Bachelor Thesis

Reinforcement Learning for 3-player Chinese Checkers

David Schulte

March 19, 2021

Mathematisches Institut
Mathematisch-Naturwissenschaftliche Fakultät
Universität zu Köln

Supervision: Prof. Dr.-Ing. Gregor Gassner

Acknowledgements

I want to thank Prof. Dr.-Ing. Gregor Gassner and the Numerical Simulation research group at University of Cologne for supervising this project, especially Moritz Schily, who always found time to answer my questions.

I also want to thank my family and friends for supporting me throughout the whole bachelor's program.

Contents

1. Introduction

In recent years, the field of machine learning has attracted a lot of attention from researchers as well as the public. Advancements in computational power and the rapidly growing amount of data lead to new possibilities in software design and problem solving. There are programs that detect patterns in visual, aural and abstract data. The quality of automatic translation between languages has increased. However, one field in machine learning that particularly interests me, is Reinforcement Learning. In Reinforcement Learning agents, entities in the program, carry out a task and gradually improve by receiving feedback of their performance. In the recent past, board games have been used to test and demonstrate state-of-the-art training techniques.

The idea of a machine that could defeat humans in a board game, especially in chess, has sparked people's curiosity for several centuries.
In the year 1770 Wolfgang von Kempelen invented a machine, he called the "Chess Turk". It consisted of a puppet attached to a wooden desk with a chess board on top of it. The puppet played the game by human-like movements of its arm. Claiming that this machine could defeat even strong players, he toured across Europe. The machine was thought of as a mechanical masterpiece and defeated many contenders, including Benjamin Franklin and Napoleon Bonaparte. Only after the machine's destruction, the hoax was revealed. During the games influential chess players from that time hid inside the wooden desk and controlled the puppet's arms.
In the 20th century, the idea of chess-playing machines and algorithms was picked up again. Great computer scientists and mathematicians, like Konrad Zuse, Alan Turing and Claude Shannon worked on chess programs and thought about ways to solve the game. However, lacking sufficient hardware, they were not able to implement their ideas.
Computer chess is also a common theme in science fiction, like in the 1968 movie "2001: A Space Odyssey", where the protagonist, an astronaut, loses a game of chess against his spaceship's board computer HAL 9000 - a scene that convinced the audience of its super-human intelligence. In the year 1997, these fantasies became reality, as the chess program Deep Blue, made by IBM, beat the former world champion, Garry Kasparov. The resource of computational power made the already established techniques applicable.
The focus of machine learning for games shifted towards the game of Go. Its complexity and popularity made it the next target for many researchers worldwide. In March 2016, the program AlphaGo[7], invented by the research company Google DeepMind, won four out of five matches against 18-time world champion Lee Sedol. In contrast to Deep Blue, it made use of Reinforcement Learning. In December 2018, DeepMind published a paper that describes and evaluates AlphaZero.[8] It is built upon AlphaGo but was generalized in such a way that it is applicable to not only Go, but also other two-player games with perfect information. It was also tested on Chess and Shogi. Another difference to its inferior predecessor is that the program learns these games by itself, given only the rules but no examples of games played by humans.

The research in this field is not only conducted to gain insights into board games. Rather, these games serve as a suitable training ground in Machine Learning. In contrast to most other real-life problems, they can be easily defined as a closed system and allow clear evaluation. Artificial neural networks as well as training methods can be tested on them.
Reinforcement Learning can enable automation of tasks that are hard to formulate. Those are commonly faced in robotics, for example.
Because of their unbiased approach, agents in Reinforcement Learning also have the potential, to find solutions to problems that researchers were not able to solve as yet.

The purpose of this thesis is to investigate the possibilities and issues of Reinforcement Learning applied to multiplayer-games and games with recurring states. As domain I chose a game I know from my childhood - Chinese Checkers. The program trains a neural network through self-play and improves its strategies over time. The construct is heavily inspired by AlphaZero. The code is built upon an open-source project[4], used in the paper[6], in which authors applied it to the game Othello. It was significantly modified to fit a 3-player game with possible repetitions of game states, using a different neural network and a modified tree search. I also enabled it to play several games in self-play simultaneously.

2. Chinese Checkers

Chinese Checkers originates from the board game Halma and dates back to 1892. While it is suited for 2-4 players, this project focuses solely on the 3-player variant. To make the game suitable for this project, the board size was reduced from the standard board with 121 fields to a board with 37 fields, and the rules were slightly modified.

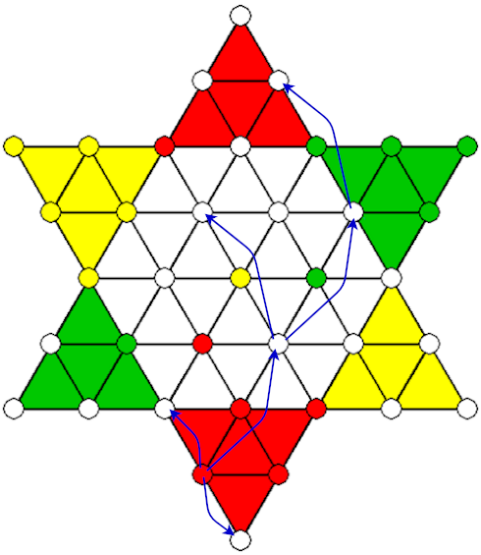


Figure 2.0.0: Example board state after two moves each
The possible moves of one specific piece are illustrated.

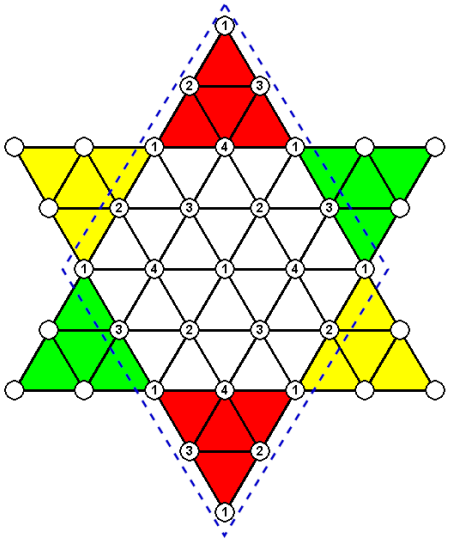


Figure 2.0.0: Accessible fields for player one
Fields with the same number lie on the same jumping grid. A piece that is moved through a jumping sequence lands within the grid it started from.

2.1. Rules

Every player starts with 6 pieces, placed in one corner of the star-shaped board. These corners are called start zones and the corners opposite of them are their respective end zones. Player 1 has red pieces. Player 2 has yellow pieces. Player 3 has green pieces.
Chinese checkers is a race game with a 1^{st}, a 2^{nd} and a 3^{rd} place. The 1^{st} place takes whoever manages to move all his pieces from his start zone into his end zone before the others. The 2^{nd} place takes, who has all fields in his end zone occupied, after the 1^{st} place is decided. Here, it does not matter, which player's pieces occupy the end zone. We call this the second-winner rule. The remaining player takes the 3^{rd} place. We award the 1^{st} place a score of 3, the 2^{nd} a score of 1 and the 3^{rd} place a score of 0.
Players take turns clockwise after each move. A move can be either a short hop or a jump sequence.
With a short hop, a piece can be moved to an adjacent space, provided that it is vacant.
A piece can jump over another piece, provided that the field behind it is not occupied. Jumps can be chained together. As long as another jump by the same piece is possible, the player is free to execute it. Paths that are accessible with multiple consecutive jumps are called ladders. Short hops and jumps cannot be combined in one move.
A player cannot move his pieces onto fields in another player's zone, except for those that inter-

sect with the neutral center of the board.

Players can use offensive as well as defensive tactics. They can move their pieces forward and build ladders to prepare for further hops. On the other hand, one can also block enemy pieces and destroy ladders by occupying the ladder fields.

2.2. Mathematical Description

This subsection will introduce a mathematical description of the game. Even though a more compact description is possible, this one is used to stay close to the implementation.
$S \subset \{0, 1, 2, 3, 4\}^{9\times 9}$ is the set of all possible board states. In a board state $s \in S$ the value of $s_{i,j}$ describes the state of a field in the respective position. 0 denotes an empty field. 1,2 and 3 denote a field occupied by the respective player. 4 denotes fields that are unreachable and only exist to make an array representation possible.
We call s a terminal state, if it represents an ended the game. The board is being represented by a 9x9 array. To achieve this, the board was distorted.

4	4	4	4	4	4	0	4	4
4	4	4	4	4	0	0	4	4
4	4	2	2	2	0	3	3	3
4	4	2	2	0	0	3	3	4
4	4	2	0	0	0	3	4	4
4	0	0	0	0	0	0	4	4
0	0	1	1	1	0	0	4	4
4	4	1	1	4	4	4	4	4
4	4	1	4	4	4	4	4	4

Figure 2.2.0: Board representation
This array represents the board at the start of the game.

	Upper left	Upper right
Left	Center	Right
Lower left	Lower right	

Figure 2.2.0: Directions

$A = \{1, 2, \ldots, 320\}$ is the set of all actions.
Intuitively, we would describe actions as functions. The reason, why they are represented by numbers, is that this enables us to use them as indices. This greatly simplifies notation and is close to the implementation. The algorithm requires a vector with each move in the game corresponding to an entry. This action vector can be split into two parts. The first part represents direct steps, and the second one represents (consecutive) jumps.

Every of the 25 accessible fields gets combined with 6 step directions. This results in 150 moves. To encode jumps, the board is divided into four grids of different sizes. A sequence of jumps that starts in a grid also has to end in it. Combinations of fields in the same grid are compiled. With grid sizes 9, 6, 6 and 4 this leads to 169 jump sequences.
Lastly, there is one passive move that does not change the board. It gets chosen if and only if

there are no other possible moves available. This is the case, when every piece of a player is blocked, or when a player has won and is waiting for his opponents to finish the game. Thus, the action vector contains 320 entries. $|A| = 320$.

