configuration of parameters tested has lead to an increase in the level of performance of the agent beyond the baseline of random play. Using a similar reward function to the EA agents, rewarding the DQN agent for moving the ball closer to the enemy goal, leads to divergence during training. Rewarding only goals and leaving it to the neural networks to figure out how to get there leads to convergence, but only to a local optimum that does not lead to successful play.
The DQN agent implemented in this thesis was incapable of solving the problem and playing the game competitively.
Some problems were noted during the evaluation of the DQN approach. The approach of filling the replay buffer initially before starting to learn is likely suboptimal, because it means a large part of the buffer lags far behind the

policy dictated by the current Q-values. A better balance between training steps and buffer updates could be tried on this problem in future work. One theory is that the initial gap from random behavior to manipulating the ball to achieve the agent's goals is too large to overcome using an approach similar to that used on Atari games by [15]. There are many parameters and design choices involved in applying Deep Q-Learning to a new problem, so it is possible that a different approach would yield better results. There have also been further improvements proposed for Deep Q-Learning like the Rainbow architecture by [12], which combines multiple proposed enhancements, or a new approach to deal with divergence by [1], which could be applied to the problem.

Question 3 Which game AI algorithms work on the Foosball Table game, and how do they compare?

Fully answering this question is beyond the scope of this thesis, but two approaches were tested and compared to answer a part of it. Random Search, Evolutionary Algorithm and Rolling Horizon Evolutionary Algorithm algorithms are capable of playing the game competitively just by optimizing the ball position towards the opposite goal using a short-term simulation as fitness function. To apply this on the hardware Foosball table the simulation would have to be tuned to model reality more closely, but once that is done these algorithms are promising for quick short-term optimizations. The small difference between Random Search and the other two shows that there is not much room for improvement using just short-term optimization in the game. Long-term optimization is made very difficult by the dynamic environment and the unknown actions of the opponent, as the DQN approach attempted in this thesis did not work out.

Lastly, once the player detection is finished on the automated Foosball Table, the simulation could be tuned to resemble the physical properties of the actual game as much as possible. Then, the EA and RHEA agents could be evaluated on the hardware and provide interesting results on the reality gap between the table and the simplified two-dimensional simulation.

Bibliography

[1] Joshua Achiam, Ethan Knight, and Pieter Abbeel. Towards characterizing divergence in deep Q-learning, March 2019. doi: 10.48550/arXiv.1903.08894.

[2] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization, July 2016. doi: 10.48550/arXiv.1607.06450.

[3] Marc G. Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The Arcade Learning Environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, June 2013. ISSN 1076-9757. doi: 10.1613/jair.3912.

[4] Murray Campbell, A. Joseph Hoane, and Feng-hsiung Hsu. Deep Blue. *Artificial Intelligence*, 134(1):57–83, January 2002. ISSN 0004-3702. doi: 10.1016/S0004-3702(01)00129-1.

[5] Erin Catto. Box2D. URL `https://box2d.org/`. Accessed on 2022-10-23.

[6] Stefano De Blasi, Sebastian Klöser, Arne Müller, Robin Reuben, Fabian Sturm, and Timo Zerrer. KIcker: An industrial drive and control foosball system automated with deep reinforcement learning. *Journal of Intelligent & Robotic Systems*, 102(1):20, April 2021. ISSN 1573-0409. doi: 10.1007/s10846-021-01389-z.

[7] Kalyanmoy Deb and Ram Bhusan Agrawal. Simulated binary crossover for continuous search space. *Complex Systems*, 9(2):115–148, 1995.

[8] Elhadji Amadou Oury Diallo, Ayumi Sugiyama, and Toshiharu Sugawara. Learning to coordinate with deep reinforcement learning in doubles pong game. In *2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 14–19, December 2017. doi: 10.1109/ICMLA.2017.0-184.

[9] Raluca D. Gaina, Jialin Liu, Simon M. Lucas, and Diego Pérez-Liébana. Analysis of vanilla rolling horizon evolution parameters in general video game playing. In Giovanni Squillero and Kevin Sim, editors, *Applications of Evolutionary Computation*, Lecture Notes in Computer Science, pages 418–434, Cham, 2017. Springer International Publishing. ISBN 978-3-319-55849-3. doi: 10.1007/978-3-319-55849-3_28.

