Universiteit Leiden

Master Computer Science

Comparing AI approaches for Tetris Link

Name: Matthias Müller-Brockhausen
Student-no: s2084740

Date: 05 / 02 / 2020

1st supervisor: Mike Preuss
2nd supervisor: Aske Plaat

Master's Thesis in Computer Science

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract. AI for well-known games such as Chess and Go has experienced decades of research focusing on playing just a little better. We will look into Tetris Link a board game that is still lacking any scientific analysis. Besides assessing the branching factor (162) and first player advantage (not significant), we compare different approaches to automatically master the game, namely: Reinforcement Learning (RL), Monte Carlo tree search (MCTS) and heuristics. We let them play against each other in a tournament, and the heuristics are strongest followed by MCTS and RL. We also let real people (n=7) familiar with the game play against the heuristic, and the AI lost the majority of the matches.
For RL, we go into detail on designing a working environment and test the effect of decisions on training success. In our findings, using different encodings for the observation (image vs array) does not affect training. For the reward function, it was best to include many factors such as scolding for unsuitable inputs instead of only giving the score or only a reward at the end of an episode.

Keywords: Tetris Link, Heuristics, Monte Carlo tree search, Reinforcement Learning, RL Environment, OpenAI Gym

Table of Contents

		Page
1	Introduction	4
	1.1 Research Questions	4
2	Related Work	4
	2.1 Tetris Link	4
	2.2 Monte Carlo tree search	6
	2.3 Reinforcement Learning	6
3	Tetris Link	7
	3.1 Verifying completability	9
	3.2 Branching Factor	9
	3.3 First move advantage	11
	3.4 Strategies	13
4	Methods	16
	4.1 Monte Carlo tree search	16
	4.1.1 Game tree	16
	4.1.2 Algorithm	16
	4.1.3 UCT / UCB1	17
	4.1.4 PoolRAVE	18
	4.1.5 Transposition Table	18
	4.2 Reinforcement Learning	20
	4.3 Heuristics	21
	4.4 Skill Evaluation	22
5	Design	23
	5.1 Implementation	23
	5.1.1 Performance	23
	5.2 Reinforcement Learning Environment / Gym	24
	5.2.1 Observation space	24
	5.2.2 Action space	25
	5.2.3 Reward function	26
	5.3 Reinforcement Agents & Training	27
	5.3.1 Algorithms	27
	5.3.2 RL Agents	27
	5.3.3 Hyperparameters	28
	5.3.4 Failed attempts	28
	5.3.5 NAN Problems	29
6	Evaluation	30
	6.1 Measurements	30
	6.2 Human Play	30
	6.2.1 Setup	30
	6.2.2 Results	31
	6.2.3 Bugs	33

6.3 Reproducibility . . . 34
6.3.1 Setup . . . 34
6.3.2 Results . . . 35
6.4 Observation Space . . . 36
6.4.1 Setup . . . 36
6.4.2 Results . . . 36
6.5 Reward Function Effectiveness . . . 37
6.5.1 Setup . . . 37
6.5.2 Results . . . 37
6.6 MCTS Effectiveness . . . 38
6.6.1 Setup . . . 38
6.6.2 Results . . . 38
6.7 RL Agents Training . . . 39
6.7.1 Setup . . . 39
6.7.2 Results . . . 40
6.8 Tournament . . . 44
6.8.1 Setup . . . 44
6.8.2 Results . . . 44
6.9 Verifying the MCTS Implementation . . . 47
6.9.1 Setup . . . 47
6.9.2 Result . . . 48
7 Conclusion . . . 50
7.1 Contributions . . . 51
7.2 Future Work / Limitations . . . 53
8 References . . . 54
Appendices . . . 59
A Additional Performance Information . . . 59
A.1 Desktop Hardware Specifications . . . 59
A.2 Server Hardware Specifications . . . 59
A.3 Rust performance optimisations . . . 59
B Hyperparameters . . . 60
B.1 PPO2 Search . . . 60
B.2 RL-Selfplay Hyperparameters . . . 61
B.3 RL-Heuristic Hyperparameters . . . 61
B.4 MCTS Search parameters . . . 61
B.5 Unusable RL Algorithms . . . 61
C Code Changes to Open Source Projects . . . 63
C.1 Reproducibility . . . 63
C.2 Self Play . . . 65
C.3 Self Play nobleed . . . 67
D Game Field Representations . . . 70
D.1 Numerical Field Representation . . . 70
D.2 Reinforcement Learning Field Representation . . . 71
D.3 JSON Gamelog . . . 72

1 Introduction

Board games are not only popular among players but also researchers. Scientific papers that analyse the game of Chess date back centuries[69,85]. Already in 1826 were they writing about machines that supposedly played Chess automatically[8], but they were not sure whether it was still operated somehow by people. Nowadays, we know for sure that there are algorithms that can, without the help of humans, automatically decide on a move. In the game of Go, these algorithms even beat the world champion[7]. In this paper, we want to investigate the game and automated play approaches for the board game Tetris Link, because it has not been done yet (section 2.1). For that, we implement a digital version of the board game (section 5.1) and take a brief look at the games theoretic aspects (section 3). Based on that theory, we develop a heuristic playing agent (section 4.3). To have an estimate of the heuristic's skill, we let humans play against the heuristic (section 6.2). For inspiration on other AI approaches, we look at the previously mentioned success in Go, namely *AlphaGo*, which combines Reinforcement Learning (RL) and Monte Carlo tree search (MCTS)[82]. We separately look at RL (section 4.2) and MCTS (section 4.1) as options to implement agents. Because we need to design an environment for the RL agent, we assess the impact of choices like observation (section 6.4) and reward (section 6.5) on training success. We also briefly touch the topic of reproducibility (section 6.3) in the deep learning field. For MCTS, we analyse the importance of the default policy (section 6.6) and underline the effect of the branching factor on a random default policy (section 6.9). In the end, we compare the performance of these agents using a skill rating algorithm (section 4.4) after letting them compete against each other in a tournament (section 6.8).

1.1 Research Questions

In our journey towards building game playing agents for Tetris Link, visualised in Figure 1, and written in section 7.1, we stumble upon some questions relevant for different scientific fields. Because many of these questions require specific knowledge of these fields we pose and answer them at the end of this thesis.

2 Related Work

2.1 Tetris Link

Searching Google Scholar for the exact quoted term "Tetris Link" brings up three papers in total[23]. Two of them are entirely unrelated and talk about optical refrigerators, where there seems to be a phenomenon that is called "Tetris", which they "link" to a specific behaviour. The third paper[65] deals with the board game Tetris Link:

> Does experiential learning improve learning outcomes in an undergraduate course in game theory? – a preliminary analysis

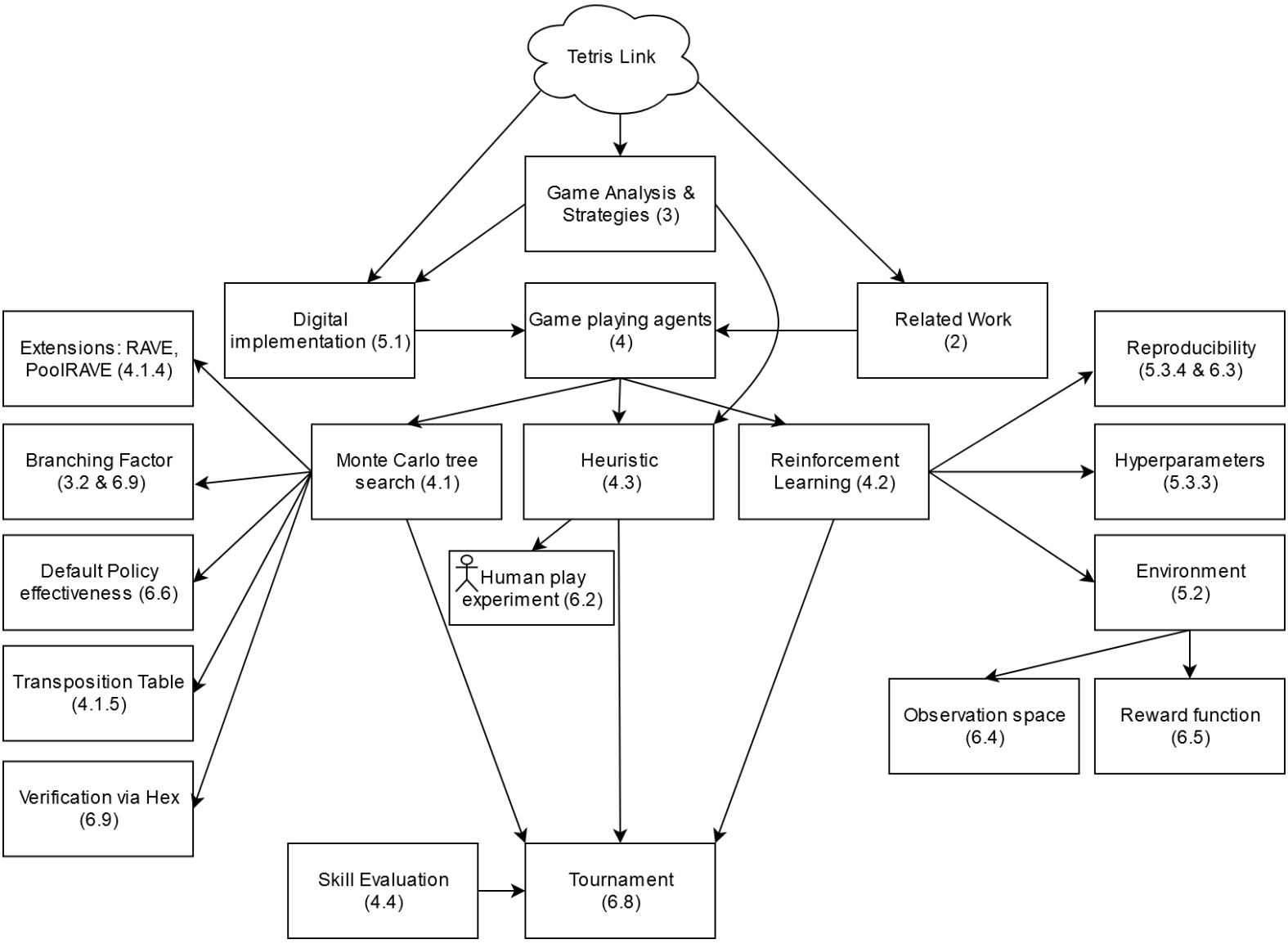


Fig. 1: A diagram visualising the journey towards building AI agents for Tetris Link. The numbers in parentheses reference the corresponding sections.

Undergraduates are supposed to learn"business decisions" by playing. Students have to play Tetris Link and another board game. Afterwards, they are required to write a report about similarities between business decisions, and the decision one has to make within the game. In the reports, they were hoping students would use the following keywords: Dominance, Blocking, Strategy, Market Share, Competition, Diversification. These keywords describe the game quite well because one has to place blocks to achieve maximum coverage (or dominance) and at the same time make sure the opponent has a hard time doing the same by blocking them off and keeping their "market share" at a minimum. The experiment is coined successful as many people reported that Tetris Link helped them understand the theories being taught in the class. Since there currently is no paper analysing the game more in-depth, we do it in section 3.

2.2 Monte Carlo tree search

Monte Carlo tree search (MCTS) is a well known and tested approach for agents in board games. The first successful applications include games such as Go (1993), Bridge (1998) and Scrabble (1998)[10]. In 2007 an MCTS agent gained its first world championship medal in a general game playing competition[10], proving its broad applicability to different games. Games like Tetris Link that focus on connections are another strong suit of MCTS[10]. These include Hex, Havannah, Y and Lines of Action[10]. Games are not the only useful application for MCTS. It can solve transportation problems[88], schedule projects[51] and even figure out NP-complete puzzles[76]. The remarkable thing about MCTS is that it does not require domain knowledge for any of this, although for some tasks, such as playing Super Mario, including some domain knowledge can increase performance[37]. Furthermore, MCTS guided by a neural network, e.g., AlphaGo[81], could be seen as including domain knowledge because the network trains on played matches.

2.3 Reinforcement Learning

Reinforcement learning (RL) is an old psychological concept that was already studied in 1957 to teach rats[83]. Dogs are another great example showing that positive and negative reinforcement can work well to shape behaviour in animals[52,55]. One of the first applications in computer science of RL, as we know it today, was in 1983 to a pole balancing problem[6]. These continuous control tasks work well and often occur in robotics. An example of this is letting two arms juggle a stick which was already learned in 1994[75]. RL is not limited to robotics. A non-exhaustive list of other applications includes profitable stock trading[39], traffic signal control for better traffic flow[5], mastering video games[58] or image segmentation[74]. Even human-like skills such as learning a chain of actions to reach a goal[44], or using existing learned knowledge to more quickly understand a new task (transfer learning)[87] are achievable. This paper focuses on applying it to board games. The most well known RL board game application is to the game of Go by David Silver[82]. It gained a lot of media

attention because it beat a human grandmaster in the game[7]. It is important to note that here RL was only used to guide an MCTS search instead of directly outputting which action to take. In the board game Settlers of Catan RL also needed help by a model tree to reach a decision[68]. In Othello, the RL agent directly determines its action without any guidance[17]. In our experiments (section 6.7), we will also let the RL agent directly decide which action to take instead of being a guide to other algorithms.

3 Tetris Link

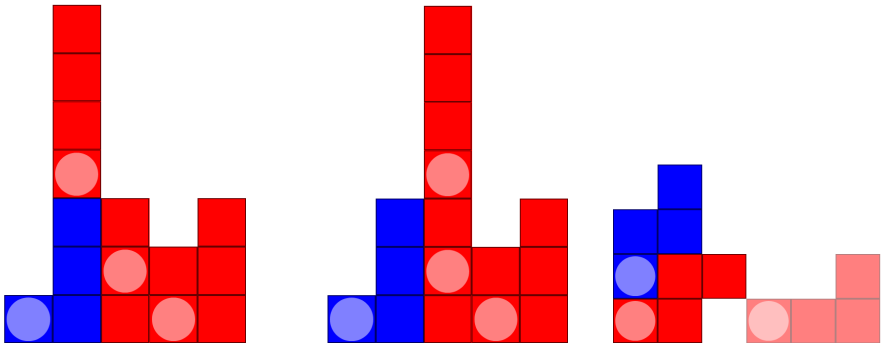
Tetris Link, depicted in Figure 2, is a turn-based board game for two to four players[1].

Fig. 2: A photo of the original Tetris Link board game. The coloured indicators on the side of the board help to keep track of the score.

[1] Information of the game Tetris Link and strategies were deduced by the paper author using the actual board game because as noted in related work (section 2.1) no scientific paper analysed Tetris Link yet.

It is a fully observable perfect information game[2] because one can deduce all relevant information, such as the number of pieces of a specific shape a player has left, from looking at the state of the board. In an imperfect information setting, not all relevant decision making information is available at all times[84].

Like the original Tetris video game, it features a ten by twenty grid in which shapes called tetrominoes[3] are placed on a board. This paper will refer to tetrominoes as blocks for brevity. The five available block shapes are referred to as: I, O, T, S, L[4]. Every shape has a small white dot, also in the original physical board game variant, to make it easier to distinguish individual pieces from each other. As can be seen in Figure 3a, without the dots one would hardly be able to make out the two individual blocks forming the U. Every player is assigned a colour for distinction and gets twenty-five blocks so five of each shape. In every turn, a player must place precisely one block. If no block that fits is available to the player anymore, then he will be skipped. A player can never voluntarily skip if one of his available blocks fits somewhere in the board even if placing it is disadvantageous. The game ends when no available block fits into the board anymore. The goal of the game is to gain the most points.



(a) No points for red because the group only has two blocks.

(b) Three points for red because the three blocks form a chained group.

(c) One minus point for red that could be fixed in Tetris but not Tetris Link.

Fig. 3: Small sections of game boards that visually explain some game rules.

One point is awarded for every block, provided that it is connected to a group of at least three blocks. Not every block has to touch every other block in the group. Figure 3b visualises this. The I block only touches the T but not the L

[2] The original rules define an imperfect information game, but because we play without rolling dice, it becomes a perfect information game.

[3] A shape built from squares that touch each other edge-to-edge is called a polyomino[15]. Because they are made out of precisely four squares, these shapes are called tetromino[95].

[4] The S and L blocks may also be referred to as Z[11] and J[12].

on the far right. Since they together form a chained group of three, it counts as three points. Blocks have to touch each other edge-to-edge, so Figure 3a would mean no points for the red player as the I is only connected edge-to-edge to the blue L.

A player loses one point per empty square created with a maximum of two minus points per turn. Figure 3c shows how one minus point for red would look like. Moreover, it underlines a fundamental difference between regular Tetris and Tetris Link. In Tetris, the blocks slowly fall, and one could nudge the transparent L under the S to fill the hole. Since Tetris Link is a physical board game, one can only throw pieces into the top and let them fall straight to the bottom. The fall cannot be influenced, so the whole can never be filled. In the original rules, a dice is rolled to determine which block is placed. If a player is out of a specific block, he gets skipped. However, since every block is potentially one point, being skipped means missing out on one point. Due to chance, it might also be that a player gets skipped multiple times and loses the ability to win the match because he is behind too much on points. Because of this disadvantage, this paper modified the rules to allow the player to freely choose a block to place so no points can be lost to the skip mechanic. Being skipped due to missing a block without rolling dice is possible, but much less probable.

3.1 Verifying completability

We will start by verifying that every game can come to an end and will not somehow loop forever. The board is ten squares wide and twenty squares high so it can accommodate 200 individual squares. Every player has twenty-five blocks, each consisting of four squares. There are always at least two players playing the game.

$$\begin{aligned} &playerPieces * squaresPerPiece * playerAmount \\ &= 25 * 4 * 2 \\ &= 200 \end{aligned}$$

So two players can place a total of 200 squares. Disregarding block shape the board can always be perfectly filled up. Even with block shape and piece limitations in mind, the game can always end, because at most they could lead to holes that then lead to a full board earlier because it can not accommodate all pieces.

3.2 Branching Factor

An essential part for tree search (see section 4.1) algorithms is the branching factor of a game. The branching factor is the number of possible moves a player can make in one turn[18]. Calculating the number of possible moves means finding out how many different positions and orientations for a block are available. First of all, we look at the number of orientations for each block. The I has

two as it can face either horizontally or vertically. The O block has only one orientation because it is a square, so if one turns or mirrors it, the shape does not change. The T and the S block have four different orientations for every side one can turn it to. The L is an unusual case as it has eight orientations. Four for every side, one can turn it to, but when one mirrors the shape, it has four additional sides to be turned to. Hence, nineteen different shapes can be placed by rotating or mirroring the available five base blocks. Since the board is ten units wide, there are also ten different drop points per shape. So in total, there are 190 possible moves one can take in one turn. The game rules state that all placed squares have to be within the bounds of the game board. Twenty-eight of these moves are always impossible because they would have some squares that are overlapping the bounds either on the left or right side of the board. So the exact branching factor or number of maximum possible moves in one turn for Tetris Link is 162. Since the board gets fuller throughout the game, the branching factor also goes down. To see how the branching factor develops throughout a match, we simulate 10,000 games using random play as well as the *user* and *random heuristic* (section 4.3). How the heuristic agents work and what their prepositions (user / tuned / random) mean will be explained in section 4.3. We graph the average number of moves per turn as can be seen in Figure 4.

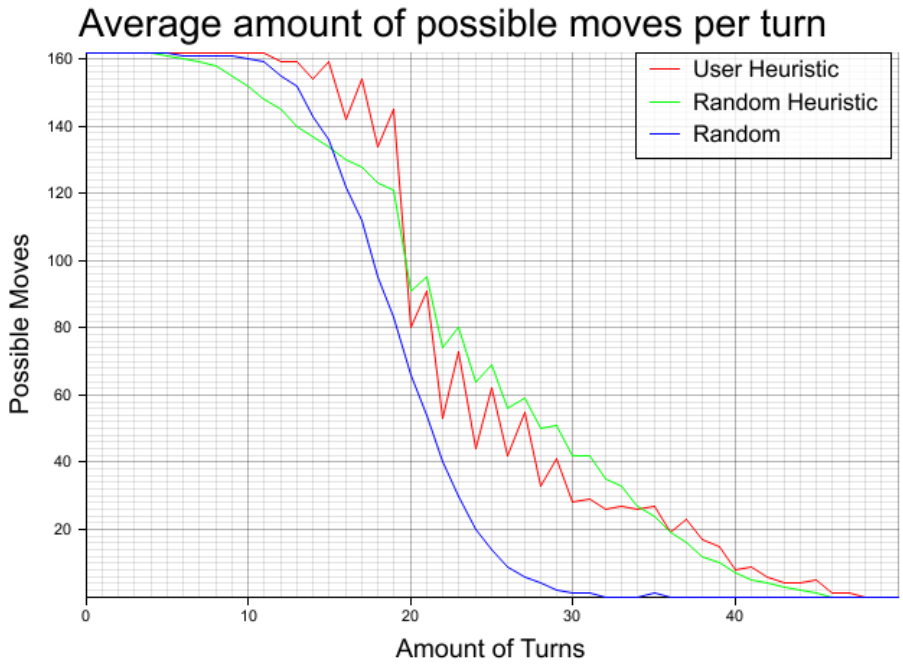


Fig. 4: The average number of possible moves throughout played turns.