It should be noted that the action space contains moves that are illegal in every board state. Those are the direct moves leading outside of the board (22) or into the forbidden parts of opponents' zones (16). Also, jumping moves that lead to the starting point are included (25). Direct moves (8) as well as jumping sequences that lead out of the end zones (31) are not allowed. Taking into account the intersection of these moves, this sums up to 102 moves or about $31,9\%$ of the total action space. Even though these impossible moves enlarge the action space, they are included, because it greatly simplifies encoding and decoding.

$V = [0,3]^3$ is the set of state values. A state value describes the expected scores of each player at a board state.
$P = \{1,2,3\}$ is the set of players.
$\alpha : S \times P \to \mathcal{P}(A)$ assigns every board state and current player a set of possible actions.
$m_s : \{(s,a,p)|(s,p) \in S \times P \wedge a \in \alpha(s,p)\} \to S$ returns the next board state, given a current board state, an action and the player that executes it.
$m_p : \{(s,a,p)|(s,p) \in S \times P \wedge a \in \alpha(s,p)\} \to P$ returns the next player, given a current board state, a action and the player that executes it.
$m : \{(s,a,p)|(s,p) \in S \times P \wedge a \in \alpha(s,p)\} \to S \times P$ returns the next board state, given a board state and an action by a specific player.
$g : V \times P \to \mathbb{R}$ returns the expected score of player p, given the state value v. In this case, it returns the p^{th} entry of v.

3. Tree search

In this section, we will introduce two decision-making algorithms used for deterministic games with perfect information. At first, we will introduce the Minimax algorithm to convey the general methodology of tree search algorithms for decision making. Afterwards, we will take a look at the Monte-Carlo tree search, which is used in this project.
Before describing the algorithms, we introduce the evaluation function ϕ.
$\phi : S \times P \to V$ assigns a state value to a game situation with board state s and current player p. As long as a game is not solved, it is not easy to find a suitable evaluation function. ϕ uses the neural network, as we will explain in the next chapters.

Minimax and the Monte-Carlo tree search are used for the same purpose. Given a board state s and the current player p, they are applied to determine the best action to take. That is the action that is expected to lead the game towards the terminal game state s^* with state value v^* that maximizes $g(v^*, p)$.

When ranking moves in a game, one would not only take into account the situations directly resulting from them, but also possible reactions by opponents. The presented algorithms also do so. They simulate several possible game developments and estimate the behavior of other players. Before deciding on which action to take, they build a game tree. The tree representation of the game is as follow:
A board state and current player $(s, p) \in S \times P$ are denoted by a tree node. The two nodes (s, p) and (s', p') can be connected by an edge, if there is an action $a \in \alpha(s, p)$ such that $m_s(s, a, p) = s'$ and $m_p(s, a, p) = p'$.
The root node corresponds to the current game situation.

For now, we will assume that no game loops are possible, meaning that every board state can only occur once in one game. Although this does not hold true for Chinese Checkers, this assumption is required for the game to be represented as a tree. We will further discuss this issue in subsection 3.3

3.1. Minimax

To illustrate the principles of Minimax, we will demonstrate it on a simpler game than Chinese Checkers. The name Minimax originates from the fact that the algorithm is generally applied to zero-sum games with two players, of which one tries to maximize the state value, while the other one tries to minimize it.
$\tilde{V} = \mathbb{Z} \cup \{-\infty, +\infty\}$
$\tilde{P} = \{1, 2\}$

$$\tilde{g}(v, p) = \begin{cases} v, & \text{if } p = 1 \\ -v, & \text{otherwise} \end{cases} \tag{3.1.1}$$

While this assumption enables an intuitive explanation, it should be noted that it can also be applied to an arbitrary amount of players as well as different evaluation functions and value sets. Minimax builds a game tree that contains every possible node down to a maximal depth, which is specified by the user. Since both players are taking turns, the decision-maker alternates between

each layer of the tree.
Leaf notes are in the deepest layer of the tree or represent terminal game states. We evaluate each of these leaf nodes (s, p). It gets assigned the state value $V_{s,p} := \phi(s)$.
These values are being back-propagated. Knowing that depending on the tree layer the current decision maker will either maximize or minimize the following state value, his action is deducible. Therefore the values of internal nodes are set to be either the maximum or minimum of the values assigned to its children.
Knowing the opponent's policy, a player can optimize not only the state value after his next move but also several moves in advance.

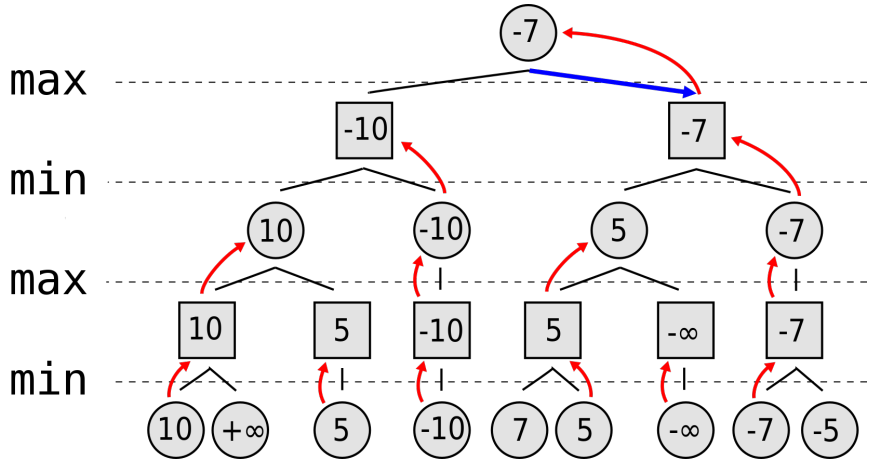


Figure 3.1.2: Minimax Tree
In this tree, both players have two available actions for every state. The state values get back-propagated in a Depth First Search. The result is that the max-player will choose the action corresponding to the right edge of the root node. Figure adapted from figure [5].

Algorithm 1 Minimax

function MINIMAX$(s, p, depth, depth_{max})$
  if s is a terminal state **or** $depth = depth_{max}$ **then**
    $v \leftarrow \phi(s, p)$
    return $v, 0$
  $g_{best} \leftarrow -\infty$
  $a_{best} \leftarrow 0$
  for each $a \in \alpha(s, p)$ **do**
    $s' \leftarrow m_s(s, a, p)$
    $p' \leftarrow m_p(s, a, p)$
    $v', \_ \leftarrow$ MINIMAX$(s', p', depth + 1, depth_{max})$
    $g \leftarrow g(v', p')$
    if $g > g_{best}$ **then**
      $g_{best} \leftarrow g$
      $v_{best} \leftarrow v'$
      $a_{best} \leftarrow a$
  return v_{best}, a_{best}

We call the function with the parameter $depth = 0$. The results are the suggested action a_{best}

and the estimated state value v_{best} after $depth_{max}$ actions.

The problem with Minimax algorithm is that the branching factor of its tree becomes fairly large, depending on the action space of its application. This makes the tree grow exponentially with every tree layer. The amount of tree nodes N satisfies:

$$N \leq \sum_{i=0}^{depth_{max}} |A|^i = \frac{|A|^{depth_{max}} - 1}{|A| - 1}, \qquad \text{if } |A| > 1 \tag{3.1.3}$$

This makes Minimax unsuitable for our problem.

3.2. Monte-Carlo tree search

The Monte-Carlo tree search (MCTS) algorithm is designed on the basis of Minimax. In contrast to Minimax, where every node down to a specified depth gets visited, it gradually builds an asymmetric tree. Every iteration of the search expands the tree by one leaf node, possibly turning previous leaf nodes into internal nodes. It heuristically chooses promising paths to explore, disregarding the others.
Because this enables the algorithm to traverse deeper into the tree without exponential growth, it is especially useful for applications with large action spaces.
There are several variants of MCTS. The following explanations refer to the variant used in this project.

The procedure can be summarized into three steps:

Selection

Starting at the root node, a path navigating through the tree is selected until a leaf node is reached.
$N_{s,a,p}$ is the number of times action a was chosen by player p in state s.
$N_{s,p}$ is the number of times that player p chose an action in state s. It is equal to the sum of values $N_{s,a,p}$ over all $a \in A$.
c_{puct} is the exploration parameter. It will be put into context in the next paragraph.
$P_{s,a,p} \in [0, 1]$ is a pre-evaluation of the action a in state s for player p. This value assigns the action a score between 0 and 1. The sum of all P-values of a node (s, p) is equal to 1. The P-values of a node (s, p) can be interpreted as a probability distribution of the move choices of player p in state s.
$Q_{s,a,p} \in [0, 3]$ is a evaluation of the profitability of action a for player p in state s. It is initialized as 0.
$U_{s,a,p}$ is the upper confidence bound of action a at state s. It describes, how important a exploration of the sub-tree, belonging to action a, is.

$$U_{s,a,p} = \overbrace{Q_{s,a,p}}^{\text{Exploitation}} + c_{puct} \cdot \overbrace{P_{s,a,p} \cdot \frac{\sqrt{N_{s,p}}}{1 + N_{s,a,p}}}^{\text{Exploration}} \tag{3.2.4}$$

From all possible actions, the one maximizing U is chosen.

$$a^* = \underset{a \in \alpha(s,p)}{\arg\max}\, U_{s,a,p} \tag{3.2.5}$$

Expansion

The leaf node (s, p) gets evaluated by the state evaluation function.

$$V_{s,p} = \phi(s, p) \tag{3.2.6}$$

$V_{s,p}$ is a state value for the node that corresponds to state s and player p. Just like in Minimax, these values are determined by ϕ in the leaf nodes and by the values of child nodes in the internal nodes.

Back-propagation

The values of Q get back-propagated up the tree until the root node is reached.

$$V_{s,p} = \begin{cases} \phi(s,p), & \text{if } N_{s,p} = 0 \\ \frac{\sum_{a \in \alpha(s,p)} V_{m(s,a,p)} \cdot N_{s,a,p}}{N_{s,p}}, & \text{otherwise} \end{cases} \tag{3.2.7}$$

$$Q_{s,a,p} = \begin{cases} g(V_{m(s,a,p)}, p), & \text{if } N_{s,a,p} > 0 \\ 0, & \text{otherwise} \end{cases} \tag{3.2.8}$$

The pre-evaluation values $P_{s,a,p}$ are obtained with the help of the neural network. First, we evaluate the function ρ in (s, p), resulting in the vector $\pi \in [0, 1]^{|A|}$.

$$\rho : S \times P \mapsto [0, 1]^{|A|} \tag{3.2.9}$$

A more detailed explanation of ρ can be found in section 4.
The vector π contains values that correspond to the P-values, but also has positive entries corresponding to illegal moves $a \notin \alpha(s, p)$. We apply algorithm 3 to it, to mask those illegal moves. Finally, we can obtain the P-values from vector π.