[10] Hado van Hasselt. Double Q-learning. In *Proceedings of the 23rd International Conference on Neural Information Processing Systems - Volume 2*, NIPS'10, pages 2613–2621, Red Hook, NY, USA, December 2010. Curran Associates Inc.

[11] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double Q-learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 30(1), March 2016. ISSN 2374-3468. doi: 10.1609/aaai.v30i1.10295.

[12] Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning, October 2017. doi: 10.48550/arXiv.1710.02298.

[13] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, March 2015. doi: 10.48550/arXiv.1502.03167.

[14] Rudolf Kruse, Sanaz Mostaghim, Christian Borgelt, Christian Braune, and Matthias Steinbrecher. *Computational intelligence: A methodological introduction.* Texts in Computer Science. Springer International Publishing, Cham, 2022. ISBN 978-3-030-42226-4 978-3-030-42227-1. doi: 10.1007/978-3-030-42227-1.

[15] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with deep reinforcement learning, December 2013. doi: 10.48550/arXiv.1312.5602.

[16] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit

Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL `https://proceedings.neurips.cc/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf`.

[17] Diego Perez, Philipp Rohlfshagen, and Simon M. Lucas. The physical travelling salesman problem: WCCI 2012 competition. In *2012 IEEE Congress on Evolutionary Computation*, pages 1–8, June 2012. doi: 10.1109/CEC.2012.6256440. ISSN: 1941-0026.

[18] Diego Perez, Spyridon Samothrakis, Simon Lucas, and Philipp Rohlfshagen. Rolling horizon evolution versus tree search for navigation in single-player real-time games. In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, GECCO '13, pages 351–358, New York, NY, USA, July 2013. Association for Computing Machinery. ISBN 978-1-4503-1963-8. doi: 10.1145/2463372.2463413.

[19] Tobias Rohrer, Ludwig Samuel, Adriatik Gashi, Gunter Grieser, and Elke Hergenröther. Foosball table goalkeeper automation using reinforcement learning. In *Proceedings of the LWDA 2021 Workshops*, volume 2993, Munich, Germany, 2021. CEUR Workshop Proceedings. URL `http://ceur-ws.org/Vol-2993/paper-17.pdf`.

[20] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay, February 2016. doi: 10.48550/arXiv.1511.05952.

[21] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, January 2016. ISSN 1476-4687. doi: 10.1038/nature16961.

[22] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering

the game of Go without human knowledge. *Nature*, 550(7676):354–359, October 2017. ISSN 1476-4687. doi: 10.1038/nature24270.

[23] Ayal Taitler and Nahum Shimkin. Learning control for air hockey striking using deep reinforcement learning. In *2017 International Conference on Control, Artificial Intelligence, Robotics & Optimization (ICCAIRO)*, pages 22–27, May 2017. doi: 10.1109/ICCAIRO.2017.14.

[24] Zhentao Tang, Yuanheng Zhu, Dongbin Zhao, and Simon M. Lucas. Enhanced rolling horizon evolution algorithm with opponent model learning: Results for the fighting game AI competition, March 2020. doi: 10.48550/arXiv.2003.13949.

[25] Sebastian Thrun. *Probabilistic robotics.* Intelligent robotics and autonomous agents. MIT Press, Cambridge, Massachusetts, 2006. ISBN 978-0-262-30380-4.

[26] Sebastian Thrun and Anton Schwartz. Issues in using function approximation for reinforcement learning. In *Proceedings of the Fourth Connectionist Models Summer School*, Hillsdale, NJ, December 1993. Lawrence Erlbaum Publisher.

[27] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. Dueling network architectures for deep reinforcement learning. In *Proceedings of The 33rd International Conference on Machine Learning*, pages 1995–2003. PMLR, June 2016. ISSN: 1938-7228.

[28] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3):279–292, May 1992. ISSN 1573-0565. doi: 10.1007/BF00992698.

[29] Hongming Zhang and Tianyang Yu. AlphaZero. In Hao Dong, Zihan Ding, and Shanghang Zhang, editors, *Deep Reinforcement Learning: Fundamentals, Research and Applications*, pages 391–415. Springer, Singapore, 2020. ISBN 9789811540950. doi: 10.1007/978-981-15-4095-0_15.

Declaration of Authorship

I hereby declare that this thesis was created by me and me alone using only the stated sources and tools.

Ruben Ortlam Magdeburg, October 25, 2022