Although the average line of random play is less jittery, the margins, visible in Figure 5, show much stronger deviations in random play. For heuristic matches, the deviation is a lot smaller. The first eight to ten turns, all moves are available

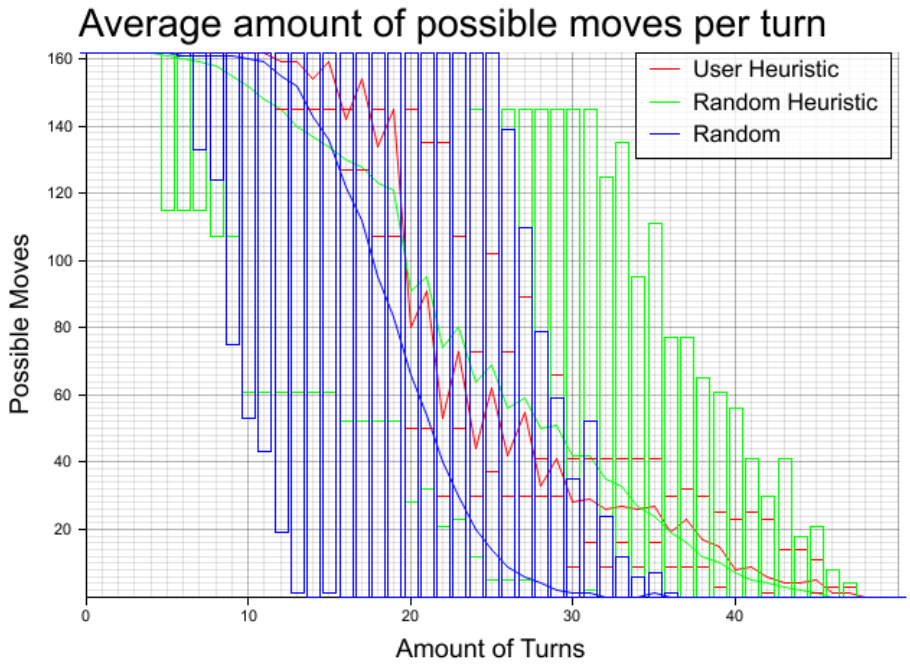


Fig. 5: The average number of possible moves throughout played turns, including the minimum and maximum values as blocks to stress the possible deviation depending on the board situation.

regardless of the quality of play. After that, there is a slow but steady decline. The fact that random moves already end at turn thirty underlines the low quality of random moves. Many holes with many minus points are created, and fewer blocks fit into the board, meaning fewer plus points can be gained and the game ends earlier. The heuristic shows that good gameplay also fills most of the board and hence takes more than forty turns. Furthermore, the amount of possible moves midgame (Turn 13-30) decline a lot slower, and hence offer more variety to the outcomes.

3.3 First move advantage

A controversial topic in turn-based games is whether making the first move gives the player an advantage[93]. To put this into numbers, we let different strategies

play against themselves 10,000 times to look at the win rate. Furthermore, the first six (#1) or all (#2) moves are recorded and checked for uniqueness. How the heuristic agents work and what their prepositions (user / tuned / random) mean will be explained in section 4.3.

	Random	**Random heuristic**	**User heuristic**	**Tuned heuristic**
Win Rate #1	47.84%	47.15%	71.93%	68.41%
Unique Games #1	10,000	2188	7	29
Win Rate #2	48.16%	47%	71.65%	70%
Unique Games #2	10,000	10,000	7	50

Table 1: A table showing whether taking the first turn gives the player an advantage over 10,000 games. Furthermore, the first six (#1) or all turns (#2) are compared for uniqueness to see whether the same games keep repeating.

As can be seen in Table 1, the win rate for *random heuristic* choices is almost 50%. Although the win rate for the first player is a lot higher for the *tuned heuristics*, these numbers are not as representative because the heuristic repeats the same tactics over and over again resulting in only seven or twenty-nine unique game starts. If we replay the same game over and over again, then we will not know whether the first player has an advantage. Especially considering that at least until turn ten all moves are always possible meaning at turn six there are around 10^{13} or 18 Trillion[5] possible outcomes but only seven or twenty-nine of those are ever chosen! Since *random heuristic* has a lot more deviation and plays properly as opposed to complete randomness, we believe it represents the actual chances of the first person winning best. Moreover, around 47% is close to an equal opportunity.

The *user heuristic* merely repeats seven unique matches, meaning its choices are quite limited. For the *tuned heuristic* there are more unique games, meaning more deviation between moves can happen. Still, 50 out of 10,000 unique games is quite limited. #1 and #2 were calculated independently, so in total 20.000 games were played. Also seeing only small deviations in the percentages suggests that these numbers are quite stable. Different match history comparisons of Chess also measure a difference of around two to five percent in win rate for the first player[93]. However, since neither Tetris Link nor Chess have been solved, one cannot know for sure that there is a definite advantage.

[5] $BranchingFactor^{TurnAmount} = 162^6 = 18,075,490,334,784$

3.4 Strategies

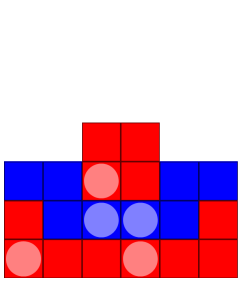
As the keywords from the related work (see section 2.1) already suggest: Tetris Link is a competitive strategy game. With its high branching factor, it is up to the player to make use of his knowledge to break these possibilities down to the most valuable moves. Strategies that can be used to assess a board state and figure out an optimal play are essential for this. We go into some example strategies that may help the player get closer to a win.

Block \ Orientation	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
I	6	9	-	-	-	-	-	-
O	6	-	-	-	-	-	-	-
T	4	6	5	6	-	-	-	-
S	5	6	5	6	-	-	-	-
L	6	7	6	6	6	7	6	6

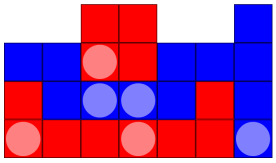
Table 2: This table displays the number of connectable edges after placing a block. The rows represent the block type and the columns the rotation value.

First of all, a player needs to connect their blocks to gain points. However, the game is turn-based, so depending on the number of players, one to three other blocks will be placed, meaning the block that was just placed might be unreachable in the next turn. To maximise the probability that one does not get blocked off until the next turn, one has to maximise the *connectability*[6] of the block. Connectability describes the number of open edges to which another block could connect after placing it. In Table 2, the connectability of each block for each rotation is listed. Most blocks offer a connectability of around six. The vertical I offers by far the highest connectability, but since one only has five of them, one should not immediately use them all up. The I is the only block that prevents taking minus points, all other blocks will leave a lot of empty spaces below them. As previously shown in Figure 3c, edges that face downwards can only be connected to existing blocks or will create minus points. Hence downwards facing edges are not included in the count of the table. Furthermore, if two edges form a corner, they are counted as one because only one square can ever fit and it will automatically be connected to both edges. However, a match of Tetris Link will not be won by solely focusing on connectability and forming groups. Blocking off the opponent and making sure they do not get any points is also a priority. Figure 6a shows an example of a situation where aggressive moves are worthwhile. Red made the mistake of not realising that it could lose two points at the bottom because it can only connect on the right-hand side

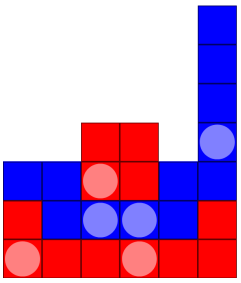
[6] Strategy terms are made up by the papers author



(a) The situation in which blue has to decide to block red or focus on connectability.



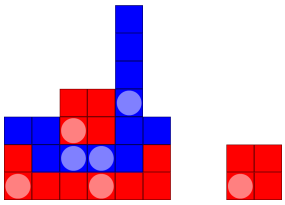
(b) Blocking off the opponent denies them two points but yields less connectability.



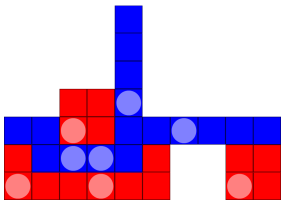
(c) Focusing on connectability allows the opponent to save the two points.

Fig. 6: Small sections of game boards that underline the importance of blocking off the opponent. The leftmost side is the end of the game board. Blocks could only be added to the right or above the existing blocks.

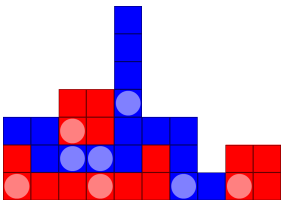
(left-hand side is already the edge of the game board). Blue can now block red off, as portrayed in Figure 6b, which makes red permanently lose two points at the cost of three connectability for blue. Alternatively, blue could play defensive and focus only on connectability, as seen in Figure 6c. Blocking off is the better play, because in a 1v1 the connectability of the I, as well as the group in total, is still high enough in Figure 6b that red cannot wholly block said group off in one turn, hence the connectability loss of three is negligible. The rule that a player can only gain a maximum of two minus points per turn can be exploited in combination with blocking. Figure 7a shows a situation where red made another mistake and is now vulnerable to loose three blocks in one turn. Figure 7b shows how blue could block red off at the cost of minus points. Red loses three Points in total while blue loses one (+1 - 2 = -1) which means a net gain of two points for



(a) The situation in which blue has to decide to block red or focus on connectability.



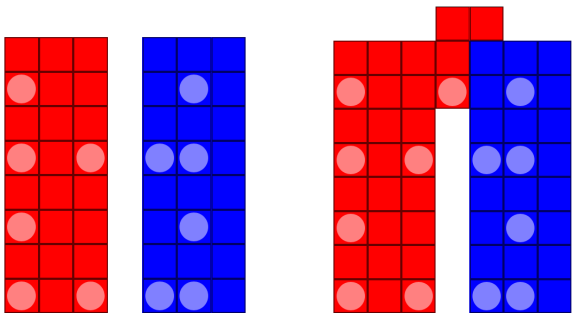
(b) Blue blocks off three possible points for red at the cost of taking two minus points.



(c) Pointwise more optimal play, making red loose two points while blue gains one point.

Fig. 7: Small sections of game boards showing that taking minus points can give the player an advantage. Left and Right of the image are the limits of the board.

blue. One might also block red off only partially, as seen in Figure 7c. Red will lose two points, and blue will gain one point, which is a net gain of three points for blue and hence would be the most valuable play for the situation. Whether Figure 7b or Figure 7c is the better play is disputable, as the red block on the right in Figure 7c leaves red with more possible moves the next turn, and makes it harder to shut them out completely which would force them to start a new block group. A player can also try to force another player to take minus points by exploiting the fact that a player may not skip a turn voluntarily. Figure 8a shows a hypothetical setup where both players filled the board, leaving out only one straight empty line and one height on the top. The only blocks that can fit here are either a horizontal I or a vertical L, which makes this an excellent example of why one should keep at least one I for the late game. We assume that the I's have already been used up to create the whole, so the L has to be placed as seen in Figure 8b. This strategy can help turn around a match right before the end. However, it is a risky strategy because it is difficult to guarantee that it is not your own turn when the whole has to be filled, and nothing else fits anymore. From personal experience of the author, this strategy backfires more often than it helps. Similar wholes also come into existence without consciously creating them during regular gameplay. These wholes also need to be taken into account of the connectability of a block. Although both red and blue have a connectability of eight within the whole, only a vertical I can fit in there. So even though theoretically a connectability of eight is given, the situation reduces this to a practical connectability of two.



(a) A setup forcing players to take minus points.

(b) Only L fits, so red has to take two minus points.

Fig. 8: Small sections of game boards showing that minus points can be forced upon a player. Assuming that there is only one square free at the top and none at the left and right side of the group.

4 Methods

The methods section presents all information related to the inner workings of the playing agents. It also explains how we will evaluate a players skill in section 4.4.

4.1 Monte Carlo tree search

4.1.1 Game tree

The connection between trees and games might not be obvious, but Wikipedia explains it very nicely: "A game tree is a directed graph whose nodes are positions in a game and whose edges are moves"[94]. So every game state represents a node. Every action one can take at the specified node results in an edge to a new node. This new node represents the state after the action was taken. A tree search then searches through a given tree to find the best possible action.

4.1.2 Algorithm

Monte Carlo tree search (MCTS) is an algorithm that specifies the way a game tree is explored. If not otherwise cited, the following information is gathered from Browne's MCTS Survey[10].

MCTS has a generic four-stage approach visualised in Figure 9:

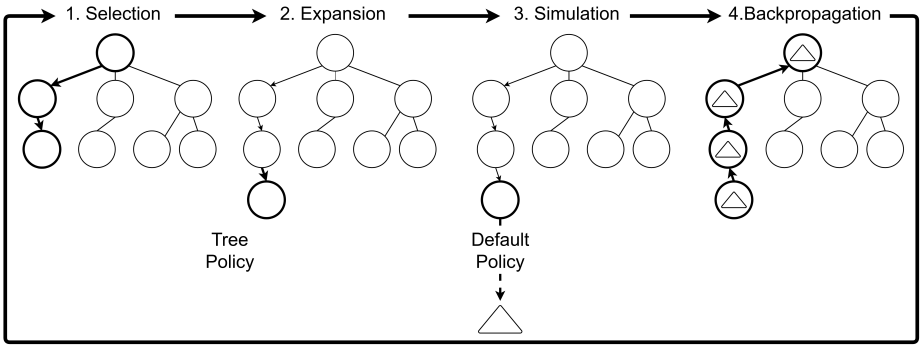


Fig. 9: A single iteration of the generic four-stage MCTS approach. Inspired by [10].

1. Selection: From the current root node children are selected using the tree policy until a node that needs expansion[7] is found. If it is a non-terminal and unvisited node, it can be expanded.

[7] A node counts as expanded if it was visited, and at least one child node was added.

2. Expansion: Child node(s) are added to expand the tree given the possible actions.
3. Simulation: From the newly added node, the game is simulated until the end to know whether the player would win or not. In the simulation, the actions of the player and the opponent are chosen using the so-called default policy.
4. Backpropagation: The result bubbles back up the tree through the selected nodes.

As can be seen in Figure 9 and read in the ordered list, these four steps rely on two policies: The tree policy to decide which node is selected and expanded next as well as the default policy which decides how the simulations are played out. The tree policy needs to find the right balance between discovering new nodes (exploration) and deepening info about existing nodes (exploitation). The default policy needs to find the right balance between speed and realistic game simulation results since its results are used for the tree policy. The speed of the default policy is essential to the quality of the algorithm because MCTS is an anytime algorithm. The MCTS Loop can be run indefinitely and hence be stopped at any point in time. The longer it runs, the more the result converges to the optimal play, but it is important that good actions can be found in a reasonable time. By modifying these policies, different MCTS variations yielding different results can be created. The following subsections introduce the different policies used for the experiment.

4.1.3 UCT / UCB1

The most commonly used MCTS policies are based on a multi-armed bandit algorithm called upper confidence bound for trees (UCT)[43]. The multi-armed bandit problem refers to problems where the player needs to choose one of k actions to maximise the cumulative reward[10]. As tree policy, the upper confidence bounds (UCB1) formula is used:

$$UCB1 = X_j + 2C_p\sqrt{\frac{2\ln n_s}{n_j}}$$

Where X_j is the average reward of leaf j, n_s is the number of the times the parent node was visited, n_j the amount of times the inspected child node was visited and C_p is a constant that is bigger than zero[10].

This formula very clearly shows the balancing between exploration and exploitation. The first component of the formula X_j makes sure that good nodes are exploited further. The second component of the formula $2C_p\sqrt{\frac{2\ln n_s}{n_j}}$ is responsible for the exploration. That is why C_p is called the exploration parameter because it can influence the balance between exploration and exploitation. For the default policy, one should use uniformly random choices.

4.1.4 PoolRAVE

Pool Rapid Action Value Estimation (PoolRAVE) is an extension to the generic MCTS to bias the search by adding the *all moves as first* (AMAF) heuristic to the tree policy[72].

RAVE modifies UCB1 by replacing the exploration constant C_p with the RAVE value[72]:

$$RAVE = x_j + \alpha x_{s,j}^{RAVE} + \sqrt{\frac{2 \ln n_s}{n_j}}$$

As in UCB1 (section 4.1.3), the underscored s refers to parent values and underscored j to the child node under consideration.

The RAVE value $x_{s,j}^{RAVE}$ is the percentage of wins established with RAVE values instead of standard wins and losses.

α is a per-node value that is initially set to one, and after every pass through, it is updated with the following fomula[27]:

$$\frac{\beta - n_{s,j}^{RAVE}}{\beta}$$

Here β is a freely choosable parameter that takes the role of the exploration parameter. If it is high already visited nodes are more likely exploited until they have been visited β times. After that exploration of new nodes takes precedence. $n_{s,j}^{RAVE}$ denotes the total number of games, including simulations, starting from s where the action that leads to j has been chosen.

The RAVE values are calculated by looking at the AMAF wins and losses instead of the actual wins and losses.

The biggest difference is in the node value updating. $x_{s,j}^{RAVE}$ and $n_{s,j}^{RAVE}$ are updated after every simulated playout, not only after a batch of simulations. In turn, the RAVE values rise quicker, and the algorithm gains more confidence in its win rates[27].

Besides tracking different visit and win counts PoolRAVE further modifies the tree selection. At the beginning of the selection step with a fixed probability p, one of the k best moves according to the RAVE value is chosen for further exploitation instead of using the normal tree policy.

4.1.5 Transposition Table

One single state might be reached through different sequences of actions. Such duplication of states is called a *transposition*[53]. The actual state of a Tetris Link game is defined by the location and orientation of blocks and whose turn it is. An example of such a representation using numerical values can be found in section D.1. By hashing theses states and storing them in a table, one can check whether the current state has already been seen[53]. By linking two different paths to one target node a whole subtree, that usually would have to be checked, can be eliminated, which significantly reduces the search space. In Chess, one of

the most popular methods for hashing is called *Zobrist Hashing*[53]. It works by storing a randomly generated number for every possible value that can be found in a specific grid field. So for Tetris Link, there can be two different numbers per player in a field, so each grid field has four random numbers. For every grid that is occupied, the accompanying random number is XOR'ed. Since the random numbers only have to be initialised once and then it is just at most 200 XOR operations this is reasonably fast. The alternative to *Zobrist* is the default hashing algorithm for *Rusts HashMaps* called *SipHash*[42]. Since MCTS has to check as many positions as possible, the transposition table lookup needs to be as fast as possible. Hence a benchmark was conducted, and its results are visible in Table 3.

	SipHash (array)	**SipHash (string)**	**Zobrist (array)**	**Zobrist (direct)**
Milliseconds	3.75	6.84	3.52	3.99

Table 3: A benchmark of different hashing functions for the transposition table.

Zobrist hashing offers two advantages. First of all, storing a single 64-bit number uses far less memory than a 1608 bit[8] array as would be the case with SipHash[9]. Second of all, *Zobrist* is $\approx 0.2ms$ quicker than the fastest *SipHash* variant. When *Zobrist* gets a reference to the game board and iterates over the placed blocks to figure out which random values need to be XOR'ed (direct), it is unexpectedly $\approx 0.38ms$ slower than first creating a full board representation array and hashing that using *Zobrist* (array). The resulting hash is a 64-bit unsigned integer. Hence it can differentiate between $\approx 1.8 * 10^{19}$ [10] board states. After eight turns, there are already $\approx 7.6 * 10^{19}$ [11] board states, so conflicts, where two different states hash to the same value, are bound to happen. Initially, this posed a problem on the results of the MCTS search, but we found a way to mitigate this by introducing a sanity check. If a transposition is found the depth and the amount of possible actions of both nodes is compared. Only if those match is the already known node used. The sanity check does not prevent conflicts, but it reduced the amount of search breaking transpositions in our experiments.