Equation 3.2.4 represents an important concept in the Monte-Carlo tree search:

Exploitation vs. Exploration

When choosing a path in the tree, promising sub-trees are favored upon those that have a low Q-value. On the other hand, it can be beneficial to explore paths that do not seem promising in the beginning, because they could be superior in the long run.
The first term refers to exploitation. Paths that already have led to a state with positive reward are investigated further.
The second term describes optimism regarding exploration. This term grows every time, other actions get favored over action a in state s. Therefore seldom chosen actions have a higher exploration-value.
The exploration parameter c_{puct} balances these two terms out. A low value leads to more focus on exploitation and therefore a deeper tree. When a high value is chosen, more different actions are explored, making the tree wider.

Another difference to Minimax is that the Monte-Carlo tree search does not return a single action a as the best choice, but a vector with values for every action in A. Those values are proportional to the number of times that the respective edge starting at the root node was traversed.

We split the algorithm into three functions. The first one expands the tree by one more leaf and updates the tree values.

Algorithm 2 search

function SEARCH(s, p)
  if s is a terminal state **then**
    return $\phi(s, p)$
  if s is a leaf node **then**
    $v \leftarrow \phi(s, p)$
    $\pi \leftarrow \rho(s, p)$
    $\pi \leftarrow$ MASKILLEGALMOVES(π)
    for each a in $\alpha(s, p)$ **do**
      $P_{s,a,p} \leftarrow \pi_a$
      $N_{s,a,p} \leftarrow 0$
      $Q_{s,a,p} \leftarrow 0$
    $N_{s,p} \leftarrow 0$
    return v
  $u_{best} \leftarrow -\infty$
  $a_{best} \leftarrow 0$
  for each a in $\alpha(s, p)$ **do**
    $u \leftarrow Q(s, a) + c_{puct} \cdot P_{s,a,p} \cdot \frac{\sqrt{N_s}}{1+N_{s,a,p}}$
    if $u > u_{best}$ **then**
      $u_{best} \leftarrow u$
      $a_{best} \leftarrow a$
  $s' \leftarrow m_s(s, a_{best}, p)$
  $p' \leftarrow m_p(s, a_{best}, p)$
  $v \leftarrow$ SEARCH(s', p')
  $Q_{s,a_{best},p} \leftarrow \frac{N_{s,a_{best},p} \cdot Q_{s,a,p} + g(v,p)}{N_{s,a,p}+1}$
  $N_{s,a,p} \leftarrow N_{s,a,p} + 1$
  $N_{s,p} \leftarrow N_{s,p} + 1$
  return v

The second function masks illegal moves to ensure correct P-values.

Algorithm 3 maskIllegalMoves

function MASKILLEGALMOVES(π)
  for each a in A **do**
    if a in $\alpha(s, p)$ **then**
      $\pi_a \leftarrow \frac{\pi_a}{\sum_{\tilde{a} \in \alpha(s,p)} \pi_{\tilde{a}}}$
    else
      $\pi_a \leftarrow 0$

The main function calls the search function num_{sim} times, constructing a tree. Then it counts how often each edge starting at the root node was chosen. It creates a normalized vector C containing these counts. The higher the value of C_a for $a \in A$ is, the more often it was chosen by the tree search. In section 4 we will have a more detailed look at how it is used to choose an action.

Algorithm 4 mcts

```
1: function MCTS(s, p, num_sim)
2:     for i ← 1 to num_sim do
3:         SEARCH(s, p)
4:     for each a in A do
5:         C(a) = N_{s,a,p}
6:     C ← C / sum(C)
7:     return C
```

3.3. Games with Repetition

Until now, we only discussed algorithms for games that have no recurring states. That means that once an action from state s is taken, there is no way for the players to go back to it. This is necessary for the game to be representable by a tree.
The game Chinese Checkers does not fulfill this condition. Because players can move their pieces back to previous positions, game loops are possible. Therefore, we can only represent the game as a directed graph, as opposed to a tree.
In order to still use the Monte-Carlo tree search, we make the following adjustments. In every iteration, we take notes of the path that is traversed in the tree. If the next node (s, p) is one that is already in that trace, it means that our path contains a loop. If we would ignore this problem, the values would be back-propagated in circles. To prevent this, we back-propagate not v, but the actual score $\tilde{v}$ of the state. $\tilde{v}$ does not describe the expected scores, but the scores realized in the game state. For all players that have not completed the game, this will evaluate to 0, making it unpromising to pursue that path further and thus preventing the exploitation of the loop.
In summary, we assign the node (s, p) a poor evaluation in the second visit, and the back-propagated evaluation when returning to the first visit. We will later refer to this as loop-cutting. Note that this only prevents loops inside the tree. Loops in actual game-play cannot be prevented this way. In subsection 7.1 we will further address this problem, and how it was handled.

4. Agents

We call an entity that chooses moves, an agent. Agents are not statically linked to a player. An agent can dictate game-play for any player and even for multiple players in one game.
Just like the other games that AlphaZero was applied to, Chinese Checkers is a symmetric game. This means that every player has the same action space and the same objective. Therefore, we do not have to design different agents for specific players in the game.
When the game is being played in real life, every player sees the board from a different perspective, with his own start zone closest to him. Before every move, an agent also sees the board in the current player's perspective. This is achieved by 120° degree turns of the board. Furthermore, the game is being displayed with the player's pieces as red ones, the next player's pieces as yellow ones and the last player's pieces as green ones. Through this generalization, an agent always plays, as if he was player one, with his start zone at the bottom and his end zone at the top side of the board. We accomplish this by applying a function r_s to the board state:

$$r_s : S \times P \to S \tag{4.0.10}$$

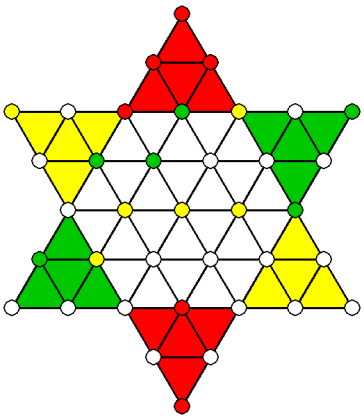


Figure 4.0.10: $r_s(s, 1)$

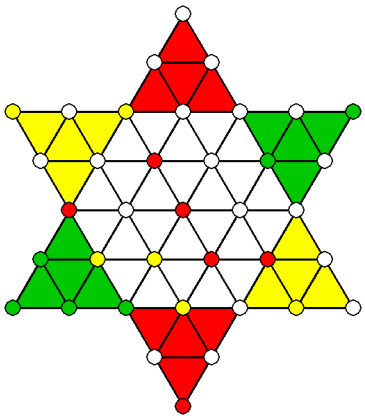


Figure 4.0.10: $r_s(s, 2)$

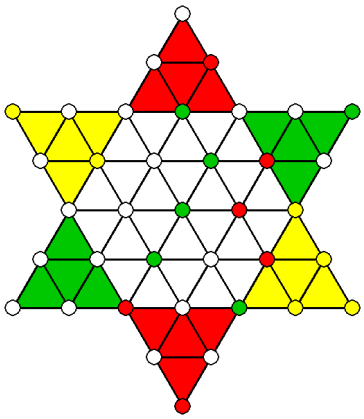


Figure 4.0.10: $r_s(s, 3)$

Figure 4.0.10: Rotations of an example board state s

There are several agents implemented in the project.

4.1. Main Agent

The Main Agent is being trained in the project. The agent chooses moves with the help of a Monte-Carlo tree search. The tree search is entwined with an artificial neural network, which fulfills two functions:
When a leaf node (s, p) is reached, it evaluates the board state and returns v. It also provides vector π that contains the the pre-evaluation values $P_{s,a,p}$ for all $a \in A$.
The pre-evaluation values of all possible moves would be enough information to choose one. However, the tree search simulates game-play several moves in advance and provides a look-ahead. In section 6 we will see that the tree search also plays an important role in the training process of the network.

ρ returns the pre-evaluation values in the tree search. It uses the function f_v that is part of the neural network.

$$\rho(s,p) = f_\pi(r_s(s,p)) \tag{4.1.11}$$

As already explained, the network does not get s as an input, but $r_s(s,p)$. It does not have the information, how the board looks in the actual game and who's turn it is. Therefore it does not return the state value v, but a permutation v_r of v. In v_r the first entry corresponds not to the expected score of player one, but the expected score of the current player et cetera. Therefore, we need a function, that maps v_r to v:

$$r_v : V \times P \mapsto V \tag{4.1.12}$$

Finally, we can formulate our state evaluation function ϕ, which also makes use of the neural network.

$$\phi(s,p) = r_v(f_v(r_s(s,p))) \tag{4.1.13}$$

In summary, the board is first rotated, then evaluated by the network and lastly, the scores are being reordered to match the actual players.

After the execution of the Monte-Carlo tree search, as described in algorithm 4, the vector C is obtained. It describes the number of visits of each edge from the root node.
There are two ways to use this vector.
We can choose the action, whose edges in the last MCTS-iterations has the highest count. If there are several edges that have maximal counts, we randomly select one of them.

$$A^* := \arg\max_{a \in A} C_a \tag{4.1.14}$$

$$P(a) = \begin{cases} \frac{1}{|A^*|}, & \text{if } a \in A^* \\ 0, & \text{otherwise} \end{cases} \tag{4.1.15}$$

$P(a)$ is the probability that action a gets chosen.
We call this procedure the best selection. If $|A^*|$ equals 1, which is the case most of the time, this leads to a problem. Until the weights of the network are being changed, and thus the probability vector C, action a^* will always be the same for a fixed state s. Given the same group of agents, a game is therefore strictly determined and every replay will be an exact repetition. To collect more data for the training and evaluation process we introduce a second selection method.

We choose randomly from the tried actions and assign the relative visits of action a as its probability.

$$P(a) = C_a \quad \text{for all } a \in A \tag{4.1.16}$$

This will be referred to as varied selection.
Both selection methods are being used in the program.
Furthermore, we can set the number of iterations num_{sim} in the tree search to 2. In this case, only the action with the highest P-value is taken once. We call this the Main Agent without tree search, or simply the neural network.

4.2. Greedy Agent

To provide an analytic evaluation of moves, we divide the board into rows and record how many rows the piece moved upwards. This is described by a function e.

$$e : A \to \mathbb{Z} \tag{4.2.17}$$

$e(a)$ describes the progress resulting in a move a. The greedy agent always chooses one of the moves with the highest progress. It is the most simple plausible game strategy.

$$A_G = \underset{a \in \alpha(r_s(s,p),1)}{\arg\max} \; e(a) \tag{4.2.18}$$

$$P(a) = \begin{cases} \frac{1}{|A_G|}, & \text{if } a \in A_G \\ 0, & \text{otherwise} \end{cases} \tag{4.2.19}$$

We use the Greedy Agent as a benchmark for the performance of the Main agent.

4.3. Initialize Agent

The Initialize Agent is used to initialize our neural network.
It chooses random moves, but has a strong tendency to move pieces towards the end zone. A neutral move to the side is twice as likely as a backward move, and a forward move is twice as likely as a neutral move to the side. We define the vector $I \in \mathbb{N}_0^a$ with:

$$I_a = \begin{cases} 1, & \text{if } a \in \alpha(r_s(s,p),1) \text{ and } e(a) < 0 \\ 2, & \text{if } a \in \alpha(r_s(s,p),1) \text{ and } e(a) = 0 \\ 4, & \text{if } a \in \alpha(r_s(s,p),1) \text{ and } e(a) > 0 \\ 0, & \text{otherwise} \end{cases} \tag{4.3.20}$$

$$P(a) = \frac{I_a}{||I||_1} \quad \text{for all } a \in A \tag{4.3.21}$$

Similar to the result of the Monte-Carlo tree search, it returns a vector $C \in [0,1]^{|A|}$.

$$C_a = P(a) \quad \text{for all } a \in A \tag{4.3.22}$$

5. Neural Network

In this section, we will take a look at the neural network and how it is trained.
An artificial neural network is a concatenation of linear and non-linear functions. We describe these functions as layers. The non-linear functions are called activation functions.
A neural network can also contain regularizing functions. These functions improve training.
The functions of the network contain parameters that we denote with the vector $w \in W \subset \mathbb{R}^n$. These parameters are often called weights. Therefore, the output depends on the function input and the weights. In the training process, we modify these parameters. If this is done correctly, the network will return better outputs than before. What exactly is considered a better output, will be measured by labeled data that consists of an input and the output that is desired by the user. The labeled data is provided as a batch B. A batch consists of n_B training examples. Every training example is composed of one example of data for each input and output.
Therefore, a neural network can be understood as a complex optimization problem.

Given a board configuration, our neural network makes two predictions that are used in the Monte-Carlo tree search. It calculates the pre-evaluation vector π.

$$f_\pi : S \times W \to [0,1]^{|A|} \tag{5.0.23}$$

Furthermore, it calculates a state value v

$$f_v : S \times W \to V \tag{5.0.24}$$

We define function f that combines both calculations.

$$f : S \times W \to [0,1]^{|A|} \times V \tag{5.0.25}$$

5.1. Network architecture

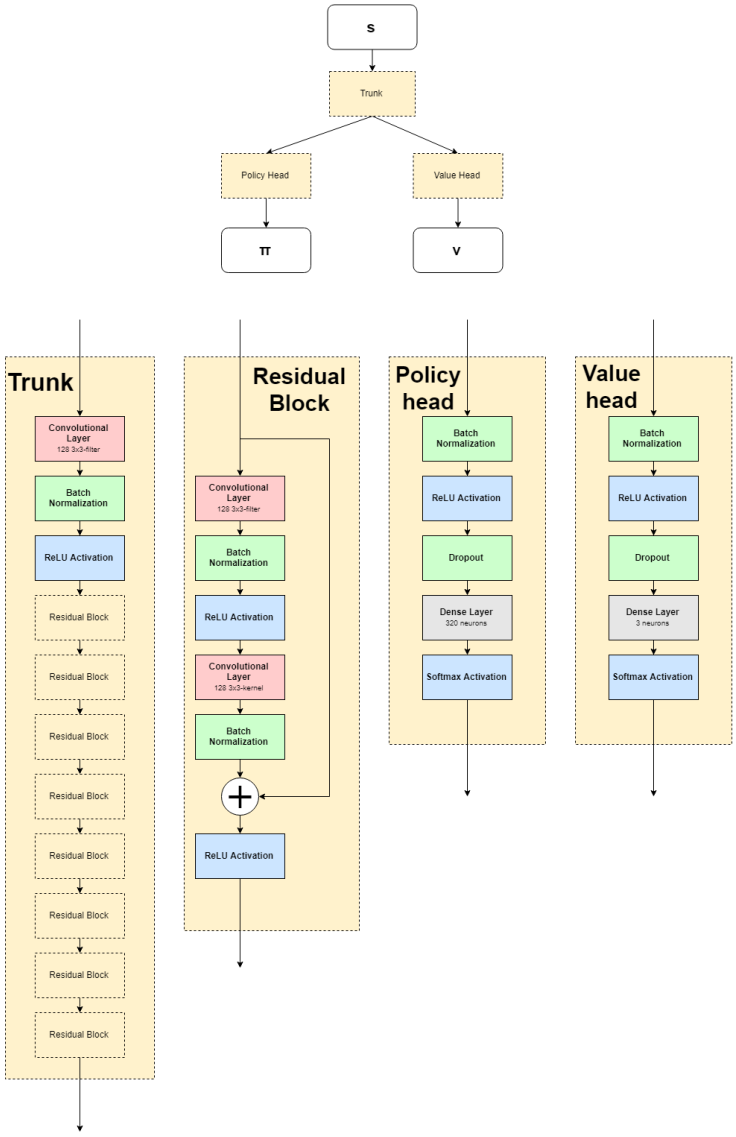


Figure 5.1.26: Neural Network Architecture
The figure describes the architecture of our neural network. The components were colored, with the use of red for convolutional layers, green for normalization layers, blue for activation layers, gray for dense layers and beige for blocks composed of several layers.

We use a residual neural network. It can be divided into three parts. The trunk, the policy head and the value head. As input, it takes the 9×9 array representation of the board state. The trunk processes the input and returns a 512-element vector. This vector then is taken as input by the policy head as well as the value head, which return the final outputs.

In the following, we will examine the layers that the network is composed of. The network also contains methods that reshape the data. For example, a $n \times m$ matrix will be represented by a $(n \cdot m)$ vector after reshaping. These reshaping methods are not listed, but their placements

can be reasoned by the form of outputs and inputs of successive layers.

5.1.1. Dense layer

Dense layers are the most basic layers used in neural networks. They take the a vector as input and return linear combinations of them. The entries of its output are called neurons.

$$\begin{aligned} d : \mathbb{R}^n \times \mathbb{R}^{n\cdot m+m} &\to \mathbb{R}^m \\ (x,u) &\mapsto \begin{pmatrix} x_1 \cdot u_1 + ... + x_n \cdot u_n + u_{n\cdot m+1} \\ \vdots \\ x_1 \cdot u_{n\cdot(m-1)+1} + ... + x_n \cdot u_{n\cdot m} + u_{n\cdot m+m} \end{pmatrix} \end{aligned} \tag{5.1.27}$$

The entries of u are learnable parameters that get modified during training.

5.1.2. Convolutional Layer

Convolutional neural networks are used for pattern detection. A convolutional layer can, for example, detect edges and corners in an image, and several layers in combination are able to detect complex patterns. We use 2-dimensional convolutional layers to detect abstract patterns in the game, like ladders and blocks of pieces.
One layer contains a fixed number of $k \times k$ matrices. Those matrices are called filters. We call k the kernel size. It has to be an uneven number.
We slide a filter across the input matrix X. The part of the array that is temporarily covered by a filter is called a window. During one iteration every array element is the center of a window once. To make this possible, a frame of the thickness $\frac{k-1}{2}$ consisting of zeros has to be put around the input.

$$\tilde{k} := \frac{k-1}{2} \tag{5.1.28}$$

$$\tilde{x}_{i,j} := \begin{cases} X_{i,j}, & \text{if } 1 \le i \le n_1 \text{ and } 1 \le j \le n_2 \\ 0, & \text{otherwise} \end{cases} \quad \text{for } i = -\tilde{k}+1, ..., n_1+\tilde{k}; j = -\tilde{k}+1, ..., n_2+\tilde{k} \tag{5.1.29}$$

We multiply each filter element by the matrix element it is currently covering. Then we sum up the products. This sum is the value for the output matrix corresponding to the center of the window.

$$\begin{aligned} C : \mathbb{R}^{n_1\times n_2} \times \mathbb{R}^{k\times k} &\to \mathbb{R}^{n_1\times n_2} \\ (X,F) &\mapsto Y \end{aligned} \tag{5.1.30}$$

$$Y_{i,j} = \sum_{l=1}^{k}\sum_{m=1}^{k} F_{k,m} \cdot \tilde{x}_{i-(\tilde{k}+1)+l, j-(\tilde{k}+1)+m} \tag{5.1.31}$$

For every filter, a different matrix is the output. If the input of the layer is one array, like in the first instance used in the project, the input array gets duplicated to match the number of filters. If the input consists of as many arrays as there are filters, every input array will be transformed by a different filter.