[8] $8bit * 201entries = 1608bit$

[9] A *HashMap* will keep all keys in its original unhashed form, so if an array is passed as key, then a clone of it will be kept in memory.

[10] 18,446,744,073,709,551,615

[11] 76,848,453,272,063,560,000

4.2 Reinforcement Learning

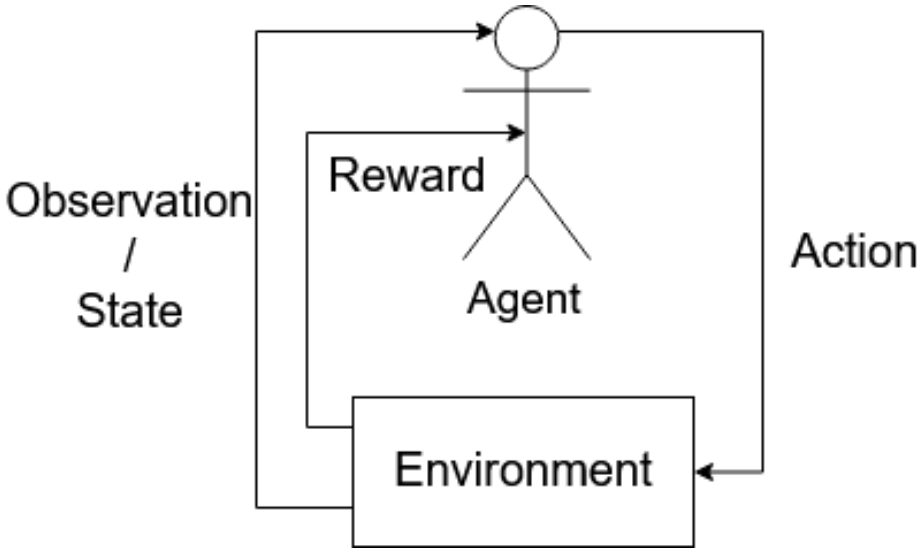


Fig. 10: A diagram visualising the standard reinforcement learning model inspired by [41].

In the standard reinforcement learning model, depicted in Figure 10, an agent is connected to an environment[41]. The environment gives the agent its observation. Based on this observation, the agent can take actions that modify the environment. For each action taken, the agent gets a reward as well as the newly modified state of the environment.

With AlphaGo Zero, Deepmind has shown that reinforcement learning can be effectively applied to guide a Monte Carlo tree search (MCTS) to acquire expert-level play[82]. We show the importance of guidance through a tree with a high branching factor in pure MCTS experiments in section 6.6. Pure reinforcement learning problems, without MCTS, usually focus on continuous control tasks[50] such as the Cartpole environment[9]. In it, a stick needs to be kept from falling over by being able to apply force to it form the left or right.

Unfortunately, Deepmind does not open source their code[12], but they publish papers for scientists to reproduce the results. The case of "minigo"[47] shows the problems with having to re-implement such an experiment. Their final results are comparable but not as good as the original AlphaGo Zero paper[47]. Reproducibility still poses a big problem in the field of neural networks[36]. We will also be experimenting with reproducibility in section 6.3. A direct competitor to Deepmind is OpenAI. OpenAI achieved great success by showing that AI can

[12] They also should not, considering that they are not making any profits yet[78].

play Dota 2 competitively[20]. They were funded by Elon Musk with the goal to make AI accessible to everyone[57]. They try to publish as much as possible, but they too try to make a profit and do not publish everything. Fortunately, the most important RL algorithms (PPO2, TRPO, A2C, DQN,...) have been published by them under the name "baselines". These are reference implementations that are not well documented, but that has been solved in a fork called "stable-baselines"[30]. Furthermore, OpenAI developed the "gym"[9]. Gym leverages the standard reinforcement learning model from Figure 10 by specifying a uniform programming interface for agent environment interaction. The algorithm implementation itself then only interacts with the provided gym without any knowledge of the game behind it. This is beneficial because the implementations of the algorithms are independent of the gym implementation. Hence we can use existing implementations of RL algorithms that have been battle-tested on various existing problems[13]. This enables us to focus on providing a suitable environment from which agents can learn well (see section 5.2). Another benefit of the gym architecture is that it enables transfer learning experiments to be carried out easily[66][14]. Transfer learning can boost learning of a new task and also allows a single model to master multiple tasks simultaneously[67].

4.3 Heuristics

A heuristic in the field of AI can be seen as any device, that one believes to provide a useful, but not necessarily correct or perfect, solution to a problem[73]. In this instance, the device is a function whose input is a game board state, and the output is the decision which blocks to place where. With the analysis of strategies (see section 3.4), we already have knowledge that can be represented numerically about how good a specific move could be. Namely: connectability, group size, blocking and points. Connectability counts the number of accessible edges to which further blocks could be connected. Group size, as the name suggests, is the total amount of blocks that are connected in one way or the other. So for example, if two groups with two blocks each exist, then this value would be four. For blocking we count the number of edges of a player that touch the edge of an opponent. If a players square touches an opponent square edge to edge the opponent can never connect his own square to said edge anymore and hence loses one connectability. Furthermore, the players' score is included in order to know whether increasing the group size was beneficial for the points as well.

Each of these four values is assigned a weight to be multiplied with. The four multiplication results are then summed up. For every possible action in a given turn, the heuristic value is calculated, and the one with the highest value is chosen. If multiple moves have the same maximum value, a random one of these best moves is chosen. The initial weights were manually set by letting the

[13] Thirty-nine papers cite the stable-baselines according to Google Scholar[24].

[14] The different gym environments will need the same action and observation space for this to work!

heuristic play against itself and seeing which combination would result in the most points gained for both players. We refer to this as *user heuristic*. We then use Optuna, a hyperparameter tuner (section 5.3.3), to tune a set of weights that reliably beat the *user heuristic* and call it *tuned heuristic*. For playstyle variety (see Table 1) we also test a so-called *random heuristic* which at every turn generates four new weights between zero to fifteen. In section 6.2, an experiment with real players against the heuristic is conducted.

4.4 Skill Evaluation

In order to evaluate the different playing agents, we need a stable numeric rating to judge their skill level. The most well-known rating that achieves this is called *ELO*, invented for the game of Chess by Arpad Elo[22]. However, the original algorithm specifies a singular numeric value that does not allow for uncertainty. This was addressed in 1929 by using a bayesian approach that allows for uncertainty[97]. Furthermore, rating two players whose ratings are far apart might distort the rating of the better player in the original algorithm, but the bayesian approach fixes this[97]. This is relevant because we expect random play in our Tournament (section 6.8) to perform badly and do not want the other ratings to be distorted by that.

Since Tetris Link could be played by more than just two players we want a rating system that supports multiplayer free-for-all ratings. We found an algorithm based on the Bradley-Terry model[22], which uses a bayesian approximation method and supports multiplayer free-for-all games[92]. We will refer to this algorithm as Bayesian Bradley Terry (BBT). A player rating in BBT consists of two floating-point numbers: Sigma (σ) and Mu(μ). Sigma is the absolute rating similar to ELO. Mu represents the skill uncertainty that prevents bad updates if peoples skills are too far apart. The final numerical value that will be found in the evaluation (section 6.1) is then calculated by:

$$\sigma - 3 * \mu$$

In the original paper, they use a skill rating from 0 to 50, similar to TrueSkill[29]. By changing the β parameter of the rating function, we can make said rating range from around 0 to 3000, which is the standard range in ELO ratings[22]. Furthermore, to have an estimate of how strong the algorithms would be against a human, we run experiments with actual human players in section 6.2.

5 Design

5.1 Implementation

For the experiments (section 6), a digital implementation of the board game is required. A client-server web app of the game using JavaScript (JS) and NodeJS is implemented for visual analysis of matches as well as gathering human play data (section 6.2 and Figure 12). However, MCTS and RL require an implementation that focuses on performance so that the search/training process is not slowed down by the game simulation. So a second implementation in Rust is done. Both implementations share a common game log format using JSON to enable interoperability. An example of what said log looks like for Figure 7b can be found in section D.3. The MCTS search is implemented in Rust and can directly interact with the game. The existing RL code (gym, stable-baselines) is written in Python. To connect Python to our Rust implementation, we compile a native shared library file and interact with it using Pythons *ctypes*. Besides preventing a third game implementation in Python, using native shared libraries is known to improve performance for computationally expensive tasks[40]. This known performance improvement is used by libraries such as numpy[90] or TensorFlow[1].

5.1.1 Performance

In order to assess and optimise the simulation performance, a benchmark is conducted using a desktop machine (section A.1). One full game where the first possible action is always chosen is played and the final score calculated. The initial implementation in JavaScript (JS) requires 82ms on average for this. A simple reimplementation of the JS code in Rust already speeds this up to 12ms. Through a list of optimisations (section A.3) one such simulation in Rust now takes 590μs on average. These changes focus mostly on preventing heap allocations by reusing and resetting existing objects and arrays or incrementally updating game-related information instead of calling a function that gathers the info from scratch into a newly allocated object. So this speedup by a factor of 20^{15} is solely achieved by reducing RAM allocations and caching results.

For MCTS, performance is also essential as one of its parameters is thought time and considering more nodes means getting closer to the optimal play (see section 4.1). This is why we chose tree parallelisation[10], so many threads can expand the same game tree simultaneously. In a single-threaded, test around 3532 nodes are visited per second on average using the optimised game simulation. Due to locking mechanisms, this does not linearly scale up. Using 12 threads around 16258 nodes are visited per second on average. Unrelated experiments with MCTS in Go claim a significant speedup when ignoring locks even though some computed info is lost to data races[19]. We also tried a lock-free variant but did not measure a noticeable difference in visited nodes and hence

[15] $\frac{12ms}{0.59ms} \approx 20.339$

will be using the locking variant in our experiments. We also experimented with a heuristic default policy to guide the MCTS. However, the heuristic calculation is so slow that it only manages to visit ten nodes per second. The heuristic takes that long because, for all possible moves, it has to clone the game board, simulate the action and then calculate the heuristic value. We know from the base implementation optimisations that RAM allocations are slow, and the game object is cloned up to 162 times per decision. Not even multi-threading the heuristic process can make this fast enough for MCTS[16].

5.2 Reinforcement Learning Environment / Gym

Implementing an RL environment poses different challenges to the developer. In OpenAI gym the observation (5.2.1), as well as the actions (5.2.2), are modelled through so-called *spaces*. A space can be a singular (un)bounded number or an array/tuple of (un)bounded numbers. Furthermore, the reward (5.2.3) function is essential for the learning process of an RL algorithm. The environment design and training process allow for many mistakes that are documented in section 5.3.4.

5.2.1 Observation space

There are two options for the observation space. Either pass a simple numerical board representation (see section D.2) into a multi-layer perceptron (MLP) or pass an image representing the current board state (see Figure 11) into a convolutional neural network (CNN). Looking at AlphaGo[81] and Othello[17], we see that they try to give as much information as possible in the observation. AlphaGo even encodes historical information to prevent illegal repetition moves as well as information about who received the "komi"[17].

Hence we try to encode every available information of the game state in an observation. Besides the position of blocks that includes: the current score of each player, the number of available pieces per shape as well as which moves are currently possible. For the numerical representation, we add the additional information below the board representation, which is why we restrict the info to be ten units wide.

The image is just a conversion of the numerical representation, so it has the same info in the same widths. To fit the info into the originally only 32x32 image we put the additional info on the right of the board instead of below it. This 32x32 representation is later scaled up to 64x64 to work out of the box with the *stable-baselines CNNPolicy*.

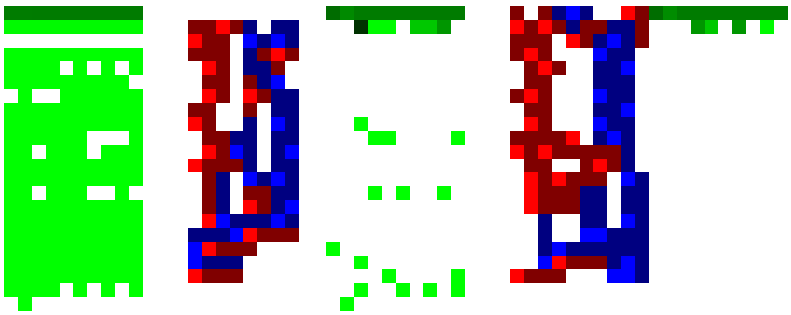
The information encoding is built so that in theory, it supports up to four players. The first four pixels in the first line represent the points of each player. The more points, the more intense the green becomes. Below that are two lines

[16] The ten nodes per second with the heuristic default policy were measured using the already multi-threaded heuristic.

[17] *Komi* refers to the first turn advantage points[81].

that represent the number of pieces per player. The first line represents the pieces for player one and two and the second line for player two and four (in the figure the second line is blank because there are only two players playing). Full green means all pieces are available, as it fades fewer pieces are available. Below that are nineteen lines in which each pixel belongs to one of the 190 available actions. If the pixel is green, the action at the index is possible, if it is not, then that action is not possible.

To make this abstract description more understandable, we will look at different board states and their observation. At the start of a game (11a), almost everything regarding the possible moves is green[18], the available pieces are fully green, and the score is neutral green. In the middle of a game (11b) the scores for player one and two are different, both have some pieces that are not available anymore, and only a few select actions are still possible. At the end of a game (11c), the score is different in shade. Almost all players pieces have been used, and there is no possible action anymore.



(a) A board before any moves were made.

(b) A board in the middle of an ongoing game.

(c) A board after the game is finished.

Fig. 11: The environments observation space represented as a 64x64 image. Unused pixels are black, but for readability, have been set to white in the above pictures.

5.2.2 Action space

For the action space, we tried using a singular discrete number between 0 and 162 (the branching factor). However, the RL agent would not understand which actions are valid in a given turn and countering with a negative reward while not progressing the game resulted in a known *Not a Number* (NAN) bug explained in section 5.3.5. We tried drawing inspiration for a solution to this from the

[18] As explained in section 3.2, there are 190 available actions, but twenty-eight of them are never possible, which are represented here as the white holes in between.

official list of available third party gyms[63]. As mentioned before, RL usually focuses on continuous problems[50], such as the Cartpole environment[9]. The third-party gym list is no exception to this. Only one of the listed environments is a board game, namely GymGo implementing Go. Looking into the source code, we quickly found that they face the same problem. The provided gym would not allow an RL algorithm to run because the observation space was not specified at all, and the action space was defined as a number where it would need to be a gym.spaces object. We offered the project an already merged pull request[19] to address these issues, but that only makes it runnable. It still faces the problem that invalid actions keep being suggested by the RL algorithm.

Back to our Tetris Link gym, we changed the action space to be an array of 190 numbers between 0 and 1. The highest number that references a valid action is then chosen. This results in ignoring all invalid actions regardless of their value. Filtering out invalid turns is common practice for RL boardgame problems (AlphaGo[81], Othello[17]). The only problem with this is that the agent will not know if we had to filter his answer, which we try to address using the *reward function* in the next section.

5.2.3 Reward function

Giving the right reward for an action is crucial in allowing an agent to understand its goal during the learning process[25]. Because of the reward functions importance, we try multiple solutions and run an experiment (section 6.5) to find out which one works best.

Our first approach *Guided* tries to guide the learning as much as possible. We sum up the player's score and group sizes and divide them by 100 so that its a number between zero and one. We include the group size because a players score can only increase after at least three good turns. For more guidance, we add 0.05 if the chosen action was valid or subtract 0.05 if the highest value referred to an invalid action. Furthermore, we want to help the agent understand the relevance of its suggested actions. So we take the value of the chosen valid action and subtract 0.0004 for every other value in the array that is not at least 0.1 smaller than the chosen action. These two scolding mechanisms are supposed to teach the agent only to suggest valid actions and to focus on suggesting one singular high value. Modifying the reward in such a way is called *shaping* and is supposed to help an agent learn how to reach the goal[25]. Existing experiments with shaping show mixed results[25], which is another reason to compare different reward types for our environment. If the agent won at the end of a game, it receives an additional ten points and if it looses we subtract ten points.

The second approach *Score* tries to do as little guidance as possible, so the agent figures out the rest for himself. We only give the current score of the player as reward (also divided by 100). Just like in the first approach winning adds and loosing subtracts ten points. Only returning the score without guidance as

[19] https://github.com/aigagror/GymGo/pull/1

a reward has been successfully used to teach RL Atari games[58]. For both *Score* and *Guided*, only the delta between the previous and current reward is used.

Our last approach focuses on simplicity by giving out zero as reward except at the end of the game, where it is either one for winning or minus one for loosing. This kind of reward is also used in the DQN application to Othello[17].

Summing it up, we have the following reward types:

1. Guided: $\frac{score+groupSize}{100} - scolding$
2. Score: $\frac{score}{100}$
3. Simple: ± 1 at the end of the game depending on win / loss

5.3 Reinforcement Agents & Training

In this section, we present the used RL algorithms (section 5.3.1) and how we use them for our agents (section 5.3.2) in the tournament. We also go into detail on how we optimise the hyperparameters for the training as well as problems that we ran into in the training process (section 5.3.4 & 5.3.5).

5.3.1 Algorithms

stable-baselines offers implementations for many existing RL algorithms. Whether an algorithm can be used depends on its support of the chosen action and observation space[30]. Given that both our observation and action space use the *Box*-space, we are able to use: A2C, TRPO, PPO2[20]. Initially, we planned on comparing all of the usable RL algorithms, but due to numerous problems with the training (section 5.3.4 & 5.3.5) and the amount of time, a training requires we had to settle on one for the final training. We chose PPO2 because it combines ideas from both TRPO and A2C in order to improve upon them[77]. This improvement helps PPO2 to outperform TRPO as well as A2C in six out of seven existing *gyms* dealing with continuous control tasks[77].

5.3.2 RL Agents

We define an RL agent as the combination of environment and algorithm choice. As stated in section 5.3.1, we will only be using PPO2 as RL algorithm. To make the best environment choices, we first did experiments on the observation space (section 6.4) and reward function (section 6.5) in order to find out with which one the agent learns best. Based on the results of these experiments, we choose the numerical observation input (section 5.2.1) and the guided reward function (section 5.2.3) and call the agent *RL-Selfplay*.

Next, we introduce the *RL-Selfplay-Heuristic* agent. We take a trained *RL-Selfplay* agent and continue training it by playing against the heuristic. Observation and reward are the same as for *RL-Selfplay*.

[20] ACER, ACKTR, DDPG and SAC should have also been usable with the observation / action space, however, due to bugs listed in section B.5 we could not.

Because the heuristic plays well even with random weights, we also introduce an agent called *RL-Heuristic*. This agent gets the numerical observation as input and outputs four numbers that represent the heuristics weights (section 4.3). We use a modified version of the guided reward function:

$$\frac{(ownScore - opponentScore) + groupSize}{100}$$

The difference in points between itself and the opponent is used, so it learns to gain more points than the opponent, which in turn encourages tactics such as blocking (section 3.4). Scolding is not necessary anymore as we do not have to filter the output in any way.

5.3.3 Hyperparameters

Every reinforcement learning algorithm comes with its own set of *hyperparameters*, which can make or break the training process of an agent[33].

Fortunately, there is more software that helps tackle the problem: Optuna[3]. Optuna is an, at the time of writing, new[21] hyperparameter optimisation framework. It stands out by currently being the only hyperparameter optimisation framework whose API is define-by-run and not define-and-run. Define-and-run networks do not let the user alter the intermediate variables after their definition in contrast to define-by-run[3]. Furthermore, Optuna uses a bayesian optimisation algorithm. Bayesian optimisation has already shown success in AlphaGo[13]. Hence we believe it should help in this application as well. Moreover, Optunas API is intuitive: Define a variable name, a range and a generation function (random number, [(log) or (discrete)] uniform distribution, or category). The newly generated parameters can then be passed to the reinforcement learning algorithm. Ranges and generation functions are chosen by us and are determined by studying the effect and the default value of each hyperparameter. The exact search settings are listed in section B.1.

5.3.4 Failed attempts

Because Optunas tuned hyperparameters did not perform as expected, we looked into alternatives. DeepMind showed that population-based hyperparameter tuning could yield better results than bayesian approaches[38], such as the one Optuna uses. There is another Python framework called tune[49] that provides a readily usable implementation of population-based search. Unfortunately, the best-reached reward is around -0.11 before the training breaks by only suggesting NAN as actions. Running into this is a known problem, as explained in the next section 5.3.5.