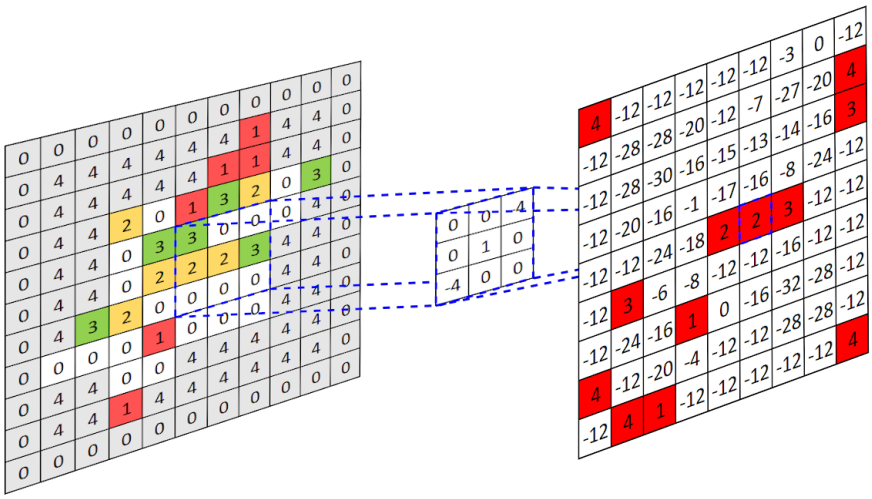


Figure 5.1.32: Convolutional layer application
This image shows an example filter applied to the board state. It detects fields than can be jumped over in the upper right or lower left direction. The positions that have positive indices and do not lie on the border indicate these fields.

The entries of each filter matrix are learnable parameters.

5.1.3. ReLU Activation

The rectified linear unit (ReLU) is an activation function. It introduces non-linearity to the neural network.
We use it as the activation function for output v.

$$g : \mathbb{R}^n \to \mathbb{R}^n$$

$$x \mapsto \begin{pmatrix} \max(0, x_1) \\ \vdots \\ \max(0, x_n) \end{pmatrix} \tag{5.1.33}$$

5.1.4. Softmax Activation

The softmax function is used as activation function for the output layer of π.
The π vector represents a probability distribution with values between 0 and 1 and a sum of 1.

The softmax function returns a vector that satisfies this condition. This is the reason, why it is usually applied as an activation for outputs that represent probability distributions.

$$\begin{aligned} g : \mathbb{R}^n &\to \mathbb{R}^n \\ x &\mapsto \frac{1}{\sum_{i=1}^{n} \exp(x_i)} \begin{pmatrix} \exp(x_1) \\ \vdots \\ \exp(x_n) \end{pmatrix} \end{aligned} \tag{5.1.34}$$

5.1.5. Dropout

A well-known problem with large neural networks is that they tend to overfit the data. Overfitting occurs, when the network adjusts its predictions too closely to the training data. In this case, the network also adapts to outliers, resulting in a loss of generality.
A way to prevent this would be to train multiple neural networks and use the mean of their predictions. Dropout builds upon this approach. Using it, we do not need to train multiple models, but make a slight variation to the architecture of our network, every time we train it. When predicting, a dropout layer does not affect the network.
A dropout layer has a specified dropout probability $p_d \in (0, 1)$. In our dropout layers it is equal to 0.3.
In every training iteration, every input neuron of a dropout layer has a probability of p_d to be ignored in the training process. The network is temporarily modified so that these neurons do not exist. Therefore, the weights corresponding to the disabled neurons are not being updated. Dropout layers slow down the training of a neural network, but increase its stability.

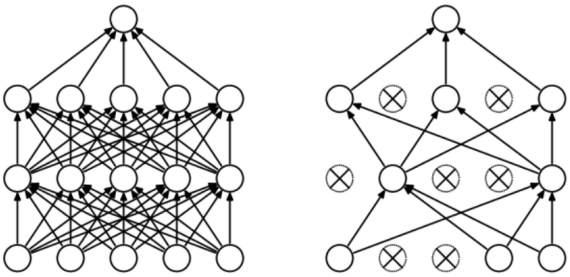


Figure 5.1.35: Network using dropout[9]
These figures show a network with dense layers and an output. In figure (b) a dropout layer is inserted between each layer. The crossed-out neurons represent the ones, that are ignored in the next layer in one specific training iteration.

5.1.6. Batch Normalization

Batch normalization is another regularization technique. The input of a layer can vary rapidly. This can lead to instability of the network. To prevent this, batch normalization was introduced.[3] Just like Dropout, it behaves differently during prediction and training phase. Batch normalization normalizes the data during training as following:
For every dimension $i \in \{1, \ldots, n\}$ of the data, the average over the training batch B is calcu-

lated.

$$\mu_i = \frac{1}{n_B} \sum_{j=1}^{n_B} x_i^{(j)} \tag{5.1.36}$$

The variance of each entry over the batch is calculated.

$$\sigma_i = \frac{1}{n_B} \sum_{j=1}^{n_B} (x_i^{(j)} - \mu_i)^2 \tag{5.1.37}$$

For every entry of x, we subtract the batch average and divide the result by the standard derivation over the batch. Therefore, the batch is now centered around the zero vector with standard derivation 1.
ϵ in the equation is a very small number that guarantees numerical stability in the case that the standard derivation is equal to 0. It does not significantly influence the results otherwise.

$$\hat{x}_i = \frac{x_i - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}} \tag{5.1.38}$$

The vector $\hat{x}$ gets shifted to and scaled, resulting in the output vector y. Therefore the center and standard deviation of the batch are influenced by the parameters $\beta_D, \gamma_D \in \mathbb{R}$.

$$y_i = \gamma_i \hat{x}_i + \beta_i \tag{5.1.39}$$

Note that the input will be just forwarded without a change, if $\gamma_i = \mu_i$ and $\beta_i = \sigma_i^2$.

When the network is in prediction phase, there is slight difference in computation. There is no batch, whose mean μ_i and variance σ_i^2 can be computed. Therefore we substitute these values with $\hat{\mu}_i$ and $\hat{\sigma}_i{}^2$, which are the respective moving averages of previous training iterations.

The learnable parameters of batch normalization are $\hat{\mu}_i$, $\hat{\sigma}_i{}^2$, γ_i and β_i. Since $\hat{\mu}_i$ and $\hat{\sigma}_i{}^2$ are computed by moving averages, they are the only learnable parameters in the network that do not get changed by the optimizer.

5.1.7. Residual Block

Residual neural networks are characterized by their residual blocks. In residual blocks, there is a skip connection implemented. Through this connection, the input of the residual block gets added to the processed data before the activation function of the block.
From a certain point on, the performance of neural networks without skip connections tends to stagnate or even decline in quality with every layer that is added. Residual blocks are used to prevent this.
A more detailed explanation can be found in the paper, in which they were proposed.[2]

5.2. Loss Function

The loss function describes the error of a prediction, given labeled outputs $\hat{\pi}$ and $\hat{v}$. For the outputs v and π different loss functions are being used.

Since π represents a probability distribution, the loss is described by Categorical Cross-Entropy.

$$
\begin{aligned}
l_\pi : S \times [0,1]^{|A|} \times W &\to \mathbb{R} \\
(s, \hat{\pi}, w) &\mapsto -\sum_{i=1}^{|A|} \hat{\pi}_i \log(f_\pi(s,w)_i)
\end{aligned}
\tag{5.2.40}
$$

The difference of score estimations in v and the labeled scores $\hat{v}$ is measured by mean squared error.

$$
\begin{aligned}
l_v : S \times V \times W &\to \mathbb{R} \\
(s, \hat{v}, w) &\mapsto \frac{1}{3} \sum_{i=1}^{3} (f_v(s,w)_i - \hat{v}_i)^2
\end{aligned}
\tag{5.2.41}
$$

The overall loss function consists of the sum of both loss functions.

$$
\begin{aligned}
l : S \times [0,1]^{|A|} \times V \times W &\to \mathbb{R} \\
(s, \hat{\pi}, \hat{v}, w) &\mapsto l_\pi(s, \hat{\pi}, w) + l_v(s, \hat{v}, w)
\end{aligned}
\tag{5.2.42}
$$

In the training process s, $\hat{v}$ and $\hat{\pi}$ are fixed. We want to find a $w \in W$ that minimizes the loss. To clarify this we use another formulation of the function that describes the loss of one tuple of training data in respect to w.
We don't train on a single tuple, but on a large number of labeled training tuples at the same time. Given the training batch $B \in (S \times [0,1]^{|A|} \times V)^{n_B}$, we can formulate the loss function that we want to minimize.

$$
\begin{aligned}
L_B : W &\to \mathbb{R} \\
w &\mapsto \sum_{i=1}^{n_B} l(s^{(i)}, \hat{\pi}^{(i)}, \hat{v}^{(i)}, w)
\end{aligned}
\tag{5.2.43}
$$

5.3. Optimizers

During the training process, an optimizer changes parameters of the model. The loss function gets minimized with respect to the weights. To accomplish this, there are several optimizers that are being frequently used in machine learning. These optimizers are iterative algorithms that start at a starting point $x^{(0)}$ and (not always successfully) step towards the minimum of a function. When training a network, w is that starting point.
In this case, the Adam Optimizer was chosen. The following section will explain gradient descent as its basis. Afterwards, different enhancements are introduced, leading to Adam Optimizer, which incorporates all of them.
We introduce function F, of which we want to estimate a minimum. We also have to know the gradient J of F.

$$\mathbf{F}\colon \mathbb{R}^n \to \mathbb{R} \quad x \longmapsto F(x), \qquad n \in \mathbb{N}$$

$$\mathbf{J}\colon \mathbb{R}^n \to \mathbb{R} \quad x \longmapsto \begin{pmatrix} \frac{\partial F}{\partial x_1} \\ \vdots \\ \frac{\partial F}{\partial x_n} \end{pmatrix} \tag{5.3.44}$$

5.3.1. Gradient Descent

Gradient descent is a numerical, iterative algorithm used to find the minimum of a function. Given an arbitrary starting value it successively keeps stepping in the direction of the current negative gradient. The step size is the gradient's magnitude multiplied by the learning rate parameter $\alpha_l \in \mathbb{R}_{>0}$. In the context of machine learning, the number of iterations that are executed is called n_{epoch}.

$$x^{(i)} = x^{(i-1)} - \alpha_l \cdot J(x^{(i-1)}) \quad \text{for } i = 1, ..., n_{epoch} \tag{5.3.45}$$

Convergence is only guaranteed for convex function with Lipschitz-continuous partial derivatives and a sufficiently small learning rate. Although there usually is no prior knowledge about the often non-convex loss functions in machine learning problems, it is widely applied with the goal of finding a local minimum. However, it cannot be guaranteed that the global minimum will be found.
The right choice for α_l depends on the function. A smaller value leads to smaller steps and can slow down the minimization. A value that is too large however can prevent the algorithm from converging or even lead to divergence.
We apply gradient descent to the loss function L_B to find parameters $w^* \in W$ that minimize the loss.

5.3.2. Mini-Batch

When we train a neural network, we feed it a large amount of labeled data. Hence, the computation of each gradient is computationally expensive. The idea of mini-Batch is to split the training batch into mini-batches of size n_b and execute each gradient descent step just over one mini-batch. Thus, each iteration minimizes w over a slightly different loss function. This speeds up the computation of its gradient, trading off accuracy for each step. This comes from the fact that the randomly selected mini-batch used for one step does not have to be a good representation of the whole training set, resulting in a loss function that greatly differs from L_B. In theory however, these inaccuracies cancel each other out.
We do not iterate over $L_b(w)$, but over the functions $L_{b_i}(w)$. We take smaller steps, but multiply the number of steps by the number of mini-batches.

$$L_{b_i}(w) := \sum_{j=(i-1)\cdot n_b+1}^{\min\{i\cdot n_b, n_B\}} l(s^{(j)}, \hat{v}^{(j)}, \hat{\pi}^{(j)}, w) \text{ for } i = 1, ..., \left\lceil \frac{n_B}{n_b} \right\rceil \tag{5.3.46}$$

$$J_{b_i}(x) := \begin{pmatrix} \frac{\partial L_{b_i}}{\partial x_1} \\ \vdots \\ \frac{\partial L_{b_i}}{\partial x_n} \end{pmatrix} \tag{5.3.47}$$

$$x^{(i)} = x^{(i-1)} - \alpha_l \cdot J_{b_i}(x^{(i-1)}) \quad \text{for } i = 1, ..., \left\lceil \frac{n_B}{n_b} \right\rceil \cdot n_{epoch} \tag{5.3.48}$$

The batch size has to be set by the user and its efficiency depends on factors, like the data size of one training tuple, the physical machine that is used for training and the variance of the data. A mini-batch gradient descent method that uses batch size 1 is called stochastic gradient descent.

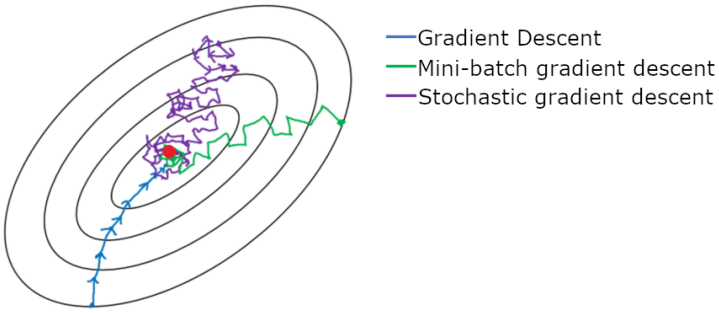


Figure 5.3.49: Mini-batch gradient descent[1]
A 2-dimensional function is minimized with standard gradient descent, Mini-batch gradient descent and stochastic gradient descent. To illustrate this better, different starting points were used.

5.3.3. Momentum

The gradients computed in every step of mini-batch gradient descent can change directions rapidly, while often alternating around the ideal directional vector that leads towards a local minimum. Gradient descent with momentum takes advantage of that. Just like in physics, where the movement of an object is influenced by its momentum, gradients from previous iterations continue to have an effect on the current step. By taking an exponential moving average of the past gradients, it dampens the directional changes. This can result in faster convergence.
The parameter $\beta_M \in (0, 1)$ controls the influence of momentum M.

$$M^{(0)} = 0 \tag{5.3.50}$$

$$M^{(i)} = \beta_M \cdot M^{(i-1)} + (1 - \beta_M) \cdot J(x^{(i-1)}) \tag{5.3.51}$$

$$\tilde{M}^{(i)} = \frac{M^{(i)}}{1 - \beta_M^i} \tag{5.3.52}$$

$$x^{(i)} = x^{(i-1)} - \tilde{M}^{(i)} \quad \text{for } i = 1, ..., n_{epoch} \tag{5.3.53}$$

5.3.52 describes a bias correction. Since M_0 is initialized with 0, this low value drags down the moving average, especially in the earlier iterations. To correct this we divide M_i by $(1 - \beta_M^i)$. This cancels out the influence of M_0 without interfering with the moving average.

5.3.4. RMSProp

Another method, which corrects gradient orientation, is Root Mean Square Propagation (RMSProp). The magnitude of the partial derivatives can vary to such an extent that the gradient takes huge steps with respect to one weight but neglects some others. Instead of letting only the gradient determine, how far to move in each direction, RMSProp uses a different learning rate for each weight respectively. It uses larger learning rates for gradient vector entries, which have a smaller absolute value. The overall learning rate is divided by the square-root of an exponential average of past squared partial derivatives. ϵ is a very small number that guarantees numerical stability.

$$s_j^{(i)} = \beta_S \cdot s_j^{(i-1)} + (1 - \beta_S) \cdot (\frac{\partial F}{\partial x_j}(x^{(i-1)}))^2 \quad \text{for } j = 1, \ldots, n \tag{5.3.54}$$

$$x^{(i)} = x^{(i-1)} - \left(\frac{\alpha_l}{\sqrt{s_1^{(i)} + \epsilon}} \cdots \frac{\alpha_l}{\sqrt{s_m^{(i)} + \epsilon}} \right) \cdot J(x^{(i-1)}) \tag{5.3.55}$$

5.3.5. Adam

Adaptive Momentum Estimation (Adam) combines the ideas of the past subsections. It is a mini-batch gradient descent algorithm using momentum and Root Mean Squared Propagation.

$$M^{(i)} = \beta_M \cdot M^{(i-1)} + (1 - \beta_M) \cdot J_{b_i}(x^{(i-1)}) \tag{5.3.56}$$

$$\tilde{M}^{(i)} = \frac{M^{(i)}}{1 - \beta_M^i} \tag{5.3.57}$$

$$s_j^{(i)} = \beta_S \cdot s_j^{(i-1)} + (1 - \beta_S) \cdot (\frac{\partial F}{\partial x_j}(x^{(i-1)}))^2 \quad \text{for } j = 1, \ldots, n \tag{5.3.58}$$

$$x^{(i)} = x^{(i-1)} - \left(\frac{\alpha_l}{\sqrt{s_1^{(i)} + \epsilon}} \cdots \frac{\alpha_l}{\sqrt{s_m^{(i)} + \epsilon}} \right) \cdot \tilde{M}^{(i)} \quad \text{for } i = 1, \ldots, \left\lceil \frac{n_B}{n_b} \right\rceil \cdot n_{epoch} \tag{5.3.59}$$

Adam Optimizer has proven to be a very good off-the-shelf optimizer for neural networks. We are using it with the parameters $n_{epoch} = 5$, $\alpha_l = 0.001$, $\beta_M = 0.9$ and $\beta_S = 0.999$.

6. Project Structure

After examining the building blocks used in the project, we will now look at how they are combined.
The main program consists of a loop that gradually improves the Main Agent.

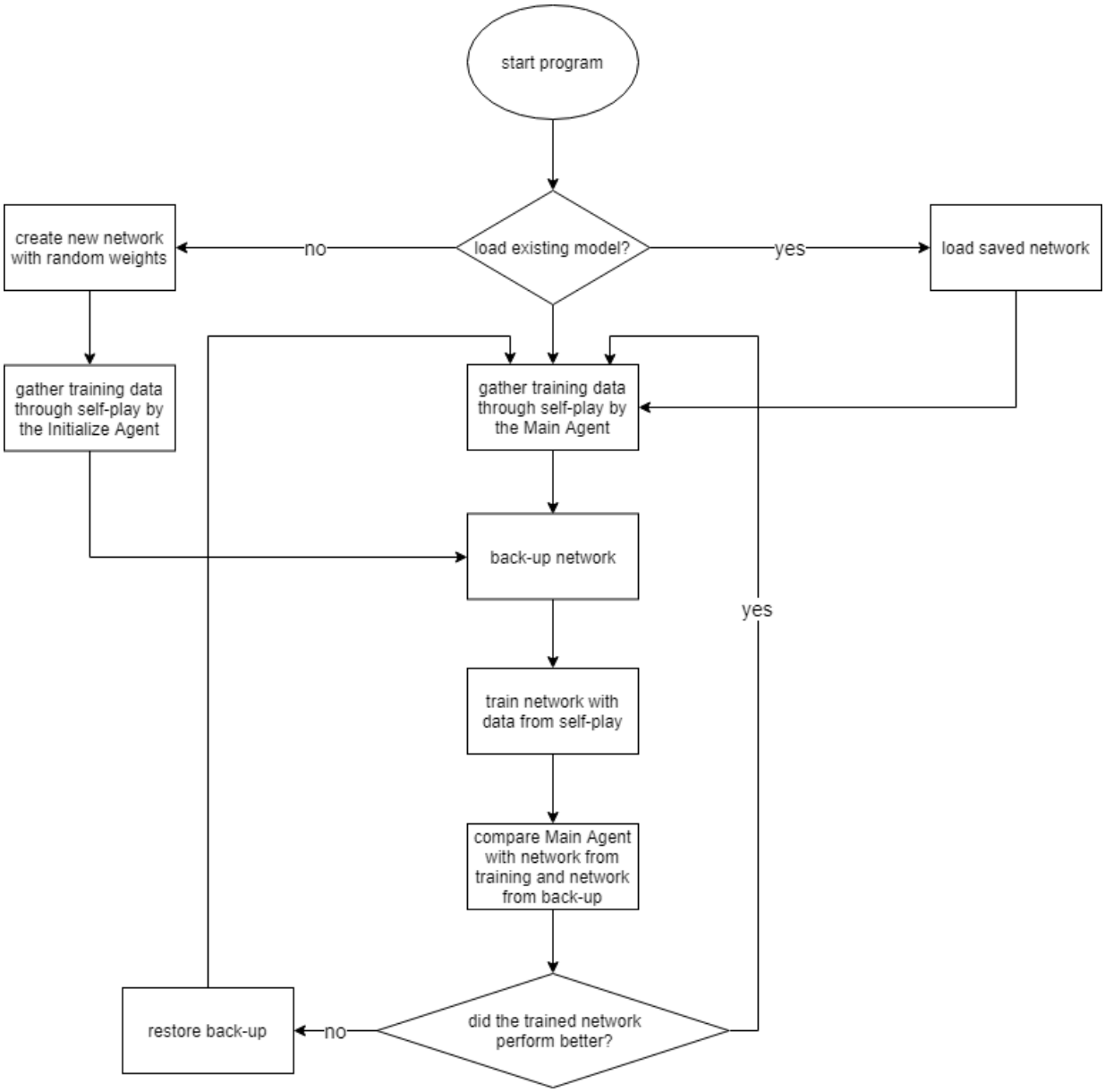


Figure 6.0.60: Flowchart of the training loop

Self-play

In this phase, training data for the neural network is created. n_{eps} games are played. Every turn is done by the same agent, trying to optimize the score of the current player. The first n_{varied} steps each are played with varied selection. Afterwards, best selection is used.