Also, we were not able to reliably reproduce the achieved reward that Optuna found when trying the hyperparameters out in regular training which is why we look into reproducibility in our evaluation (section 6.3).

[21] It was published on the 4th of August 2019 shortly after hyperparameter tuning was evaluated for this work.

Initially, we only trained against the *random heuristic*. However, the agent never seemed to learn much or quickly deteriorated after a bit of improvement. In an experiment with the board game Othello, it has been shown that temporal difference algorithms perform better when playing against themselves while pure Q-Learning works well against a fixed opponent[89]. That is why the final RL training focuses on learning from self-play (section 6.7) instead of a fixed opponent.

5.3.5 NAN Problems

A trained agent can become unusable and only produce NAN values as output. This is a known problem[31] and can be caused by many things, but in particular bad hyperparameters can provoke it more easily. We only ran into this issue with our initially discrete action space (section 5.2.2) and when using tune's hyperparameter search (section 5.3.4) but never during more than ten million steps of training (section 6.7) or Optunas hyperparameter search (section 6.4).

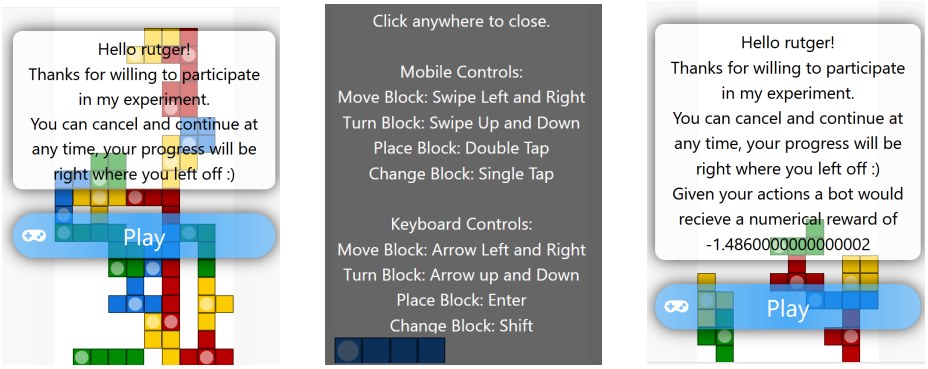
6 Evaluation

6.1 Measurements

All measurements are taken using the desktop machine described in section A.1. Only RL hyperparameter search (section 6.5 and 6.4) and RL training (section 6.7) were conducted on a server provided by the University of Leiden that is detailed in section A.2.

6.2 Human Play

6.2.1 Setup



(a) The intro screen shown after opening the customised link.

(b) The info screen explaining the controls of the game.

(c) The after game screen informing the user of his reward.

Fig. 12: Screenshots of the modified browser game implementation provided to participants of the human play experiment.

To gain an estimate on the strength of the heuristic we collect data with real human players. We modified the web version of the game (section 5.1) to give players the ability to face the heuristic in 1v1 battles, as shown in Figure 12. For every game, we collect player name, bot type, full game log and the reward an RL agent would have received for the *Guided* reward (section 5.2). Using the full game log, we can extract the score/winner and visually inspect the board state at every turn. We invited thirteen people to participate in the experiment: four experienced[22], six casual[23] and three beginner[24]. Participants are only familiar with the actual board game and have never used or played the

[22] These players have played countless matches over many years.
[23] These players have played more than a dozen matches.
[24] These players have played at least one round of the game.

digital version provided here. Every participant receives a unique link that logs them in (Figure 12a) to ensure that we know who played which match. The controls are shown to the player (Figure 12b), and then they can begin playing. At the end of a match, players can see their achieved reward[25] and are invited to play another round (Figure 12c). The opponent alternates between the *user heuristic* and the *random heuristic*. Due to the small participant size, this is a qualitative and not a quantitative study[56]. The results of this experiment are presented in the next section 6.2.2.

6.2.2 Results

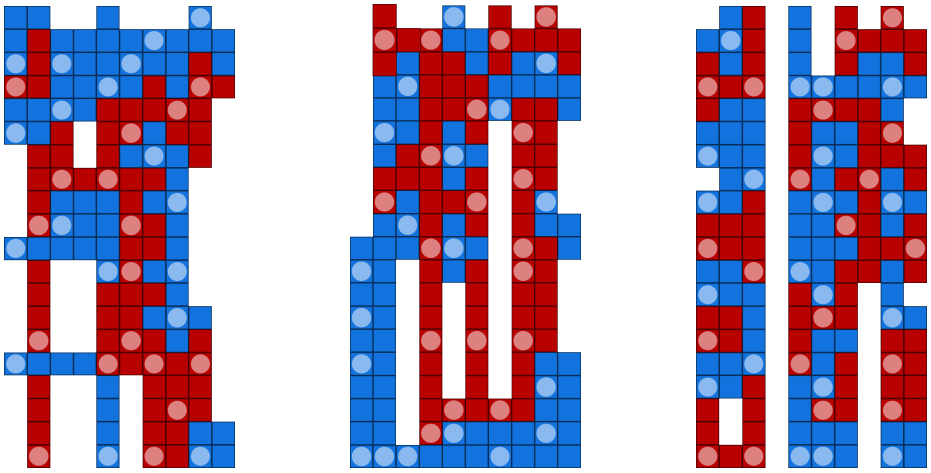
Out of the thirteen invitees six have not played any game (two experienced, three casual, one beginner) which leaves the survey with n=7 (two experienced, three casual, two beginner). In total, twenty-five matches were played, but only

	All	**Experienced**	**Casual**	**Beginner**	**Random heuristic**	**User heuristic**
Games (Won / Total)	14 / 19	5 / 6	7 / 10	2 / 3	3 / 13	2 / 6
Win rate	73.68%	83.33%	70%	66.66%	23.07%	33.33%
Draws	1	0	1	0	1	0
Won by points	2.92	4.2	2	1.5	6.3	9
Average points	14.53	16.17	13.90	13.34	15.77	11.17

Table 4: The results of participants versus heuristics. Won by points refers to the point advantage a player had at the end of a game and is displayed as average over all matches.

nineteen were completed. As can be seen in Table 4, participants won 73.68% of those nineteen matches. A steady rise in average point advantage, win rate and average achieved points from beginner (1.5, 66.66%, 13.34) to casual (2, 70%, 13.90) to experienced (4.2, 83.33%, 16.17) confirms the skill estimate of the participants. The difference in win rate between the *user* and *random heuristic* suggests that due to the randomness of the weights, bad moves become more probable. Interesting to see is that when the heuristic won its point advantage was much higher than when players won, even though players won more matches but then by fewer points.

[25] *Guided* Reward from section 5.2.3.



(a) *User heuristic* (red, 8 points) vs Casual (blue, 5 points).

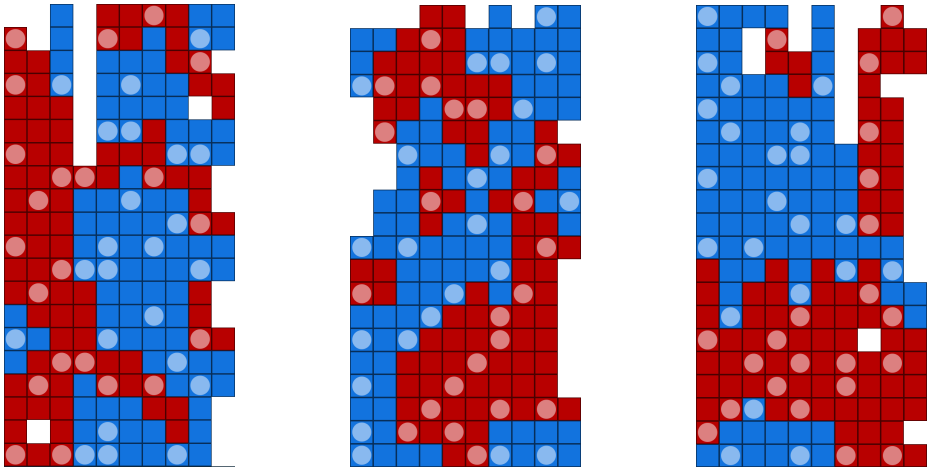
(b) *User heuristic* (red, 17 points) vs Casual (blue, 2 points).

(c) *User heuristic* (red, 5 points) vs Casual (blue, 11 points).

Fig. 13: Screenshots of the final game boards from participants playing against the *user heuristic*.

Thanks to the web interface (section 5.1), we can also look at the boards of the played matches and how they developed over the turns. Through that, we found out that the only two matches that the *user heuristic* won, were matches where players tried to exploit its predictability. The *user heuristics* connectability weight is high, so it always places all vertical I's it has in the first five turns (section 3.4). In Figure 13a, the player tries to prevent the heuristic from gaining height by taking minus points themselves with horizontal I's. This try was unsuccessful, and the points lost to blocking made the player unable to catch up again. In another match, shown in Figure 13b, a player first builds a big base at the bottom until the heuristic used up all of its vertical I's and then blocks the left and right access to the vertical I's. The player succeeded in not letting the bot connect to its vertical I's, but again the player was unable to regain the points lost to blocking. Figure 13c shows a third match that the heuristic lost. The player successfully exploited the lack of vertical I's in the late game. They force a one width whole in the playfield effectively rendering one side of the I's useless, and the player successfully keeps the heuristic from gaining points by blocking off access to two block groups. The *random heuristic* games also allow for strategy interpretations. In all three matches displayed in Figure 14, one can see that bad weights can result in a block being placed in the middle of nowhere. This block can never be connected and is effectively a lost point. This mistake cost the bot two points in the match that was a draw (Figure 14b), so without them, the bot could have won. The randomness can also lead to taking

minus points at times where it could have been prevented. An example of that is shown in Figure 14a. The block at the far right, making a lot of minus points could have been rotated so that no minus points are taken. Compared to the *user heuristic*, the *random heuristic* seems to play more defensive, resulting in more dense and connected structures. In the match shown in Figure 14c, one can see that the end game of the bot was weak. Instead of building upwards, it should have started to slowly spread to the left to also gain more width instead of height. This match also shows wits from the human player. To prevent the bot from connecting its block on the left with its blocks on the right, the player puts a vertical I between them. Besides not being able to connect its blocks anymore, the game is also instantly over. If the last vertical I would have been placed one more square to the right the game could have continued for two more turns and the bot would have been able to connect, however, the outcome of the match would stay unchanged.



(a) *Random heuristic* (red, 19 points) vs Experienced (blue, 13 points).

(b) *Random heuristic* (red, 16 points) vs Casual (blue, 16 points).

(c) *Random heuristic* (red, 16 points) vs Casual (blue, 20 points).

Fig. 14: Screenshots of the final game boards from participants playing against the *random heuristic*.

6.2.3 Bugs

In an initial test with only one participant, there was a bug where under certain circumstances the AI would not make its turn, but the participant could then control the opponent, so effectively the participant played against themselves.

Fortunately, this was discovered before sending links out to all participants. The bug was fixed, and the matches affected by it removed from the records. Furthermore, there was a bug in the selection of the opponent, which is the reason that there are thirteen matches against the *random heuristic* and only six matches against the *user heuristic*.

6.3 Reproducibility

6.3.1 Setup

Reproducibility is a big problem in the field of neural networks[36]. Besides the hyperparameter settings, a lot of weights relevant for training are initialised using pseudo-random number generators[96]. These pseudo-random numbers are a deterministic row of numbers that are calculated using a specific seed[54], so changing the seed results in a different row of numbers. If not set via an API-call a random seed is chosen. Hence not training on a set seed will lead to different results at every run. So, in theory, using a set seed and the same hyperparameters should yield the same result.

It is common to parallelise the mathematical calculations required for the training process using multi-threading or GPUs[62]. We are using stable-baselines which use TensorFlow for the neural networks. It is a known problem that GPU calculations in TensorFlow are less precise than CPU calculations[79], and the problem is actively being worked on[61]. When we started the RL experiments[26] stable-baselines was at version 2.6.0. In that version, only the *learn* function receives a seed. However, we know that already in the network initialisation random numbers are used. Furthermore, tuned hyperparameters did not yield the same result when training on them, so we became suspicious and checked the stable-baselines documentation. In the currently not yet released[27] version 2.9.0, which is already represented in the documentation, they say that GPUs never yield reproducible results and for full reproducibility multi-threading needs to be disabled for TensorFlow[32]. As 2.9.0 is not yet released, we slowly patch in different changes to the reproducibility. First, we changed the number of used CPUs to one in the PPO2 class file and used the *set_global_seed* function already available in 2.6.0 before instantiating the model. These results still were not fully reproducible either. Upon further investigation, we saw that a new function called *set_random_seed* was introduced to the *ActorCriticRLModel* class in 2.9.0, which is the base class of PPO2. Now the seed gets passed to the constructor of an RL algorithm, before initialisation of the model as we tried earlier. The reason why our experiment with *set_global_seed* did not work is that the newly introduced *set_random_seed* calls the seed function on the environment, action and observation space, which *set_global_seed* did not do. With these code changes in place[28], we were able to get the same results with a set seed.

[26] At the start of July.
[27] Writing and experimenting at 20.11.2019.
[28] Full patch detailed in section C.1.

We train using a set seed 102 steps for ten times and see if results are reproducible. Only with the final code patches, and GPU disabled were the results reproducible. With this seemingly reproducible code, we train using all three different reward options until a local optimum is reached (or 10,000 steps have been done). We define a local optimum as the same match repeating three times in a row. Each seed is trained five times with, and we measure the number of steps it took, the average reward achieved, and what the average score of the players was. We repeat this process for 52 different randomly selected seeds.

6.3.2 Results

Even with all these changes, the results are still not reliably reproducible, as can be seen in Table 5. Only 20 out of 52 tried seeds came to the same result more than once. Interestingly enough, these twenty successfully reproducible seeds are the same across all three reward functions, implying that some seeds are less reproducible than others. It is baffling that two other seeds were only reproducible with the least effective *simple* reward. We want to stress that the reproducibility patches use code from an unreleased version of *stable-baselines* that tries to address this issue. So our experiments suggest that more changes are necessary and probably will follow before actual release to ensure consistent, reproducible results. The large standard deviation for the number of required steps underlines the impact the seed has on training success.

Reward Type	All Equal	Amount of Steps	Episode Reward	Score
Guided	20 of 52	326.01	0.22	2.21
Simple	22 of 52	0.0	0.0	4.44
Score	20 of 52	860.61	0.03	2.92

Table 5: Results of self-play with different reward types until either a local optimum or 10,000 steps have been reached. Step, Reward and Score show the standard deviation between seeds.

At the end of January 2020, OpenAI announced that they choose *PyTorch* as a standard framework rather than *TensorFlow* or others[64]. Besides claiming increased productivity, no detailed comparison of the available options and reasons to why *PyTorch* was chosen is given. Reproducibility might be one factor that played into that decision. In a *TensorFlow* Github issue, a user claims that in *PyTorch*, GPU calculations are already deterministic[80]. Another user suggests replacing *TensorFlow* with *PyTorch*, in an *AI Stack Exchange* issue about problems with reproducibility in *TensorFlow*[14]. Regardless of the used implementations, researchers have already identified all possible sources of non-determinism in RL and urge framework developers to eliminate every non-determinism in these sources[60].

Our reproducibility experiment focuses only on our application, and the issues with *TensorFlow* provided that data and parameters are given. However, there are also other issues in reproducing papers published in scientific journals or conference. Because of this, the International Conference on Learning Representations (ICLR) started a reproducibility challenge in 2015 that is still repeated every year[71]. They invite researchers to choose an AI paper, read it and try to reproduce its results. Pineau summarised the results in a talk at the ICLR 2018[70]. Only 32.7% were able to reproduce most of the work in a paper. 54.5% were only able to reproduce some of the work. 9.1% say that they were not able to reproduce most of the work, which leaves 3.7% that were not reproducible at all. And we have not even touched the problem that some RL papers are so computation-intensive that it is near impossible for regular researchers to reproduce them. For example, in the AlphaGo Zero paper, they claim that training took 40 days on their TPUs[82]. If one were to rent as many TPUs as used in their experiment for that period of time, it would cost approximately 35 Million dollars[34].

6.4 Observation Space

6.4.1 Setup

In this experiment, we want to find out if training success is dependent on the way information is encoded in the observation. For that, the same information is encoded into a numerical array and an RGB image (see section 5.2.1). For comparison, we run the hyperparameter search of the *RL-Heuristic* with both kinds of input. All available hyperparameters of PPO2 are being evaluated at the same time. Section B.1 lists the exact hyperparameter name and the ranges that are tried out. We train for 100,000 steps using the suggested hyperparameters. After the training, we let it play 500 episodes and take the average reward achieved as a measure of goodness of the parameters.

6.4.2 Results

It seems that there is no difference in training effectiveness between the two, as can be seen in Figure 15. Both arrive at a maximum reward of around 0.07 after forty tries. The results of unpredictability between seeds and settings from the previous section 6.3.2 also reflects well in this graph. Both reach a reward of around 0.65 at the first step, which is only 0.05 less than the maximum found, and the development of the reward is not steady at all but jitters heavily. Ideally, a hyperparameter search would slowly converge towards a maximum[3]. One reason for the jitteriness could be that all available hyperparameters are being evaluated at the same time with rather large ranges for each individual parameter. That makes the space of viable hyperparameters smaller while at the same time enlarging the space of hyperparameters that probably do not work well. There are other things to consider when choosing the observation space, such as memory. The RGB image uses 768kb[29] per observation, while the

[29] Each pixel consists of three floats. A float uses 64bit. $\frac{64*64*3*64}{1024} = 786,432$

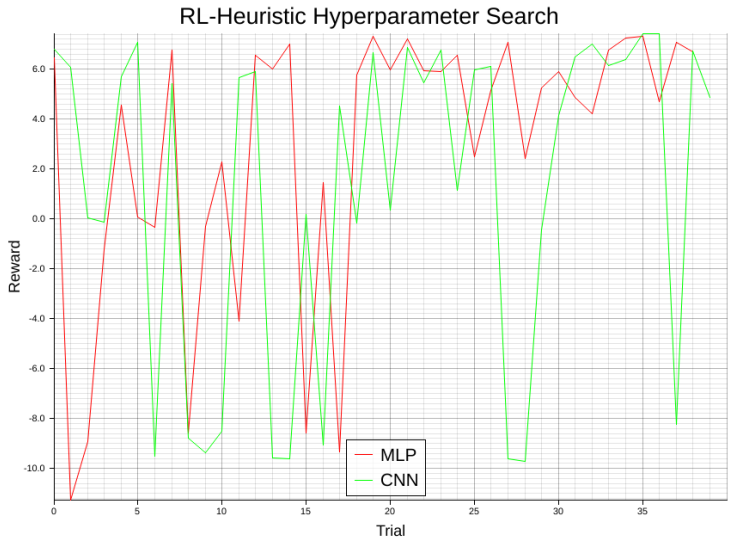


Fig. 15: The development of the achieved reward in the Optuna hyperparameter search for *RL-Heuristic*.

numerical array is only 26.56kb[30] big. Moreover, every number in the observation has a neuron in the input layer with a weight attached to it[2]. In an update of the weights after receiving a reward, all weights are adjusted[86], so having a smaller input layer results in faster training. Hence not using the image requires less memory and training is faster.

6.5 Reward Function Effectiveness

6.5.1 Setup

In order to figure out the effectiveness of the different reward functions (section 5.2.3), we reuse data collected for the reproducibility experiments (section 6.3). Per reward function, we collect the averages for the number of steps it took, the average reward achieved, and what the average score of the players was in the results.

6.5.2 Results

Our results, shown in Table 6, indicate that the *Guided* reward function works best. It only takes around 3183 steps on average to reach a local optimum, and the average scores achieved in the matches is the highest. The *Score* reward function also lets the agent reach a local optimum, but it takes twice as long as

30 $\frac{10*42*64}{1024}$

the *Guided* function, and the score is slightly lower as well. The *simple* reward function seems unfit for training. It never reached a local optimum in the 10,000 steps we allowed it to run and it got the lowest score in its games.

Reward Type	Amount of Steps	Episode Reward	Score
Guided	3183.49	-0.17	-5.6
Simple	10000.0	-0.0	-12.25
Score	6214.45	-0.09	-6.88

Table 6: Results of self-play with different reward types until either a local optimum or 10,000 steps have been reached. Step, Reward and Score show the average of all seeds.

6.6 MCTS Effectiveness

6.6.1 Setup

Initial test matches of MCTS against the *user heuristic* showed a zero percent win rate and a look at the game boards suggested a near-random play. We use a basic version of MCTS and a random default policy because heuristic guidance was too slow (see section 5.1.1). AlphaZero has shown that even games with high branching factor such as Go can be played well by MCTS when guided by a neural network[82]. However, without a network or a heuristic, we rely on random guidance. To prove that this guidance is at fault for the bad performance, we abuse the fact that the *user heuristic* plays very predictably (section 3.3). We use the RAVE-MCTS variant (without the POOL addition), pre-fill the RAVE values with 100 games of the *user heuristic* playing against itself and then let the MCTS play 100 matches against the *user heuristic*. We run this experiment with different RAVE β parameters (section 4.1.4) which is responsible for the exploration/exploitation. The closer the RAVE visits of a node reach β, the smaller the exploration component becomes. Furthermore, we will be using the slow heuristic default policy in this experiment.

6.6.2 Results

Our MCTS implementation can play well and have a high win rate, as can be seen in Figure 16. This result underlines that in games with high branching factors, MCTS needs good guidance through the tree in order to perform well. The declining win rate with a higher beta value suggests that exploration on an already partially explored game tree worsens the result if the opponent does not deviate from its paths. The rise in win rate for a β value of 5000 after the large drop in 2500 underlines the effect that the randomness involved in the search process can have. Even though the heuristic default policy worked in this

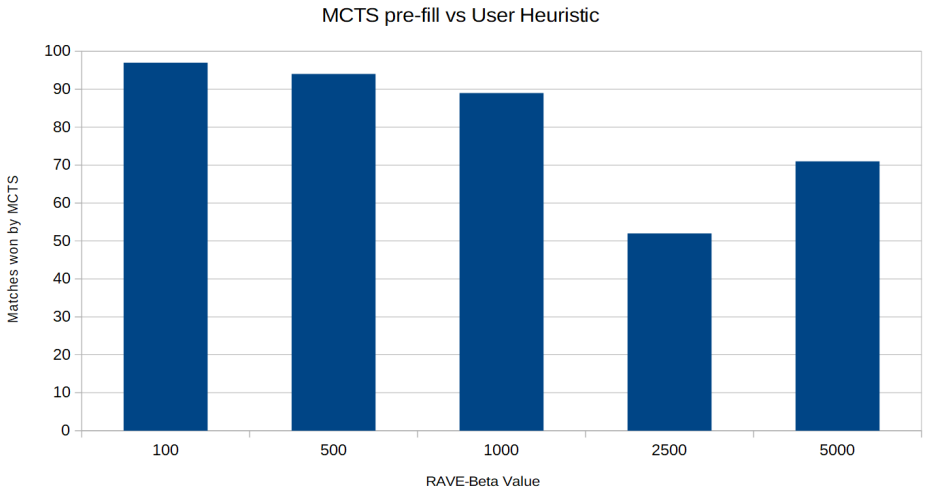


Fig. 16: Win rates of pre-filled MCTS playing against the *RL-Heuristic* compared by the RAVE-β parameter.

experiment, we will still use a random default policy for the tournament (section 6.8). This is because there will be no pre-filling the tree in the tournament and with the heuristic default policy nearly no nodes will be visited of the empty tree (section 5.1.1).

6.7 RL Agents Training

6.7.1 Setup

In this section, we detail the training process of the RL agents. Due to the reproducibility problematic (section 6.3), each training is done four times, and only the best run is shown. *RL-Selfplay* and *RL-Selfplay-Heuristic* are trained with the default PPO2 Hyperparameters given in section B.2. For the *RL-Heuristic*, we have tuned hyperparameters available from the observation space evaluation (section 6.4) that are given in section B.3.

Furthermore, we increase the hidden layer amount from two hidden layers with size 64 to three hidden layers with size 128 because increasing the network size decreases the chances of getting stuck in local optima[46].

When playing only against themselves, the networks still quickly reached a local optimum even with increased layer size. This optimum manifested in the same game being played on repeat and the reward per episode staying the same. This repetition is a known problem in self-play and can be called "chasing cycles"[91]. To prevent these local optima, we train five different agents against each other in random order. To be able to train against other agents, we directly modified *stable-baselines* code. The exact changes are detailed in C.2.

6.7.2 Results

The training process for *RL-Selfplay* can be seen in Figure 17. In the beginning, it keeps improving, but after peaking around 1.5 million steps, it only deteriorates. For *RL-Selfplay-Heuristic* we use the two best candidates from

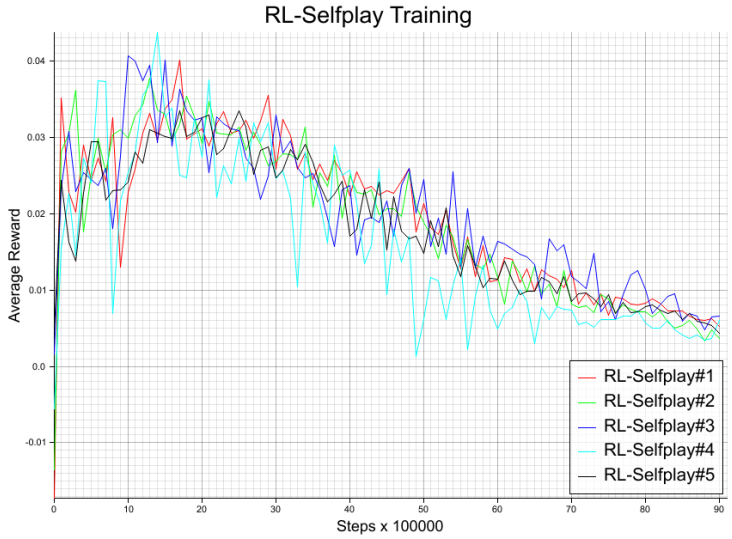


Fig. 17: The training process of *RL-Selfplay* visualised by the average achieved reward per 100,000 Steps.

RL-Selfplay namely #3 after one million steps with a reward of 0.04 and #1 after 1.5 million steps[31] with a reward of 0.034. Initially, we only trained for one million steps, but because it still seemed to improve its reward, we increased that amount to 4.6 million steps. This proved to be a good decision, as can be seen in Figure 18 because *RL-Selfplay#1-Heuristic* reaches its peak after 3.44 and *RL-Selfplay#3-Heuristic* after 3.64 Million steps with a reward of 0.032 and 0.024. This is our first pure RL trained agent that is able to achieve a positive reward while playing against the heuristic. A peculiar thing about the training process is that two different models in different training units have the same performance drops around the same step size.

Usually, a reward training graph should, although jittery, steadily improve and climb in the reward achieved[59]. We believe the constant deterioration, shown in Figure 17, to be caused by the way we implement self-play. After critical review, we realised that the opponent's decisions are bleeding into the training. For example, when model A plays against model B, then decisions,

[31] The actual peak is at 1.7 million steps, but the model was only saved every 500,000 steps.

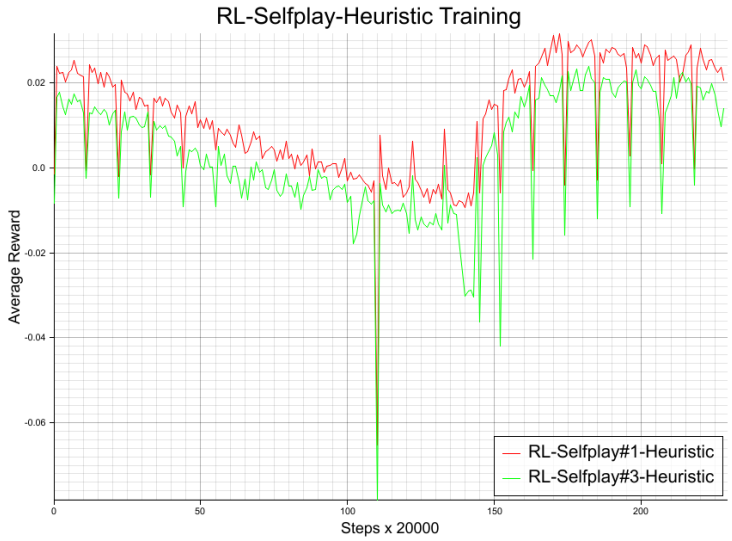


Fig. 18: The training process of *RL-Selfplay-Heuristic* visualised by the average achieved reward per 20,000 Steps.

observations and reward of both models are used to update model A, but model B stays unchanged. Only when model B is then the primary trained model and plays against model A is model B updated but model A not. So we do the same experiment again with the fix that decisions from model B do not factor into the training of model A (code see C.3). We call this agent *RL-Nobleed*. Unfortunately, as shown in Figure 19, the training still deteriorates after peaking. The achieved reward does not jitter as much anymore. Even though the training process goes on for longer, the deterioration is less. It only degrades to around 0.02 after around 12 million steps, whereas the previous version dropped down to 0.05 within just 9 million steps. So there is a notable improvement, but it did not solve the problem of deterioration. Furthermore, *RL-Nobleed* arrives at the same maximum reward of around 0.046, but it does so after 20-25 million steps instead of 5-15 million steps. The training of *RL-Nobleed-Heuristic*, portrayed in Figure 20, does not show us anything new or interesting. The peculiar periodic drops in the reward still occur.

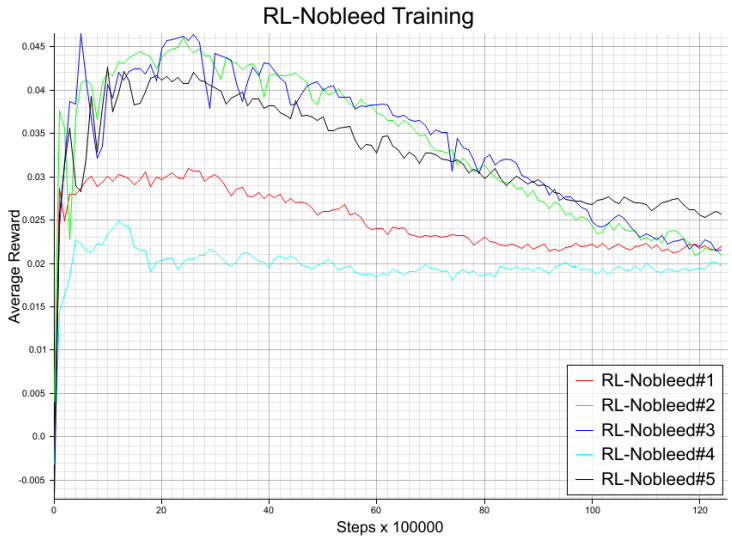


Fig. 19: The training process of *RL-Nobleed* visualised by the average achieved reward per 100,000 Steps.

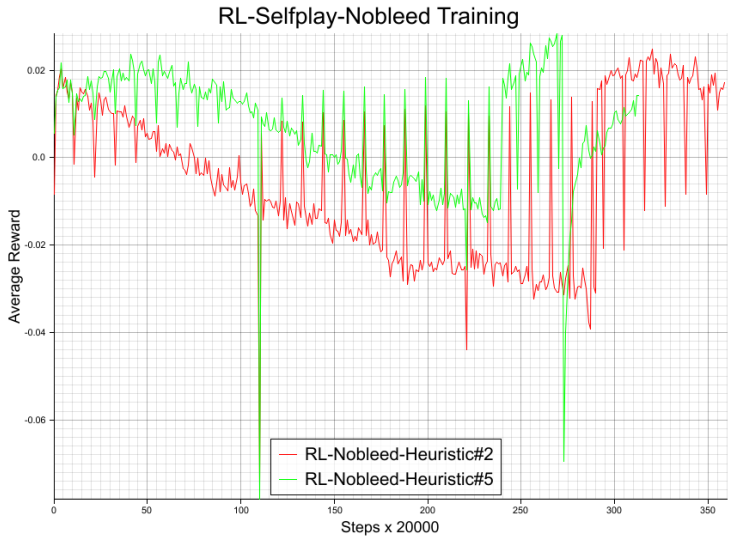


Fig. 20: The training process of *RL-Nobleed-Heuristic* visualised by the average achieved reward per 20,000 Steps.

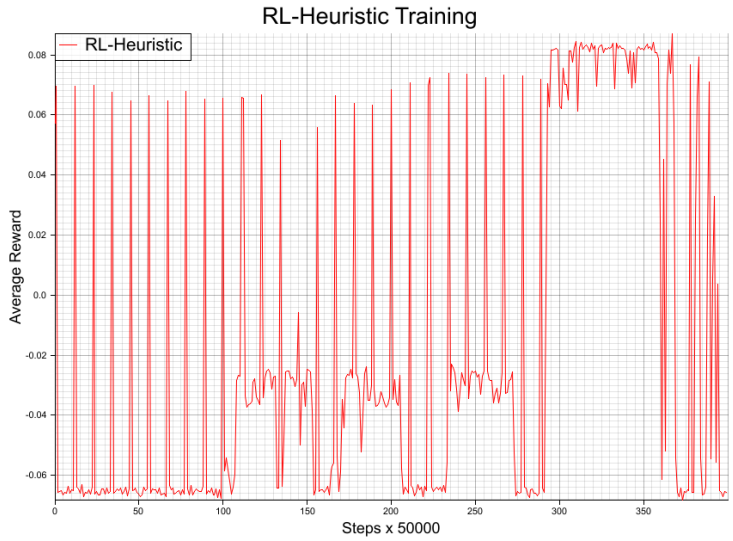


Fig. 21: The training process of *RL-Heuristic* visualised by the average achieved reward per 50,000 Steps.

The *RL-Heuristic* reward curve, visualised in Figure 21, is more jittery than the self-play training. We looked at the output values of *RL-Heuristic* and figured the reward function design was a mistake. It sets the weights for its own score and group values to zero, the enemy block value to fifteen and the connectability varies between four and seven. So by negating the players score with the opponent's score, we have unwillingly forced the heuristic to focus only on blocking the opponent over everything else. Needless to say with these weights, *RL-Heuristic* rarely wins. Although it manages to keep opponents score low, it does not focus on gaining points which leaves it with a point disadvantage.

6.8 Tournament

6.8.1 Setup

In the tournament, we will pit all previously shown AI approaches against each other. Every bot will play 100 matches against every other bot. In the end, every bot will have played 1400 matches. We have nine different RL bots (see section 6.7), three MCTS bots (see section 4.1) and three heuristic bots (see section 4.3). The parameter settings for the MCTS bots can be found in section B.4. The bots skill will be compared via a BBT skill rating (section 4.4). We expect MCTS to play badly due to its random default policy (section 6.6) in a high branching factor environment. From the training of the RL-Agents (section 6.7) we know that they can hardly beat the heuristic. Hence we expect the heuristic to play the best followed by RL and MCTS.

6.8.2 Results

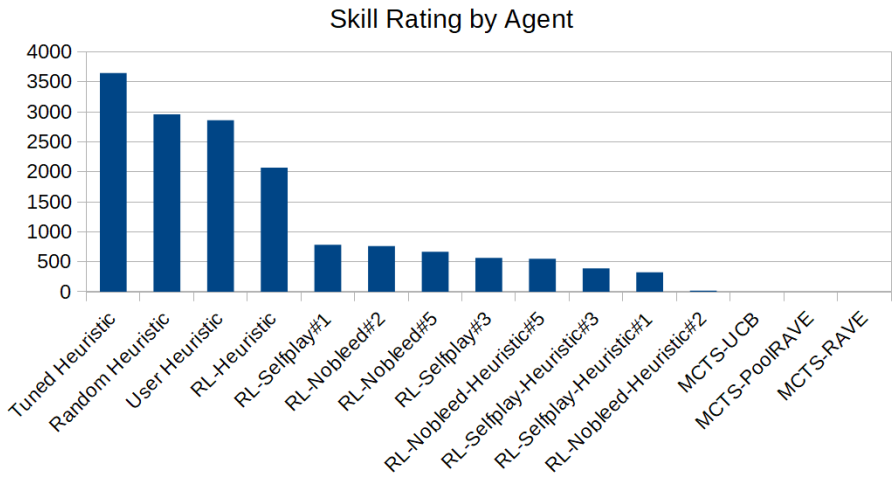


Fig. 22: The skill rating (section 4.4) of the agents that participated in the tournament.

The skill rating portrayed in Figure 22 confirms our expectations. The three heuristic agents take the top 3, followed by RL and MCTS. Remarkably, the *tuned heuristic* performed best, even though it is only optimised to play well against the *user heuristic*, but yet it performs best across all agents. In the human play experiment, the *random heuristic* performed better than the *user heuristic*, which the skill rating confirms.

Seeing *RL-Heuristic* as the best *RL* approach further proves that the other RL agents have not learned anything close to good gameplay. In section 6.7, we have shown that *RL-Heuristic* only focuses on blocking off the opponent, so even though it uses the heuristic to make decisions, the weights make it choose bad actions. Interestingly, there seems to be little difference in the skill of *RL-Selfplay* and *RL-Nobleed*, even though *RL-Nobleed* tries to improve upon *RL-Selfplay*. Seeing that all RL agents consistently beat MCTS with a random default policy proves that the agents definitely learned to play something. However, it is not good enough yet to beat the heuristic.

We know that a random default policy for MCTS is infeasible for good play, so the bad skill rating in this tournament is expected. In a previous experiment (section 6.6), we have shown that if MCTS analyses paths that are very probable to be taken by the enemy, then it works excellent with win rates close to 90+%. However, it is interesting that the MCTS-UCB variant performed best because the other two variants try to improve the performance of UCB via slight modifications (section 4.1). Considering that the heuristic was not even strong enough to beat beginner human players (section 6.2), but yet leads this skill rating shows that there is still much work to be done to improve the strength of Tetris Link agents.

The skill rating omits information about the quality of the individual moves. To gain further insight into that, we provide Figure 23. Here we can see that every agent manages at least once to gain 8 points or more. This means that every agent had at least one match where they played reasonably well. Looking at the lowest achieved score and average score, we can see that every agent except for the pure heuristic ones play badly, considering that on average, they only make ± 3 points.

The lousy performance raises the question of whether it was chance that every agent had at least one good match with 8+ points. To find that out, we inspect the board state from some of the matches in which agents achieved their maximum score. Looking at the best match of *MCTS-RAVE* in Figure 24a, we can see that the tree search can work even with a random default policy. Only three blocks at the right are placed badly and never connect. All other blocks are perfectly connected. *RL-Nobleed#2*'s best match, shown in Figure 24b, suggests a good understanding of the core game principle. Only two blocks at the top right are not connected in a group because the enemy blocked it off. Other than that, only four blocks are put down so that minus points are taken. Only one mistake less and the match could have been a draw. We know that *RL-Heuristic* focuses on blocking off its opponent while completely ignoring its own points (section 6.7). Yet in a match portrayed in Figure 24c, it still managed to achieve 16 points. Blocking off is very successful in this match, and the opponent only manages once to connect one single group of three blocks to gain points. Based on these three matches, we conclude that it was not up to chance but up to skill that these agents gained the number of points they did.