Before every move the tuple $(s, C, p) \in S \times [0,1]^{|A|} \times P$ is saved, where s is the current board state, C is the vector obtained by the Monte-Carlo tree search and p is the current player. When the game is over, the final state is evaluated to state value v^*. For every tuple (s, C, p) that we collected, we create a corresponding tuple $(r_s(s,p), C, r_v(v^*, p)) \in S \times [0,1]^{|A|} \times V$. They consist of the rotated board, vector C and the final scores out of perspective from player p. These tuples are used as training data for the neural network.

If this is the first iteration and no existing model was loaded, the Initialize agent is being used. Otherwise the agent corresponds to the current version of the neural network.

Back-up

We save a copy of the current neural network. If the network does not improve through the training, we can restore the old version later.

Training

We use the training data of the last 10 self-play sessions to train the network, provided that they were generated by the Main Agent. The training will result in the following changes in the network: The evaluation of a game state gets shifted towards the evaluation of the final board state s^* that it led to. The pre-evaluation π will shift towards the moves that the tree search favored in past simulations. This can result in problems, as we will discuss in subsection 7.3.

If this is the first iteration and the network is being initialized by the Initialize agent, we use the training examples once and discard them afterwards.

Arena

After the training, we have our old back-up version of the neural network and the new one. We let two agents with the respecting neural networks compete against each other.

The agents use varied selection for the first 5 steps each and best selection afterwards. To avoid bias, we play 2 vs. 1 games and alternate, which agent will play for two players and which one will play for one player. We also alternate, which player is controlled by which agent. Therefore, a set consists of 6 games. We play a fixed number of sets and add up the points scored of each agent. If the agent with the just trained network did significantly better (scored 55% of the points), we take it as the new standard. Otherwise we restore the old version.

This process is repeated with the goal of gradually improving the network's predictions and therefore making the agent a stronger player.

7. Difficulties

During the realization of the project, several problems occurred. This chapter will explain them and the measures taken to overcome them.

7.1. Loops

As already discussed, Chinese Checkers is a game that allows game loops. subsection 3.3 explains, how the tree search is modified to address this problem. However, since the tree is reset and built before every move, this modification does not prevent loops in actual game-play.
Similar to the trace, we save inside the tree search, we also list the actual board states of a game and generalize the idea of loop-cutting. We also intervene, if state s was reached in the game and it was the turn of player p. This is only done in the self-play games.
These counter-measures can prevent game loops but quasi-loops are still possible. Meant by that are series' of board states, in which no state appears twice, but that would likely be perceived as game loop by human players, because no progress in the game is made. These quasi-loops are especially conducted by players that are steps away from losing the game, preferring to drag the game over losing.

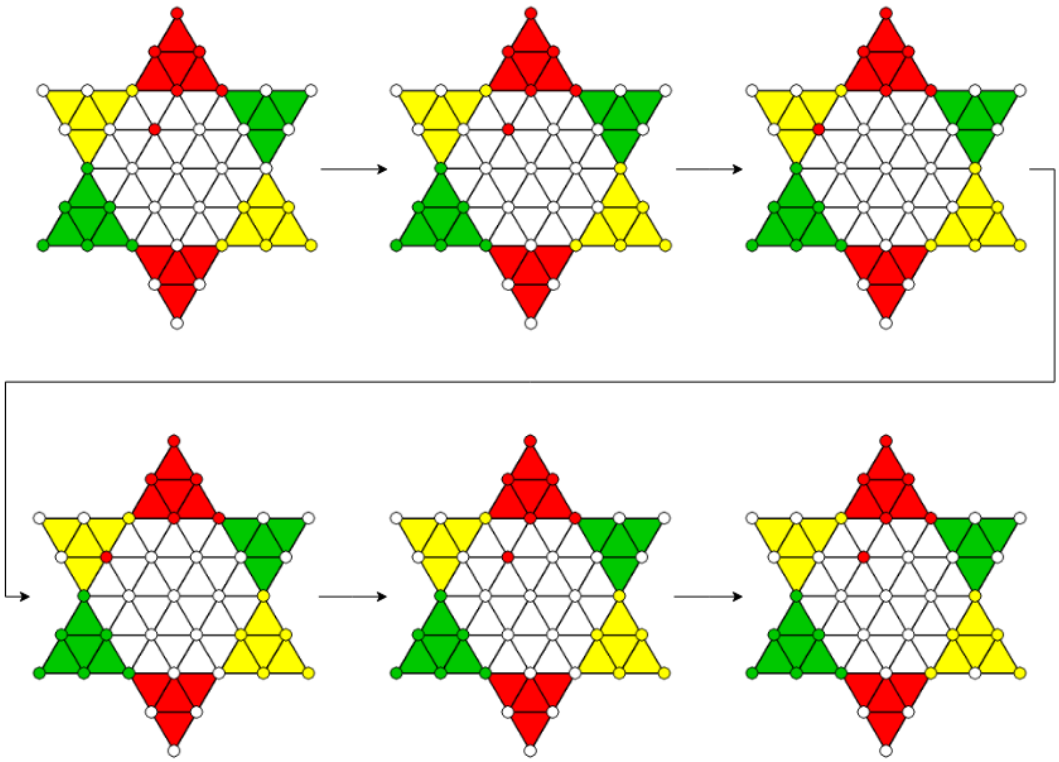


Figure 7.1.61: Example of a quasi-loop before the introduction of the second-winner rule
In the first board state, it is the turn of player two. Knowing that moving his piece out of player one's end zone would result in a loss, he just moves his other pieces around.

After the quasi-loop is played a specific number of times, all variations of it have been played. If this situation occurs in a self-play game during training, and it's the turn of the player with

the upper hand, he will be forced to prevent an actual game loop and alter the situation to his disadvantage. This way it is possible that he will be overtaken by the player that conducted the quasi-loops.
We do not want this scenario in our training. To prevent it, we introduced the second-winner-rule.
In the case that all these measures fail to prevent an infinite game, we stop the game after 40 moves each. Every player, that has scored points until then gets them awarded, while the others receive a score of 0. The agents do not have knowledge about how many steps were taken in game. Therefore, this will not influence their behavior. However, after the training process the board states that led to the situation will be evaluated with a lower score.

7.2. Triple Win

The second-winner-rule makes it possible that one player wins and the others share the second place. In this case, we award both of them a score of 1.

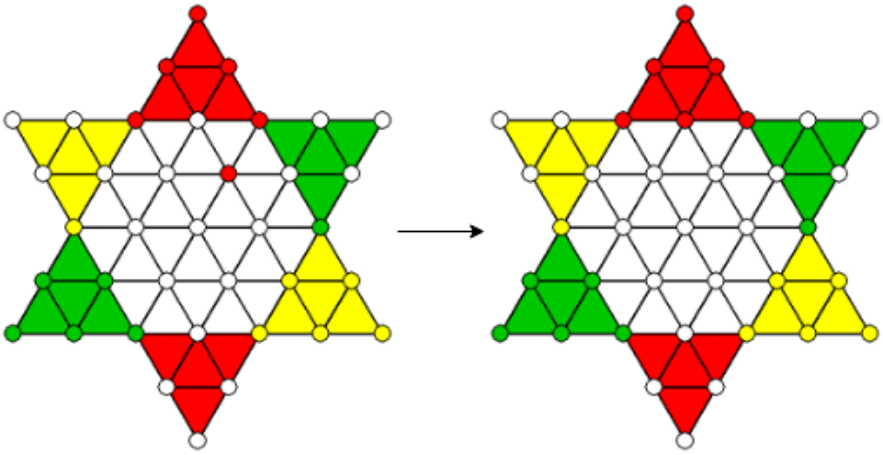


Figure 7.2.62: Triple Win

This scenario occurs very rarely, because both blocking players have no reason to rely on each other, and cooperation over multiple games is not possible in the project.

7.3. The right rate of exploration

The right choice for the exploration parameter c_{puct} is one of the most crucial factors determining the quality of the agent. In section 3, the mechanics of the Monte-Carlo tree search and the role of c_{puct} were already explained. In combination with the training process the influence of c_{puct} is even stronger. The lower c_{puct} is, the more distinct are the values of C. Because we feed C to the network as labeled training data for π, this effect gets amplified over time. When arriving at the same board state, the distribution of π will be more unbalanced and the tree search will focus even more on the paths that were previously favored, neglecting the unpromising ones. In an extreme consequence, the distribution of π can become so unbalanced, that the tree search does not contribute to the decision at all, because from the root node only the path suggested by the network gets explored. In the long run, this can stop the improvement of the neural network.