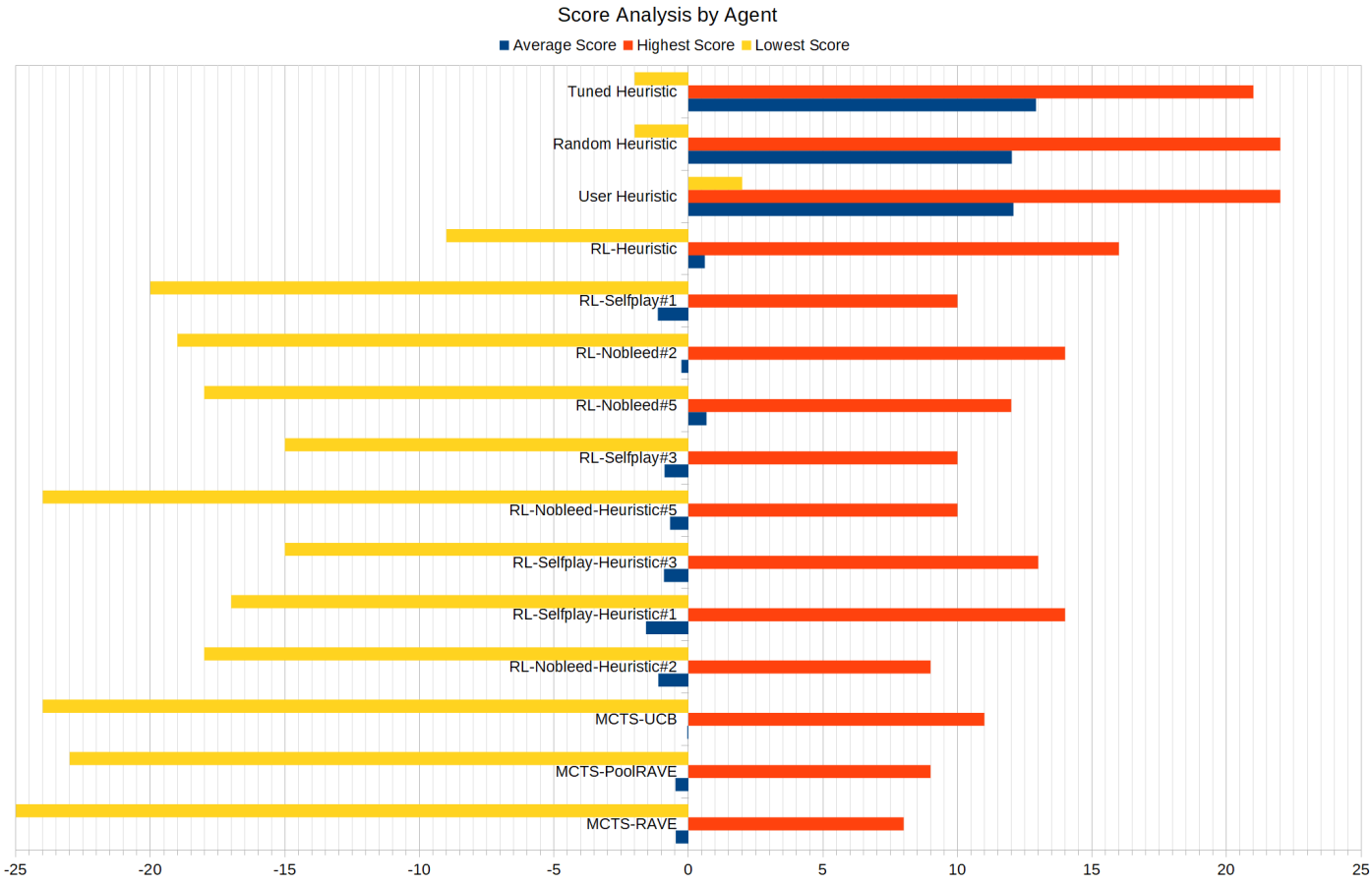
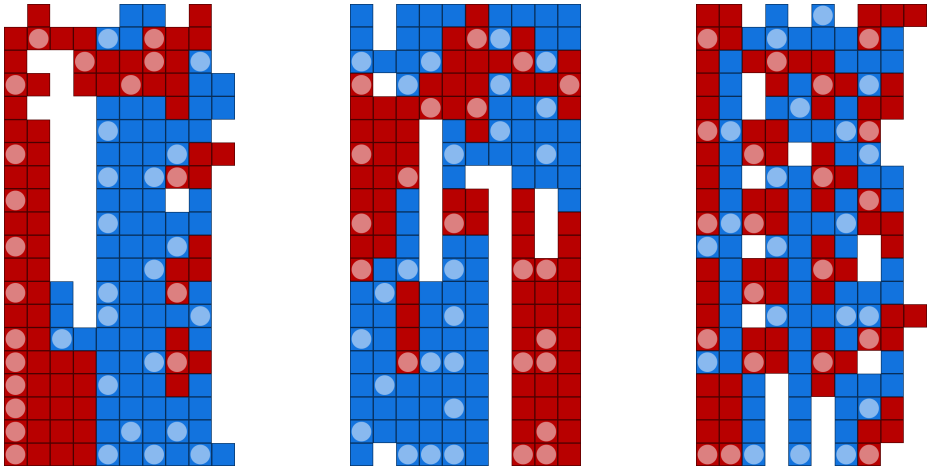


Fig. 23: Visualisation of the scores that agents achieved in the tournament. Agents are sorted by the skill rating portrayed in Figure 22.



(a) *MCTS-RAVE* (red, 9 points) vs. *Random heuristic* (blue, 18 points)

(b) *Tuned heuristic* (red, 16 points) vs *RL-Nobleed#2* (blue, 14 points).

(c) *RL-N-H#5* (red, -15 points) vs *RL-Heuristic* (blue, 16 points).

Fig. 24: Selection of screenshots from tournament matches where an agent achieved their maximum score.

6.9 Verifying the MCTS Implementation

6.9.1 Setup

Because MCTS takes the last place in the tournament (section 6.8), we want to prove that our MCTS implementation is not faulty. With a heuristics pre-filled tree (section 6.6) the implementation can perform well in Tetris Link against a limited opponent.

The fact that the default policy is crucial to MCTS performance can be seen in practical applications such as the board game Hex. MoHex 2.0, the 2013 Hex champion[26], uses a special default policy and includes learned patterns to guide the tree search[35]. For an even stronger agent, Mohex v3 now uses neural networks to guide through the search, just like AlphaGo[21]. As further correctness proof, we implement a version of the board game Hex in Rust. We use a simple heuristic based on Dijkstra's shortest path algorithm[16], to have something to compare MCTS to. In a test with a 7x7 Hex game, our tree parallelised MCTS implementation can visit around 35000 nodes per second with a random default policy and 4500 nodes per second with a heuristic default policy. This implementation of Hex is not optimised for performance as the Tetris Link implementation is, so a higher visit count could probably be reached. We believe that because for Hex the heuristic policy is reasonably fast, the better-guided version will perform a lot better. Hence we will let all three MCTS agents play against the heuristic once with a random and once with a heuristic

default policy. In Hex, the first player has a definite advantage[4]. This can be countered by the pie rule stating that the second player can switch colour in their first turn[48]. We play without the pie rule and give the heuristic the first turn advantage so that MCTS has to outplay the heuristic to win. To also underline the effect of the branching factor on MCTS performance, we will play on Hex boards of size 2x2 to 12x12.

6.9.2 Result

The same MCTS implementation that lost in the Tetris Link tournament can win consistently in smaller Hex grids regardless of the default policy suggesting that the implementation is not at fault.

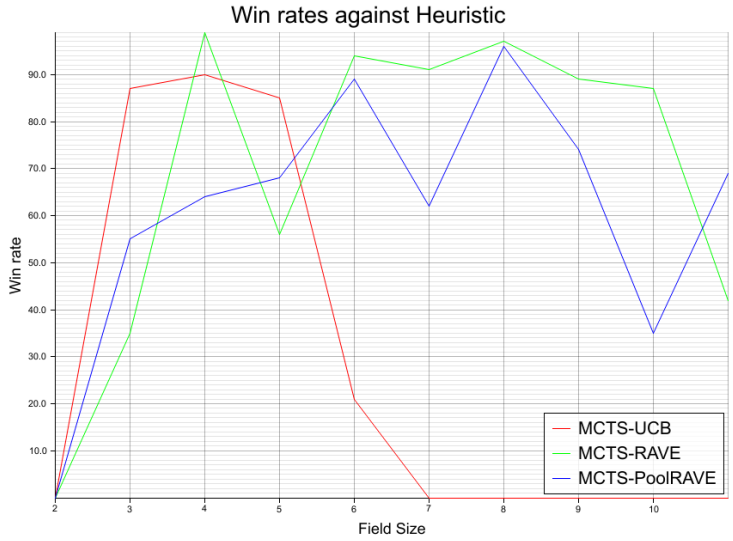


Fig. 25: The win rate of our MCTS implementation with a heuristic default policy playing against the dijkstra based heuristic in the game of Hex.

By comparing Figure 25 with Figure 26, one can clearly see that the heuristics default policy outperforms a random default policy. No variant wins in a 2x2 field because we play without the pie rule. The heuristic always makes the same first move, and the second player never has the ability to win, as shown in Figure 27. In bigger sizes, MCTS has a chance because the heuristic is not guaranteed to play perfectly and if a certain path is blocked, then it might make a suboptimal move allowing MCTS to take the lead.

For the heuristic default policy, UCB performs best in smaller search spaces, but as soon as the field is bigger than 7x7, UCB can not hold up anymore. RAVE seems to have problems with too small search spaces, but as soon as the

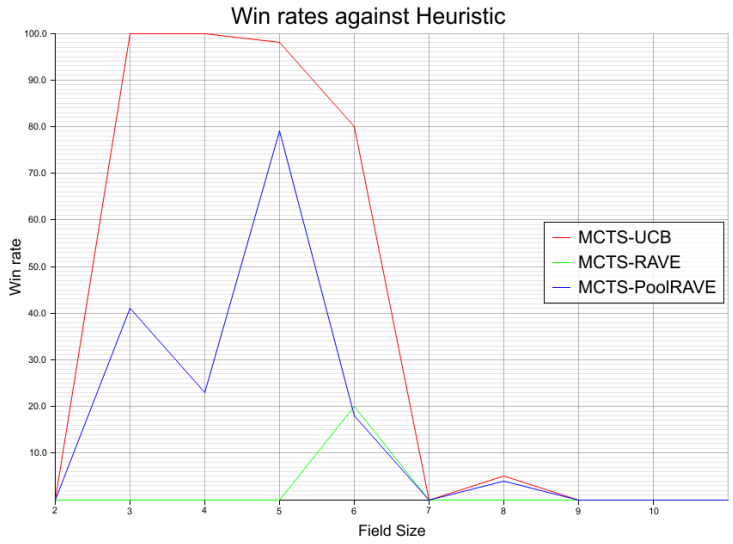


Fig. 26: The win rate of our MCTS implementation with a random default policy playing against the dijkstra based heuristic in the game of Hex.

grid size is larger than three, it consistently wins the majority of the matches (up until 11x11). PoolRAVE wins most consistently but also slowly deteriorates with growing fields. Throughout all three variants, a regression of performance with an increasing branching factor can be seen. This further underlines that in games like Tetris Link with a rather large branching factor, the guidance through the tree is essential for success and also the reason for the bad tournament performance. This is also shown by MoHex v3, where they use a neural network as guidance through the tree to further increase performance over their previous custom-built default policy for the game of Hex[21]. Further underlining the effect of the branching factor is the random default policy result in Figure 26. As long as the branching factor does not exceed 36 (6x6), there is a good chance that matches can be won with UCB. Anything above that is almost impossible to win regardless of MCTS variant.

Besides the branching factor influence, we believe that a bad turn in Hex has fewer consequences than in Tetris Link[32]. First of all, Hex has no concept of minus points. One can only win or lose. Moreover, completely blocking access to one piece requires up to six enemy pieces around it in Hex. In Tetris Link, one to two turns can be enough to block off multiple blocks completely (see figure 6b in section 3.4).

[32] Assuming that both players do not play perfectly.

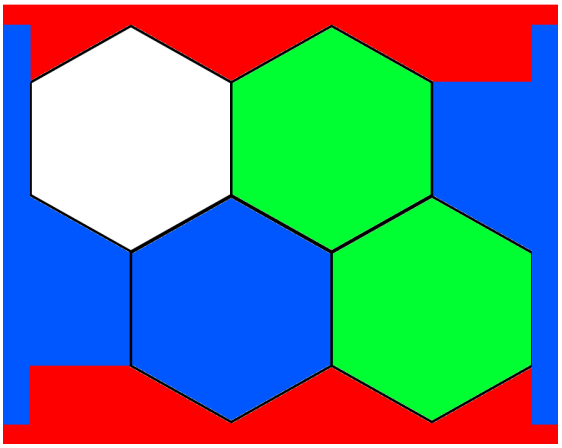


Fig. 27: This board shows why no MCTS variant (red) can win against the heuristic (blue) in 2x2 fields without the pie rule. The heuristic will always choose the lower left as the first move, which gives two options (green) to win in the next turn. Hence red can never win in this situation because it can only fill one of the green spots and blue will take the other one and win.

7 Conclusion

Board game strategy analysis has been done for decades, and especially games like Chess and Go have seen countless papers analysing the game, patterns and more to find the best play strategies[82]. We contributed to that field by taking a close look at the board game Tetris Link. We have shown play strategies (section 3.4) that can be used to develop a reasonably well playing heuristic (section 4.3). While the strategy is key to winning, some games, such as Hex, give the first player a definite advantage. In our experiments, there is no clear advantage for the starting player in Tetris Link (section 3.3).

With a solid understanding of the game itself, we investigated different approaches for AI agents to play the game, namely heuristic, RL and MCTS. We have shown that all tested approaches can perform well against certain opponents. The best automatic approach has proven to be the heuristic (section 4.3) although it can not consistently beat human players even when they are new to the game (section 6.2.2).

Training an RL agent (section 6.7) for Tetris Link has proven to be complicated. Just getting the network to produce positive rewards required much trial and error, and in the end, the agent did not perform very well even when consistently achieving a positive reward. We believe the learning difficulty in Tetris Link comes from the many opportunities to make minus points in the game. One turn offers at most one plus point, or three and more if a group is connected, but that means that the previous two or more turns at most gave zero points if not even more minus points. Hence recovering from minus points

is difficult, and the quickest way to do it is to make the enemy take minus points. However, that is also a risky strategy that can backfire (section 3.4).

Reproducible training is also a problem we stumbled over. It is well known that neural network algorithms results can differ strongly by just changing little of the input (e.g. data, or parameters such as the random seed). Nevertheless, if the input stays the same, then the results should also stay unchanged. However, in our experiments (section 6.3), that was not the case, and less than half of the time, training was actually reproducible. This is a known problem related to *stable-baselines* and *TensorFlow*, of which developers are aware and actively working on[61,32]. Our reproducibility problems only focused on the code that runs it, but there are other reproducibility problems in the AI field. At the ICLR challenge in 2018, where they asked researchers to try to reproduce papers, only 32.7% of checked papers were mostly reproducible[70][33]. Luckily people are working on identifying sources of reproducibility issues and propose solutions[28]. Software like OpenAI Gym tackles one of the problems. Namely, uniform environments for more reliable benchmarks across different RL algorithm implementations. So with our Tetris Link Gym implementation, we offer researchers another environment for benchmarking. The used software environment is also still rapidly growing and developing, as can be seen with OpenAI's standardisation of *PyTorch*[64].

Although MCTS performed poorly in our tournament, we have shown that with proper guidance through the tree MCTS can perform very well in Tetris Link (section 6.6) and also Hex (section 6.9). That is why a combination where RL guides an MCTS through the tree works very well, e.g. AlphaZero[82] or MoHex v3[21], and is something to try in future work (section 7.2).

7.1 Contributions

This section poses and answers research questions that were encountered during the work on this thesis.

For Tetris Link, our main question was: Which of the AI approaches performs best in a tournament? In our experiment (section 6.8), the heuristic works best followed by RL and MCTS. Because of MCTS bad performance, we wondered: How big is the branching factor and how does it develop over the course of a match? The branching factor is 162, and through a match, it slowly declines (section 3.2). However, the available moves decline faster if the move quality is bad, e.g. a random player. In the game of Hex, the first player has an advantage which is the reason for the pie rule. Because of that, we speculated: Does the starting player have an advantage in Tetris Link? By doing an experiment, we found out that the starting player had a win rate of around 48%, which does not indicate an advantage (section 3.3).

To build a heuristic, we stumbled over the question: What numerical values can be extracted from a game state to build a heuristic? We found four different strategies that can be represented as numerical values (section 3.4 & 4.3).

[33] To enable reproducibility of this work all used code is available at https://github.com/Hizoul/masterthesis

Considering that the heuristic is the winner of the tournament reassures us that these four measures have been a good choice. We know that compared to our approaches, the heuristic is the strongest, but one question remains: How does the heuristic fair out against humans? In our experiments, the heuristic is easily beatable even by beginner players (section 6.2). The *user heuristic* had a win rate of 33.33% and the *random heuristic* a win rate of 23.07%.
Our initial experiments with MCTS in Tetris Link were quite unsuccessful, and it also comes in last in the tournament. That is why we asked ourselves: Is a random default policy actually feasible for a game with a high branching factor? To find this out, we let our MCTS implementation play in increasingly big Hex fields, and we can see a clear deterioration as the field size and hence branching factor increases (section 6.9). Because the random default policy did not work well in Tetris Link, we wondered: Could the heuristic be used as default policy? It is possible, and we did try it out however our heuristic calculation is too slow and did not allow the MCTS to inspect a lot of nodes which makes it infeasible for usage (section 5.1.1). From the Hex MCTS verification (section 6.9), we know that MCTS can work well in a smaller branching factor. In the first player advantage experiment, we have seen that the *user heuristic* plays very deterministically without branching out a lot so we thought: Can pre-filling the search tree make up for a bad default policy? The answer is yes. Through pre-filling the tree MCTS achieved a win rate of 94% against the *user heuristic*.
In order to use a Reinforcement Learning agent, an environment needs to be built. An environment consists of an observation space, action space and a reward (section 4.2). For each of these parts, questions arose during implementation. The observation space can be passed as a numerical array processed by an MLP or an image processed by a CNN, which brings the question: Does it affect training? In an experiment, we found out that there is no measurable impact on training success (section 6.4). However, size-wise an image requires more bytes which in turn also leads to slower training. The action space can be a discrete number for the specific move to make or a probability distribution over all possible actions. So the question is: Which of the two should one use? With a discrete number, we were not able to bring the RL agent to learn anything, so we had to go with the probability distribution where invalid moves are masked. The reward is the trickiest part because it allows for much more flexibility. What should be rewarded? One thing is for sure: To win needs to increase and to loose decrease the reward. However, should it only reward at the end of a game, or also give the delta of the current score after every move? Furthermore, should bad suggestions where the highest number had to be masked because it was an illegal move be scolded? In our experiment giving as much information in the reward worked best, so we give the delta of the score, scold for answers that had to be corrected and increase/decrease the reward for a win/loss. In the training of the *RL-Heuristic*, we also saw that with a bad decision in the reward, one could guide the agent to learn the wrong thing. In this instance, it learned to ignore its own points and focus only on blocking the opponent because, as a reward, we gave the difference in points between the agent and its opponent. If

opponents make fewer points because they are being blocked, then the reward of the agent is less negative. Because of reproducibility issues, we wondered: Does training with the exact same parameters yield the same result if done multiple times? In theory, it definitely should. Every algorithm is deterministic, and if the pseudo-random suggestions are deterministic by using the same seed, then the training result should be the same. In our experiment, 20 out of 52 seeds were reproducible. This is not the expectation, and something in the code base is causing these issues. However, we also used alpha software that is actively working on this problem, so this might be solved soon.
Moreover, we contributed code to open source projects in the RL field. We fixed the action and observation space code for the OpenAI Gym *GymGo* (section 5.2), and we have modified the *stable-baselines* to allow agents to play against themselves or even other agents (section 5.3 and C.2).

7.2 Future Work / Limitations

Our experiments are far from exhaustive, and there is still much that could be tried in order to improve the results. First of all Tetris Link is a game for two to four players with imperfect information through dice rolling. Our experiments and analyses have focused only on a two-player setup without dice rolling to have perfect information. Increasing player count and introducing imperfect information would impose a whole new difficulty level on the AI approaches. Furthermore, the reward function test (section 6.5) is limited in scope. Longer training units that are not cut off after 10,000 steps and also a tournament at the end to determine the best reward function would increase the relevance of the results. There is also a lack of variety in RL algorithms. All agents are using PPO2, but especially for the reward function test seeing whether the results are consistent across different algorithms would be interesting. For the tournament (section 6.8) it would be nice to test an agent that combines RL and MCTS like AlphaZero[82] to verify that the combination of the two also works well in Tetris Link. Moreover, the branching factor development (section 3.2) and first turn advantage (section 3.3) analysis are limited to data generated by the heuristic and random play. More agents that play the game more human-like would underline the correctness of the results. This paper only uses RL implementations from OpenAI (gym and baselines). Other frameworks, such as OpenSpiel[45], could be evaluated as alternatives. Even more interesting would be to do the reproducibility (section 6.3) experiment with different implementations of the same algorithm and the same hyperparameters.

8 References

Out of all 97 references 19 (≈ 20%) are websites, and 3 of these websites are Wikipedia articles.

1. Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al.: Tensorflow: A system for large-scale machine learning. In: 12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16). pp. 265–283 (2016)
2. Abraham, A.: Artificial neural networks. Handbook of measuring system design (2005)
3. Akiba, T., Sano, S., Yanase, T., Ohta, T., Koyama, M.: Optuna: A next-generation hyperparameter optimization framework. In: Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. pp. 2623–2631. ACM (2019)
4. Arneson, B., Hayward, R.B., Henderson, P.: Solving hex: beyond humans. In: International Conference on Computers and Games. pp. 1–10. Springer (2010)
5. Balaji, P., German, X., Srinivasan, D.: Urban traffic signal control using reinforcement learning agents. IET Intelligent Transport Systems **4**(3), 177–188 (2010)
6. Barto, A.G., Sutton, R.S., Anderson, C.W.: Neuronlike adaptive elements that can solve difficult learning control problems. IEEE transactions on systems, man, and cybernetics (5), 834–846 (1983)
7. Borowiec, S.: Alphago seals 4-1 victory over go grandmaster lee sedol. The Guardian **15** (2016)
8. Bradford, G.: The History and Analysis of the Supposed Automation Chess Player of M. de Kempelen: Now Exhibiting in this Country, by Mr. Maelzel. Hilliard, Gray & Company (1826)
9. Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., Zaremba, W.: Openai gym. arXiv preprint arXiv:1606.01540 (2016)
10. Browne, C.B., Powley, E., Whitehouse, D., Lucas, S.M., Cowling, P.I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., Colton, S.: A survey of monte carlo tree search methods. IEEE Transactions on Computational Intelligence and AI in games **4**(1), 1–43 (2012)
11. Burgiel, H.: How to lose at tetris. The Mathematical Gazette **81**(491), 194–200 (1997)
12. Carr, D.: Applying reinforcement learning to tetris. Department of Computer Science Rhodes University (2005)
13. Chen, Y., Huang, A., Wang, Z., Antonoglou, I., Schrittwieser, J., Silver, D., de Freitas, N.: Bayesian optimization in alphago. arXiv preprint arXiv:1812.06855 (2018)
14. David Rubio, C.H.: tensorflow - how to reproduce neural network training with keras (2020-02-02), `https://ai.stackexchange.com/questions/17412/how-to-reproduce-neural-network-training-with-keras/17415#17415`
15. Demaine, E.D., Demaine, M.L.: Jigsaw puzzles, edge matching, and polyomino packing: Connections and complexity. Graphs and Combinatorics **23**(1), 195–208 (2007)
16. Dijkstra, E.W., et al.: A note on two problems in connexion with graphs. Numerische mathematik **1**(1), 269–271 (1959)
17. van Eck, N.J., van Wezel, M.: Application of reinforcement learning to the game of othello. Computers & Operations Research **35**(6), 1999–2017 (2008)

18. Edelkamp, S., Korf, R.E.: The branching factor of regular search spaces. In: AAAI/IAAI. pp. 299–304 (1998)
19. Enzenberger, M., Müller, M.: A lock-free multithreaded monte-carlo tree search algorithm. In: Advances in Computer Games. pp. 14–20. Springer (2009)
20. Fernandez, J.M.F., Mahlmann, T.: The dota 2 bot competition. IEEE Transactions on Games (2018)
21. Gao, C., Hayward, R., Müller, M.: Move prediction using deep convolutional neural networks in hex. IEEE Transactions on Games **10**(4), 336–343 (2017)
22. Glickman, M.E., Jones, A.C.: Rating the chess rating system. CHANCE-BERLIN THEN NEW YORK- **12**, 21–28 (1999)
23. Google: "tetris link" - google schloar (2019-09-09), `https://scholar.google.nl/scholar?hl=nl&as_sdt=0%2C5&q=%22Tetris+Link%22&btnG=`
24. Google: Hill: stable baselines - google schloar (2019-09-15), `https://scholar.google.nl/scholar?cites=7029285800852969820`
25. Grzes, M., Kudenko, D.: Plan-based reward shaping for reinforcement learning. In: 2008 4th International IEEE Conference Intelligent Systems. vol. 2, pp. 10–22. IEEE (2008)
26. Hayward, R., Arneson, B., Huang, S.C., Pawlewicz, J.: Mohex wins hex tournament. ICGA Journal **36**(3), 180–183 (2013)
27. Helmbold, D.P., Parker-Wood, A.: All-moves-as-first heuristics in monte-carlo go. In: IC-AI. pp. 605–610 (2009)
28. Henderson, P., Islam, R., Bachman, P., Pineau, J., Precup, D., Meger, D.: Deep reinforcement learning that matters. In: Thirty-Second AAAI Conference on Artificial Intelligence (2018)
29. Herbrich, R., Minka, T., Graepel, T.: Trueskill$^{\text{TM}}$: a bayesian skill rating system. In: Advances in neural information processing systems. pp. 569–576 (2007)
30. Hill, A., Raffin, A., Ernestus, M., Gleave, A., Kanervisto, A., Traore, R., Dhariwal, P., Hesse, C., Klimov, O., Nichol, A., Plappert, M., Radford, A., Schulman, J., Sidor, S., Wu, Y.: Stable baselines. `https://github.com/hill-a/stable-baselines` (2018)
31. Hill, A., Raffin, A., Ernestus, M., Gleave, A., Kanervisto, A., Traore, R., Dhariwal, P., Hesse, C., Klimov, O., Nichol, A., Plappert, M., Radford, A., Schulman, J., Sidor, S., Wu, Y.: Dealing with nans and infs (2019-10-24), `https://stable-baselines.readthedocs.io/en/master/guide/checking_nan.html`
32. Hill, A., Raffin, A., Ernestus, M., Gleave, A., Kanervisto, A., Traore, R., Dhariwal, P., Hesse, C., Klimov, O., Nichol, A., Plappert, M., Radford, A., Schulman, J., Sidor, S., Wu, Y.: Rl algorithms, stable baselines 2.9.0a0 documentation (2019-11-20), `https://stable-baselines.readthedocs.io/en/master/guide/algos.html`
33. Hoos, H., Leyton-Brown, K.: An efficient approach for assessing hyperparameter importance. In: International conference on machine learning. pp. 754–762 (2014)
34. Huang, D.: How much did alphago zero cost? (2020-02-04), `https://www.yuzeh.com/data/agz-cost.html`
35. Huang, S.C., Arneson, B., Hayward, R.B., Müller, M., Pawlewicz, J.: Mohex 2.0: a pattern-based mcts hex player. In: International Conference on Computers and Games. pp. 60–71. Springer (2013)
36. Hutson, M.: Artificial intelligence faces reproducibility crisis (2018)
37. Jacobsen, E.J., Greve, R., Togelius, J.: Monte mario: platforming with mcts. In: Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation. pp. 293–300. ACM (2014)

38. Jaderberg, M., Dalibard, V., Osindero, S., Czarnecki, W.M., Donahue, J., Razavi, A., Vinyals, O., Green, T., Dunning, I., Simonyan, K., et al.: Population based training of neural networks. arXiv preprint arXiv:1711.09846 (2017)
39. Jangmin, O., Lee, J., Lee, J.W., Zhang, B.T.: Adaptive stock trading with dynamic asset allocation using reinforcement learning. Information Sciences **176**(15), 2121–2147 (2006)
40. Jun, L., Ling, L.: Comparative research on python speed optimization strategies. In: 2010 International Conference on Intelligent Computing and Integrated Systems. pp. 57–59. IEEE (2010)
41. Kaelbling, L.P., Littman, M.L., Moore, A.W.: Reinforcement learning: A survey. Journal of artificial intelligence research **4**, 237–285 (1996)
42. Keck, P.: Analysing and improving the crypto ecosystem of Rust. Master's thesis (2017)
43. Kocsis, L., Szepesvári, C.: Bandit based monte-carlo planning. In: European conference on machine learning. pp. 282–293. Springer (2006)
44. Konidaris, G., Barto, A.G.: Skill discovery in continuous reinforcement learning domains using skill chaining. In: Advances in neural information processing systems. pp. 1015–1023 (2009)
45. Lanctot, M., Lockhart, E., Lespiau, J.B., Zambaldi, V., Upadhyay, S., Pérolat, J., Srinivasan, S., Timbers, F., Tuyls, K., Omidshafiei, S., et al.: Openspiel: A framework for reinforcement learning in games. arXiv preprint arXiv:1908.09453 (2019)
46. Lawrence, S., Giles, C.L., Tsoi, A.C.: Lessons in neural network training: Overfitting may be harder than expected. In: AAAI/IAAI. pp. 540–545. Citeseer (1997)
47. Lee, B., Jackson, A., Madams, T., Troisi, S., Jones, D.: Minigo: A case study in reproducing reinforcement learning research
48. Liang, X., Wei, T., Wu, I.C.: Job-level uct search for solving hex. In: 2015 IEEE Conference on Computational Intelligence and Games (CIG). pp. 222–229. IEEE (2015)
49. Liaw, R., Liang, E., Nishihara, R., Moritz, P., Gonzalez, J.E., Stoica, I.: Tune: A research platform for distributed model selection and training. arXiv preprint arXiv:1807.05118 (2018)
50. Lillicrap, T.P., Hunt, J.J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., Wierstra, D.: Continuous control with deep reinforcement learning. arXiv preprint arXiv:1509.02971 (2015)
51. Mańdziuk, J.: Mcts/uct in solving real-life problems. In: Advances in Data Analysis with Computational Intelligence Methods, pp. 277–292. Springer (2018)
52. Marschark, E.D., Baenninger, R.: Modification of instinctive herding dog behavior using reinforcement and punishment. Anthrozoös **15**(1), 51–68 (2002)
53. Marsland, T.A.: Computer chess methods. Encyclopedia of Artificial Intelligence **1**, 159–171 (1987)
54. Matsumoto, M., Nishimura, T.: Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. ACM Transactions on Modeling and Computer Simulation (TOMACS) **8**(1), 3–30 (1998)
55. McBride, A.: The human–dog relationship. The Waltham book of human–animal interactions: Benefits and responsibilities of pet ownership pp. 99–112 (1995)
56. Merriam, S.B.: Qualitative Research and Case Study Applications in Education. Revised and Expanded from" Case Study Research in Education.". ERIC (1998)
57. Metz, C.: Inside openai, elon musk's wild plan to set artificial intelligence free. Wired. Retrieved from https://www.wired.com/2016/04/openai-elon-musk-sam-altman-plan-to-set-artificial-intelligence-free (2016)

58. Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., Riedmiller, M.: Playing atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602 (2013)
59. Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G., et al.: Human-level control through deep reinforcement learning. Nature **518**(7540), 529 (2015)
60. Nagarajan, P., Warnell, G., Stone, P.: The impact of nondeterminism on reproducibility in deep reinforcement learning (2018)
61. NVIDIA: Nvidia/tensorflow-determinism: Tracking, debugging, and patching non-determinism in tensorflow (2019-07-05), https://github.com/NVIDIA/tensorflow-determinism
62. Oh, K.S., Jung, K.: Gpu implementation of neural networks. Pattern Recognition **37**(6), 1311–1314 (2004)
63. OpenAI: gym/environments.md at master - openai/gym (2019-11-11), https://github.com/openai/gym/blob/master/docs/environments.md#third-party-environments
64. OpenAI: Openai => pytorch (2020-02-02), https://openai.com/blog/openai-pytorch/
65. Orenstein, P.: Does experiential learning improve learning outcomes in an undergraduate course in game theory–a preliminary analysis
66. Pan, S.J., Yang, Q.: A survey on transfer learning. IEEE Transactions on knowledge and data engineering **22**(10), 1345–1359 (2009)
67. Parisotto, E., Ba, J.L., Salakhutdinov, R.: Actor-mimic: Deep multitask and transfer reinforcement learning. arXiv preprint arXiv:1511.06342 (2015)
68. Pfeiffer, M.: Reinforcement learning of strategies for settlers of catan. In: Proceedings of the International Conference on Computer Games: Artificial Intelligence, Design and Education (2004)
69. Philidor, F.D.: Analysis of the Game of Chess. P. Elmsly (1790)
70. Pineau, J.: Reproducibility, reusability, and robustness in deep reinforcement learning (2020-01-18), https://www.youtube.com/watch?v=Vh4H0gOwdIg, international Conference on Learning Representations (ICLR)
71. Pineau, J., Sinha, K., Fried, G., Ke, R.N., Larochelle, H.: Reproducibility challenge (2020-01-18), https://www.cs.mcgill.ca/~jpineau/ICLR2018-ReproducibilityChallenge.html, international Conference on Learning Representations (ICLR)
72. Rimmel, A., Teytaud, F., Teytaud, O.: Biasing monte-carlo simulations through rave values. In: International Conference on Computers and Games. pp. 59–68. Springer (2010)
73. Romanycia, M.H., Pelletier, F.J.: What is a heuristic? Computational Intelligence **1**(1), 47–58 (1985)
74. Sahba, F., Tizhoosh, H.R., Salama, M.M.: Application of opposition-based reinforcement learning in image segmentation. In: 2007 IEEE Symposium on Computational Intelligence in Image and Signal Processing. pp. 246–251. IEEE (2007)
75. Schaal, S., Atkeson, C.G.: Assessing the quality of learned local models. In: Advances in neural information processing systems. pp. 160–167 (1994)
76. Schadd, M.P., Winands, M.H., Van Den Herik, H.J., Aldewereld, H.: Addressing np-complete puzzles with monte-carlo methods. In: Proceedings of the AISB 2008 Symposium on Logic and the Simulation of Interaction and Reasoning. vol. 9, pp. 55–61 (2008)
77. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O.: Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347 (2017)

78. Shead, S.: Alphabets deepmind losses soared to 570 million dollar in 2018 (2019-08-18), https://www.forbes.com/sites/samshead/2019/08/07/deepmind-losses-soared-to-570-million-in-2018/#2a5fd25b3504
79. tensorflow user shiviser: Mention that gpu reductions are nondeterministic in docs - issue nr. 2732 - tensorflow/tensorflow (2019-07-05), https://github.com/tensorflow/tensorflow/issues/2732
80. tensorflow user shiviser: Mention that gpu reductions are nondeterministic in docs - issue nr. 2732 - tensorflow/tensorflow (2020-02-02), https://github.com/tensorflow/tensorflow/issues/2732#issuecomment-473720902
81. Silver, D., Huang, A., Maddison, C.J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al.: Mastering the game of go with deep neural networks and tree search. nature **529**(7587), 484 (2016)
82. Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., et al.: Mastering the game of go without human knowledge. Nature **550**(7676), 354 (2017)
83. Skinner, B.F.: Reinforcement today. American Psychologist **13**(3), 94 (1958)
84. Smets, P.: Imperfect information: Imprecision and uncertainty. In: Uncertainty management in information systems, pp. 225–254. Springer (1997)
85. Sprague, T.: On the different possible non-linear arrangements of eight men on a chess-board. Proceedings of the Edinburgh Mathematical Society **8**, 30–43 (1889)
86. Sutton, R.S., McAllester, D.A., Singh, S.P., Mansour, Y.: Policy gradient methods for reinforcement learning with function approximation. In: Advances in neural information processing systems. pp. 1057–1063 (2000)
87. Taylor, M.E., Stone, P.: Transfer learning for reinforcement learning domains: A survey. Journal of Machine Learning Research **10**(Jul), 1633–1685 (2009)
88. Trunda, O., Barták, R.: Using monte carlo tree search to solve planning problems in transportation domains. In: Mexican International Conference on Artificial Intelligence. pp. 435–449. Springer (2013)
89. Van Der Ree, M., Wiering, M.: Reinforcement learning in the game of othello: learning against a fixed opponent and learning from self-play. In: 2013 IEEE Symposium on Adaptive Dynamic Programming And Reinforcement Learning (ADPRL). pp. 108–115. IEEE (2013)
90. Van Der Walt, S., Colbert, S.C., Varoquaux, G.: The numpy array: a structure for efficient numerical computation. Computing in Science & Engineering **13**(2), 22 (2011)
91. Vinyals, O., Babuschkin, I., Czarnecki, W.M., Mathieu, M., Dudzik, A., Chung, J., Choi, D.H., Powell, R., Ewalds, T., Georgiev, P., et al.: Grandmaster level in starcraft ii using multi-agent reinforcement learning. Nature pp. 1–5 (2019)
92. Weng, R.C., Lin, C.J.: A bayesian approximation method for online ranking. Journal of Machine Learning Research **12**(Jan), 267–300 (2011)
93. Wikipedia: First-move advantage in chess (2019-08-18), https://en.wikipedia.org/wiki/First-move_advantage_in_chess
94. Wikipedia: Game tree (2019-08-18), https://en.wikipedia.org/wiki/Game_tree
95. Wikipedia: Tetromino (2019-09-09), https://en.wikipedia.org/wiki/Tetromino
96. Yam, J.Y., Chow, T.W.: A weight initialization method for improving training speed in feedforward neural network. Neurocomputing **30**(1-4), 219–232 (2000)
97. Zermelo, E.: Die berechnung der turnier-ergebnisse als ein maximumproblem der wahrscheinlichkeitsrechnung. Mathematische Zeitschrift **29**(1), 436–460 (1929)

Appendices

A Additional Performance Information

A.1 Desktop Hardware Specifications

The following are the exact specifications of the desktop computer used in most of the experiments:

Part	Specification
CPU	Intel(R) Core(TM) i7-5930K CPU @ 3.50 GHz, 6 Core(s)
GPU	NVIDIA GTX 980 Ti
RAM	32 GB
Storage	256 GB SDD
OS	Arch Linux
Mainboard	ASUS X99-A
Containerisation	Docker

A.2 Server Hardware Specifications

The hyperparameter optimisation (section 6.4) and the RL model (section 6.7) training was done on the *duranium* server offered by LIACS to students of the University of Leiden which has the following specifications:

Part	Specification
CPU	Intel(R) Xeon(R) CPU E5-2650 v3 @ 2.30GHz
GPU	NVIDIA GTX 980 Ti and NVIDIA Titan
RAM	126 GB
Storage	3 TB HDD
OS	CentOS 7
Containerisation	None

A.3 Rust performance optimisations

The following is a complete list of performance optimisation changes to the Rust game implementation. Every entry starts with the average time a game simulation took (see section 5.1.1) followed by a textual description of the code change.

– 12ms - naive JS reimplementation

- 4.5ms - cache possible plays instead of using a function that computes it every time
- 4.5ms - use number instead of string for player identification
- 4.2ms - Determine highest_y once in get_possible_plays and cache result instead of twice without cache in can_place_block
- 1.5ms - use one global shape_cache instead of each game_field recalculating its own
- 1.5ms - introduce redundant constants for existing constants that were casted to usize at runtime
- 1.7ms - (reverted because regression) use i64 instead of i8 for numerical types
- 0.9ms - all allocations of objects and arrays moved to constructor and in the functions use clear() / reset() instead of new allocations
- 0.85ms - only update lowest_y once per newly placed block using update_lowest_for_block
- 0.59ms - incrementally update highest_y, lowest_ys_cmp, gather_positions, update_lowest_for_block and player score
- 0.59ms - target host cpu with export RUSTFLAGS='-Ctarget-cpu=native'

B Hyperparameters

B.1 PPO2 Search

```
def optimise_ppo2(trial):
    return {
        'n_steps': trial.suggest_int('n_steps', 32, 5000),
        'gamma': trial.suggest_loguniform('gamma', 0.6, 0.9999),
        'vf_coef': trial.suggest_loguniform('vf_coef', 0.1, 1),
        'max_grad_norm': trial.suggest_loguniform('max_grad_norm', 0.1, 1)
            ↪ ,
        'learning_rate': trial.suggest_loguniform('learning_rate',
            ↪ 0.000005, 0.05),
        'ent_coef': trial.suggest_loguniform('ent_coef', 1e-20, 0.01),
        'cliprange': trial.suggest_uniform('cliprange', 0.05, 1.9),
        'noptepochs': trial.suggest_int('noptepochs', 1, 48),
        'lam': trial.suggest_uniform('lam', 0.6, 1.2)
    }
```

Fig. 28: The Optuna settings used to search the hyperparameterspace of PPO2.

B.2 RL-Selfplay Hyperparameters

Parameter	Value
gamma	0.99
n_steps	128
ent_coef	0.01
learning_rate	0.00025
vf_coef	0.5
max_grad_norm	0.5
lam	0.95
nminibatches	4
noptepochs	4
cliprange	0.2

B.3 RL-Heuristic Hyperparameters

Parameter	Value
n_steps	1912
gamma	0.893772595247008
vf_coef	0.124083762089201
max_grad_norm	0.790427907967279
learning_rate	0.0106800972074722
ent_coef	3.11268993114624e-07
cliprange	1.13010015406973
noptepochs	7
lam	0.80055096907225

B.4 MCTS Search parameters

Parameter	Value
Thought time	1000ms
Simulation amount	50
Exploration constant	2.0
Lookahead limit for default policy playouts	None
RAVE Beta parameter	1000

B.5 Unusable RL Algorithms

Some *stable-baselines* algorithms should have been usable with the chosen action/observation spaces according to the documentation. However, upon trying

them out, some did not. The not working algorithms with the respective error message are ACER (Figure 29), ACKTR (Figure 30), DDPG and SAC (Figure 31).

```
Traceback (most recent call last):
  File "libs\optuna\study.py", line 468, in _run_trial
    result = func(trial)
  File "acer_tuner.py", line 44, in optimise_agent
    model.learn(int(2e4), seed=seed)
  File "libs\stable_baselines\acer\acer_simple.py", line 474, in learn
    runner = _Runner(env=self.env, model=self, n_steps=self.n_steps)
  File "libs\stable_baselines\acer\acer_simple.py", line 596, in __init__
    obs_height, obs_width, obs_num_channels = env.observation_space.shape
ValueError: not enough values to unpack (expected 3, got 2)
```

Fig. 29: The error message produced when trying to use ACER.

```
Traceback (most recent call last):
  File "libs\optuna\study.py", line 468, in _run_trial
    result = func(trial)
  File "acktr_tuner.py", line 35, in optimise_agent
    model = ACKTR("MlpPolicy", env, nprocs=1,verbose=0, **model_params)
  File "libs\stable_baselines\acktr\acktr_disc.py", line 103, in __init__
    self.setup_model()
  File "libs\stable_baselines\acktr\acktr_disc.py", line 118, in
      ↪ setup_model
    raise NotImplementedError("WIP: ACKTR does not support Continuous
        ↪ actions yet.")
```

Fig. 30: The error message produced when trying to use ACKTR.

```
Traceback (most recent call last):
  File "libs\optuna\study.py", line 468, in _run_trial
    result = func(trial)
  File "ddpg_tuner.py", line 39, in optimise_agent
    model = DDPG("MlpPolicy", env, verbose=0, observation_range
        ↪ =(-126,126),
    **model_params)
  File "libs\stable_baselines\ddpg\ddpg.py", line 298, in __init__
    self.setup_model()
  File "libs\stable_baselines\ddpg\ddpg.py", line 336, in setup_model
    **self.policy_kwargs)
  File "libs\stable_baselines\ddpg\policies.py", line 239, in __init__
    feature_extraction="mlp", **_kwargs)
  File "libs\stable_baselines\ddpg\policies.py", line 111, in __init__
    scale=(feature_extraction == "cnn"))
  File "libs\stable_baselines\ddpg\policies.py", line 26, in __init__
    assert (np.abs(ac_space.low) == ac_space.high).all(),
    "Error: the action space low and high must be symmetric"
```

Fig. 31: The error message produced when trying to use DDPG and SAC (for SAC the code paths are different but it gave the same *ValueError*.

C Code Changes to Open Source Projects

C.1 Reproducibility

For a reproducibility experiment (section 6.3), we modified code in *stable-baselines*. The git diff between the changes and v2.6.0 is shown below.

```
diff --git a/stable_baselines/common/base_class.py b/
    ↪ stable_baselines/common/base_class.py
index a0637d8..86c822d 100644
--- a/stable_baselines/common/base_class.py
+++ b/stable_baselines/common/base_class.py
@@ -69,6 +69,28 @@ class BaseRLModel(ABC):
                                        " environment.")
                self.n_envs = 1

+ def set_random_seed(self, seed):
+ """
+ :param seed: (int) Seed for the pseudo-random generators. If
    ↪ None,
+ do not change the seeds.
+ """
+ # Ignore if the seed is None
+ if seed is None:
```

```
+ return
+ # Seed python, numpy and tf random generator
+ set_global_seeds(seed)
+ if self.env is not None:
+ if isinstance(self.env, VecEnv):
+ # Use a different seed for each env
+ for idx in range(self.env.num_envs):
+ self.env.env_method("seed", seed + idx)
+ else:
+ self.env.seed(seed)
+ # Seed the action space
+ # useful when selecting random actions
+ self.env.action_space.seed(seed)
+ self.action_space.seed(seed)
+
     def get_env(self):
         """
         returns the current environment (can be None if not
             ↪ defined)
diff --git a/stable_baselines/ppo2/ppo2.py b/stable_baselines/ppo2
    ↪ /ppo2.py
index 330f585..cef6dcb 100644
--- a/stable_baselines/ppo2/ppo2.py
+++ b/stable_baselines/ppo2/ppo2.py
@@ -50,7 +50,7 @@ class PPO2(ActorCriticRLModel):
     def __init__(self, policy, env, gamma=0.99, n_steps=128,
         ↪ ent_coef=0.01,
     learning_rate=2.5e-4, vf_coef=0.5, max_grad_norm=0.5, lam
         ↪ =0.95,
     nminibatches=4, noptepochs=4, cliprange=0.2, cliprange_vf=
         ↪ None, verbose=0,
     tensorboard_log=None, _init_setup_model=True, policy_kwargs=
         ↪ None,
- full_tensorboard_log=False):
+ full_tensorboard_log=False,seed=None):

         super(PPO2, self).__init__(policy=policy, env=env, verbose
             ↪ =verbose,
         requires_vec_env=True, _init_setup_model=_init_setup_model
             ↪ ,
         policy_kwargs=policy_kwargs)
@@ -95,6 +95,7 @@ class PPO2(ActorCriticRLModel):
         self.n_batch = None
         self.summary = None
         self.episode_reward = None
```

```
+ self.seed = seed

        if _init_setup_model:
            self.setup_model()
@@ -113,12 +114,13 @@ class PPO2(ActorCriticRLModel):

            self.n_batch = self.n_envs * self.n_steps

- n_cpu = multiprocessing.cpu_count()
- if sys.platform == 'darwin':
- n_cpu //= 2
+ n_cpu = 1 #multiprocessing.cpu_count()
+ # if sys.platform == 'darwin':
+ # n_cpu //= 2

            self.graph = tf.Graph()
            with self.graph.as_default():
+ self.set_random_seed(self.seed)
                self.sess = tf_util.make_session(
                num_cpu=n_cpu, graph=self.graph)

                n_batch_step = None
```

C.2 Self Play

For a self play experiment (section 6.7), we modified code in *stable-baselines*. The git diff between the changes and v2.6.0 is shown below.

```
diff --git a/stable_baselines/ppo2/ppo2.py b/stable_baselines/ppo2
    ↪ /ppo2.py
index cef6dcb..d105f67 100644
--- a/stable_baselines/ppo2/ppo2.py
+++ b/stable_baselines/ppo2/ppo2.py
@@ -50,7 +50,7 @@ class PPO2(ActorCriticRLModel):
    def __init__(self, policy, env, gamma=0.99, n_steps=128,
        ↪ ent_coef=0.01,
    learning_rate=2.5e-4, vf_coef=0.5, max_grad_norm=0.5, lam
        ↪ =0.95,
    nminibatches=4, noptepochs=4, cliprange=0.2, cliprange_vf=
        ↪ None, verbose=0,
    tensorboard_log=None, _init_setup_model=True, policy_kwargs=
        ↪ None,
- full_tensorboard_log=False,seed=None):
+ full_tensorboard_log=False,seed=None, self_play=False):
```

```
        super(PPO2, self).__init__(policy=policy, env=env, verbose
            ↪ =verbose,
        requires_vec_env=True, _init_setup_model=_init_setup_model
            ↪ ,
        policy_kwargs=policy_kwargs)
@@ -95,6 +95,8 @@ class PPO2(ActorCriticRLModel):
        self.n_batch = None
        self.summary = None
        self.episode_reward = None
+ self.self_play = self_play
+ self.opponent = None
        self.seed = seed

        if _init_setup_model:
@@ -317,7 +319,7 @@ class PPO2(ActorCriticRLModel):
                as writer:
            self._setup_learn(seed)

- runner = Runner(env=self.env, model=self, n_steps=self.n_steps,
- gamma=self.gamma, lam=self.lam)
+ runner = Runner(env=self.env, model=self, n_steps=self.n_steps,
+ gamma=self.gamma, lam=self.lam, self_play=self.self_play,
+ opponent=self.opponent)
          self.episode_reward = np.zeros((self.n_envs,))

           ep_info_buf = deque(maxlen=100)
@@ -431,7 +433,7 @@ class PPO2(ActorCriticRLModel):


 class Runner(AbstractEnvRunner):
- def __init__(self, *, env, model, n_steps, gamma, lam):
+ def __init__(self, *, env, model, n_steps, gamma, lam,
+ self_play, opponent=None):
        """
        A runner to learn the policy of an environment for a model

@@ -444,6 +446,8 @@ class Runner(AbstractEnvRunner):
        super().__init__(env=env, model=model, n_steps=n_steps)
        self.lam = lam
        self.gamma = gamma
+ self.self_play = self_play
+ self.opponent = opponent

    def run(self):
        """
```

```
@@ -463,8 +467,14 @@ class Runner(AbstractEnvRunner):
        mb_obs, mb_rewards, mb_actions, mb_values, mb_dones,
            ↪ mb_neglogpacs =
        [], [], [], [], [], []
        mb_states = self.states
        ep_infos = []
+ env_to_change =
+  self.env.envs[0] if self.env.envs is not None else self.env
+ env_to_change.self_play = self.self_play
+ env_to_change.self_play_is_second_player = False
        for _ in range(self.n_steps):
- actions, values, self.states, neglogpacs =
-   self.model.step(self.obs, self.states, self.dones)
+ model_to_use = self.model
+ if self.opponent is not None and
+   env_to_change.self_play_is_second_player:
+ model_to_use = self.opponent
+ actions, values, self.states, neglogpacs =
+   model_to_use.step(self.obs, self.states, self.dones)
            mb_obs.append(self.obs.copy())
            mb_actions.append(actions)
            mb_values.append(values)
@@ -480,6 +490,10 @@ class Runner(AbstractEnvRunner):
                if maybe_ep_info is not None:
                    ep_infos.append(maybe_ep_info)
            mb_rewards.append(rewards)
+ if not self.dones:
+ env_to_change.self_play_is_second_player =
+    not env_to_change.self_play_is_second_player
+ else:
+ env_to_change.self_play_is_second_player = False
        # batch of steps to batch of rollouts
        mb_obs = np.asarray(mb_obs, dtype=self.obs.dtype)
        mb_rewards = np.asarray(mb_rewards, dtype=np.float32)
```

C.3 Self Play nobleed

For a self play experiment (section 6.7), we modified code in *stable-baselines*. This is the version that also corrects the problem with *bleeding* over information from model b to model a. The git diff between the changes and v2.6.0 is shown below.

```
diff --git a/stable_baselines/ppo2/ppo2.py b/stable_baselines/ppo2
    ↪ /ppo2.py
index cef6dcb..121f6f7 100644
```

```
--- a/stable_baselines/ppo2/ppo2.py
+++ b/stable_baselines/ppo2/ppo2.py
@@ -50,7 +50,7 @@ class PPO2(ActorCriticRLModel):
    def __init__(self, policy, env, gamma=0.99, n_steps=128,
        ↪ ent_coef=0.01,
    learning_rate=2.5e-4, vf_coef=0.5, max_grad_norm=0.5, lam
        ↪ =0.95,
    nminibatches=4, noptepochs=4, cliprange=0.2, cliprange_vf=
        ↪ None, verbose=0,
    tensorboard_log=None, _init_setup_model=True, policy_kwargs=
        ↪ None,
- full_tensorboard_log=False,seed=None):
+ full_tensorboard_log=False,seed=None, self_play=False):

        super(PPO2, self).__init__(policy=policy, env=env, verbose
            ↪ =verbose,
        requires_vec_env=True, _init_setup_model=_init_setup_model
            ↪ ,
        policy_kwargs=policy_kwargs)
@@ -95,6 +95,8 @@ class PPO2(ActorCriticRLModel):
        self.n_batch = None
        self.summary = None
        self.episode_reward = None
+ self.self_play = self_play
+ self.opponent = None
        self.seed = seed

        if _init_setup_model:
@@ -317,7 +319,7 @@ class PPO2(ActorCriticRLModel):
                as writer:
            self._setup_learn(seed)

- runner = Runner(env=self.env, model=self, n_steps=self.n_steps,
- gamma=self.gamma, lam=self.lam)
+ runner = Runner(env=self.env, model=self, n_steps=self.n_steps,
+ gamma=self.gamma, lam=self.lam, self_play=self.self_play,
+ opponent=self.opponent)
            self.episode_reward = np.zeros((self.n_envs,))

            ep_info_buf = deque(maxlen=100)
@@ -431,7 +433,7 @@ class PPO2(ActorCriticRLModel):


 class Runner(AbstractEnvRunner):
- def __init__(self, *, env, model, n_steps, gamma, lam):
```

```
+ def __init__(self, *, env, model, n_steps, gamma, lam,
+ self_play, opponent=None):
        """
        A runner to learn the policy of an environment for a model

@@ -444,6 +446,8 @@ class Runner(AbstractEnvRunner):
        super().__init__(env=env, model=model, n_steps=n_steps)
        self.lam = lam
        self.gamma = gamma
+ self.self_play = self_play
+ self.opponent = opponent

    def run(self):
        """
@@ -463,23 +467,38 @@ class Runner(AbstractEnvRunner):
        mb_obs, mb_rewards, mb_actions, mb_values, mb_dones,
            ↪ mb_neglogpacs =
        [], [], [], [], [], []
        mb_states = self.states
        ep_infos = []
- for _ in range(self.n_steps):
+ env_to_change =
+  self.env.envs[0] if self.env.envs is not None else self.env
+ env_to_change.self_play = self.self_play
+ env_to_change.self_play_is_second_player = False
+ has_opponent = self.opponent is not None
+ n_steps_to_use = self.n_steps * 2 if has_opponent else self.
    ↪ n_steps
+ for _ in range(n_steps_to_use):
+ model_to_use = self.model
+ is_opponents_turn =
+   has_opponent and env_to_change.self_play_is_second_player
+ if is_opponents_turn:
+ model_to_use = self.opponent
            actions, values, self.states, neglogpacs =
            self.model.step(self.obs, self.states, self.dones)
- mb_obs.append(self.obs.copy())
- mb_actions.append(actions)
- mb_values.append(values)
- mb_neglogpacs.append(neglogpacs)
- mb_dones.append(self.dones)
+ if not is_opponents_turn:
+ mb_obs.append(self.obs.copy())
+ mb_actions.append(actions)
+ mb_values.append(values)
```

```
+ mb_neglogpacs.append(neglogpacs)
+ mb_dones.append(self.dones)
            clipped_actions = actions
            # Clip the actions to avoid out of bound error
            if isinstance(self.env.action_space, gym.spaces.Box):
                clipped_actions = np.clip(actions, self.env.
                    ↪ action_space.low,
                self.env.action_space.high)
            self.obs[:], rewards, self.dones, infos =
            self.env.step(clipped_actions)
- for info in infos:
- maybe_ep_info = info.get('episode')
- if maybe_ep_info is not None:
- ep_infos.append(maybe_ep_info)
- mb_rewards.append(rewards)
+ if not is_opponents_turn:
+ for info in infos:
+ maybe_ep_info = info.get('episode')
+ if maybe_ep_info is not None:
+ ep_infos.append(maybe_ep_info)
+ mb_rewards.append(rewards)
+ if not self.dones:
+ env_to_change.self_play_is_second_player =
+    not env_to_change.self_play_is_second_player
+ else:
+ env_to_change.self_play_is_second_player = False
        # batch of steps to batch of rollouts
        mb_obs = np.asarray(mb_obs, dtype=self.obs.dtype)
        mb_rewards = np.asarray(mb_rewards, dtype=np.float32)
```

D Game Field Representations

D.1 Numerical Field Representation

This is a numerical array representation of the current state of a Tetris Link game board. Every player has two numbers because every block consists of one special highlighted square and 3 regular squares. The highlight square is required to be able to count the number of blocks and be able to distinguish individual blocks from each other when they form a group.

```
0020440220
1223401200
3444000440
2000003400
1200000220
```

```
2000001200
3444004400
2000003400
1200002000
2000001200
4000002440
3400003420
4000001220
2200000440
1200003400
3444000220
0220001200
1200004400
4400003400
3400001222
```

D.2 Reinforcement Learning Field Representation

This is a numerical array representation of the current state of a Tetris Link game board including game state information such as score, available pieces and possible moves. Every player has two numbers because every block consists of one special highlight square and 3 regular squares. The highlight square is required to be able to count the amount of blocks and be able to distinguish individual blocks from each other when they form a group. After 20 rows the additional info begins. For details see section 5.2.

```
0000000000
0000000000
0000000000
0000000000
0000000000
0000000000
0000000000
0000000000
0000000000
0000000000
0022000000
0012000000
0122200400
1222004400
0122204400
0400004300
0400003400
0403444404
0301222404
```

```
1222000334
-85-1-1-1-1-1-1-1-1
0455505554
-1-1-1-1-1-1-1-1-1-1
-1-1-1-1-1-1-1-1-1-1
-1-1-1-1-1-1-1-1-1-1
111111111-1
-11-1-1111111
1111111111
1111111111
111111-1-1-11
11-1111-1111
1111111111
1111111111
11-1111-1-11-1
1111111111
1111111111
1111111111
1111111111
1111111111
1111111111
1111-11-11-11
-11-1-1-1-1-1-1-1-1-1
```

D.3 JSON Gamelog

Below is an example of the game log format used for interoperability between the JS and Rust game implementation. This specific log produces the board state seen in Figure 7b.

```
{
  "log": [{
    "PayloadRolled": {
      "from": 1,
      "block": 3
    }
  }, {
    "PayloadConsidering": {
      "play_index": 98
    }
  }, {
    "PayloadPlaced": {
      "from": 1,
      "block": 4,
      "orientation": 4,
```

```
    "x": 0,
    "y": 0
  }
}, {
  "PayloadRolled": {
    "from": 0,
    "block": 0
  }
}, {
  "PayloadConsidering": {
    "play_index": 68
  }
}, {
  "PayloadPlaced": {
    "from": 0,
    "block": 3,
    "orientation": 2,
    "x": 2,
    "y": 1
  }
}, {
  "PayloadRolled": {
    "from": 1,
    "block": 0
  }
}, {
  "PayloadConsidering": {
    "play_index": 118
  }
}, {
  "PayloadPlaced": {
    "from": 1,
    "block": 4,
    "orientation": 0,
    "x": 3,
    "y": 0
  }
}, {
  "PayloadRolled": {
    "from": 0,
    "block": 0
  }
}, {
  "PayloadConsidering": {
    "play_index": 70
```

```
    }
  }, {
    "PayloadPlaced": {
      "from": 0,
      "block": 3,
      "orientation": 0,
      "x": 3,
      "y": 1
    }
  }, {
    "PayloadRolled": {
      "from": 1,
      "block": 0
    }
  }, {
    "PayloadConsidering": {
      "play_index": 19
    }
  }, {
    "PayloadPlaced": {
      "from": 1,
      "block": 1,
      "orientation": 0,
      "x": 2,
      "y": 2
    }
  }, {
    "PayloadRolled": {
      "from": 0,
      "block": 0
    }
  }, {
    "PayloadConsidering": {
      "play_index": 9
    }
  }, {
    "PayloadPlaced": {
      "from": 0,
      "block": 0,
      "orientation": 1,
      "x": 4,
      "y": 3
    }
  }, {
    "PayloadRolled": {
```

```
      "from": 1,
      "block": 0
    }
  }, {
    "PayloadConsidering": {
      "play_index": 25
    }
  }, {
    "PayloadPlaced": {
      "from": 1,
      "block": 1,
      "orientation": 0,
      "x": 8,
      "y": 0
    }
  }, {
    "PayloadRolled": {
      "from": 0,
      "block": 0
    }
  }, {
    "PayloadConsidering": {
      "play_index": 12
    }
  }, {
    "PayloadPlaced": {
      "from": 0,
      "block": 0,
      "orientation": 0,
      "x": 6,
      "y": 2
    }
  }, {
    "PayloadRolled": {
      "from": 1,
      "block": 0
    }
  }, {
    "PayloadConsidering": {
      "play_index": 0
    }
  }]
}
```