On the other hand, a value for c_{puct} that is too high, can shape π towards an uniform distribution, defying its purpose.
If one of these scenarios occur during training, there are several ways to address the problem. If the exploration parameter is too low, but the user is satisfied with the state of the neural network, the exploration parameter can be turned up to an extreme during the use of the agent. One can also change c_{puct} to the opposite extreme for a few training iterations with the goal of balancing out the distribution of π. The best solution is to experiment, until a suitable exploration parameter is found, and restart the training process. However, this is the most time- and resource-consuming approach.

7.4. Performance

The training process requires a lot of computational power.
At first, a version of the game with larger board size was implemented, but it soon became clear that the training process would take too long, and the board size was reduced. This is also the reason, why the number of possible board states and the action space were reduced with the rule that forbids players to move their pieces deeper into the zones of opponents.

Originally, the network was supposed to be trained with data obtained solely by self-play from previous versions of the Main Agent. Because of the recurring states in Chinese Checkers, a game consisting of randomly chosen moves is very long. Therefore, the first iterations of random self-play would be too time-consuming. The Initialize agent was designed to nudge the network in the right direction without influencing it too much.

The most costly process of the training loop is the generation of training examples via self-play. During a game of self-play, there were three tasks that require a lot of time, compared to the other ones. They are the board rotations, the determination of legal moves and the predictions by the neural network. To speed up the predictions, the games are played simultaneously. A more detailed description can be found in the next chapter.

8. Implementation

My project builds upon an open-source project that I found. This project by Surag Nair implements the training process described in the chapter section 6 and is a simplification of the structure used in AlphaZero. This project was applied to Othello, but is generalized in such a way that it can be used for other two-player board games as well. It is written in Python 3.

The overall project structure was adapted. To be applicable for 3-player games, every class was drastically changed and new classes were written. The Monte-Carlo tree search had to be adjusted for three players. The competition between the old and the just trained network is more complicated, because we have to evaluate two players in a 3-player game. The old project contained neural network templates written with the frameworks PyTorch and Tensorflow 1. I wrote a network in Tensorflow 2, using the Keras API. The design is more similar to the one used in AlphaZero.

The game Chinese Checkers was written from scratch, including a graphical user interface that is the source of the board figures in this thesis.

To speed up the self-play games, multiple games are played simultaneously.

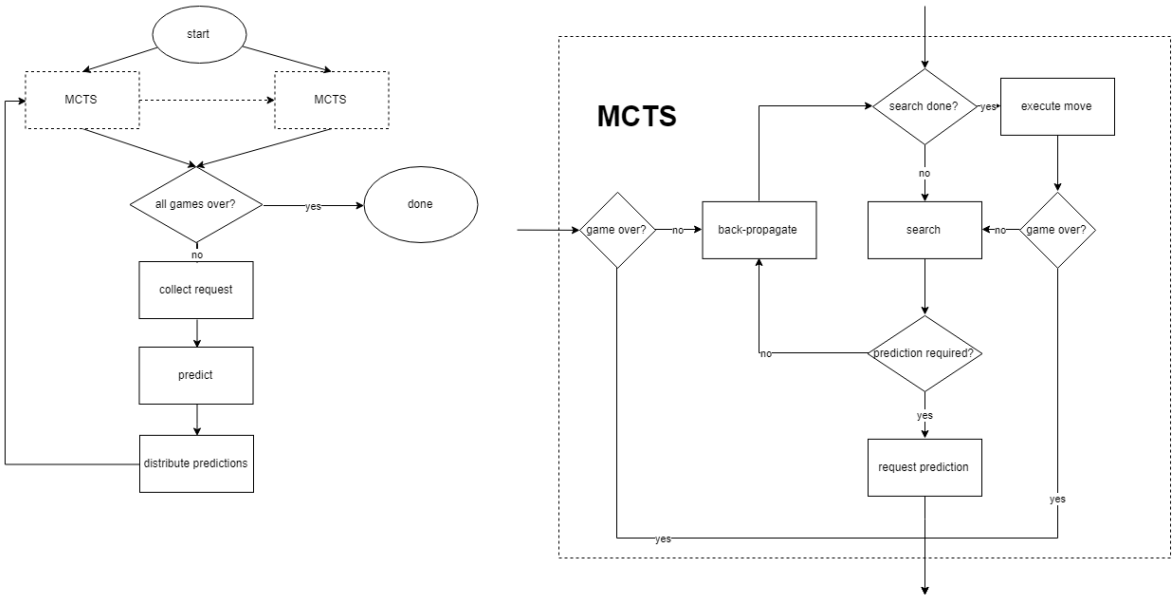


Figure 8.0.63: Simultaneous self-play

In every game instance, the agents play and search in the game tree, until a prediction by the network is requested. Then, the game-play is paused until every instance reaches this point. The game states, of which a prediction is needed, are collected and forwarded to the network. Afterwards, the outputs get distributed back to the game instances, in which they were requested, and the values are back-propagated. This process is repeated, until every game is over. To realize this, the Monte-Carlo tree search had to be written as an iterative function. The tree traversal and the back-propagation were separated into two functions. Because the path that was traversed in the tree cannot longer be retrieved from a recursion stack, it has to be saved in a list. The paths, the board states and all variables from the Monte-Carlo tree

search have to be saved for all games. Thus, this solution is very memory-intensive. A balance between memory usage and performance improvement was found with 50 games played at a time.

The training was executed on a Google Cloud Virtual Machine with 4 virtual CPU's, 24GB RAM, an SSD hard drive and one Tesla K80 GPU over a period of two weeks.

9. Results

The program was tested with following parameters:
The network was initialized by 500 games played by the Initialize Agent.
In one self-play session 200 games were played.
The tree search executes 200 search iterations.
The agents used varied selection for the first 15 moves each.
The exploration parameter c_{puct} was set to 15.

The neural network was trained, until it reached its 20^{th} version. Because the network was not updated in every training loop, 41 self-play sessions were executed. Thus, the last version was trained with 500 games played by the Initialize Agent and 8000 games played by previous versions.
When referring to version 1, the version after the initializing process is meant.

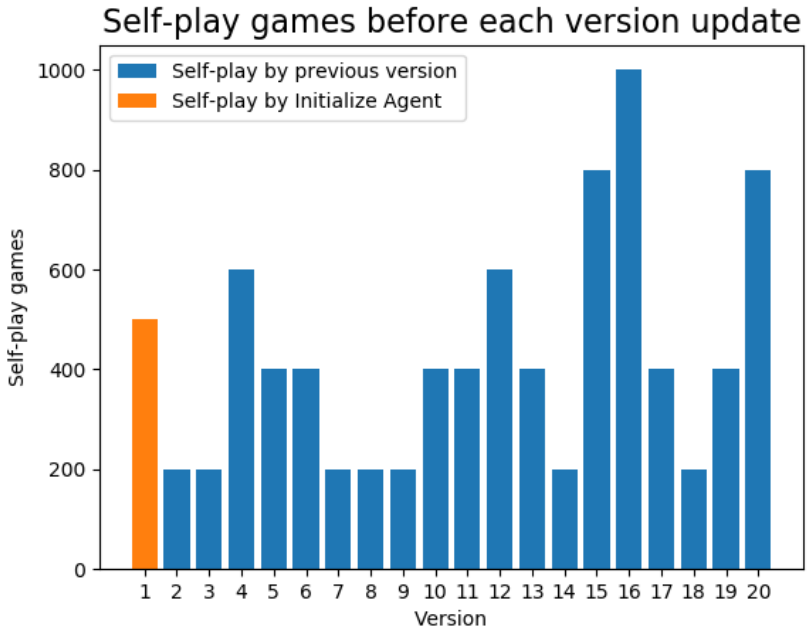


Figure 9.0.64: Self-play games played before each version update

The different versions were tested with the use of a Monte-Carlo tree search and also without it, relying just on the network. The tree search always uses best selection. In every game, the first two moves of every player are selected randomly to ensure variation. When we let two agents play against each other, we separate the results of the games, in which the first agent played as one player against two instances of the second agent and vice-versa. We refer to these games as single games and double games or "1 vs. 2"-games and "2 vs. 1"-games. One should not be misled by this nomenclature, thinking that players form actual teams. Every player still tries to maximize his own score only. However, we want to examine the combinations of placements, when two instances of our main agents play in one game.
In evaluations that contain the tree search 30 games were played. In all other evaluations 120 games were played.

9.1. Evaluation against previous versions

Inside the training loop, the Main Agent competes against its previous version to determine, if it has improved. Only the sum of scores but no separate scores from single and double games are compared. It can be seen that the performance can vary between single and double games.

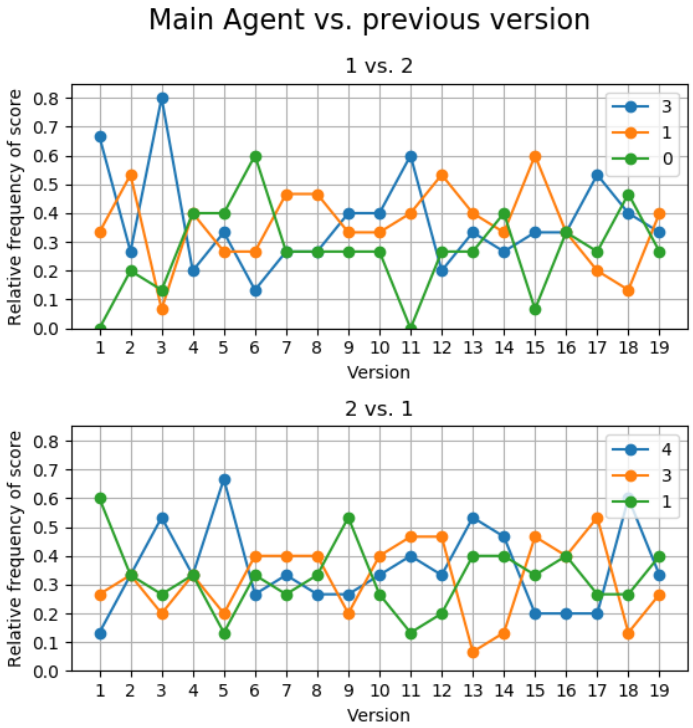


Figure 9.1.65: Main Agent vs. previous version

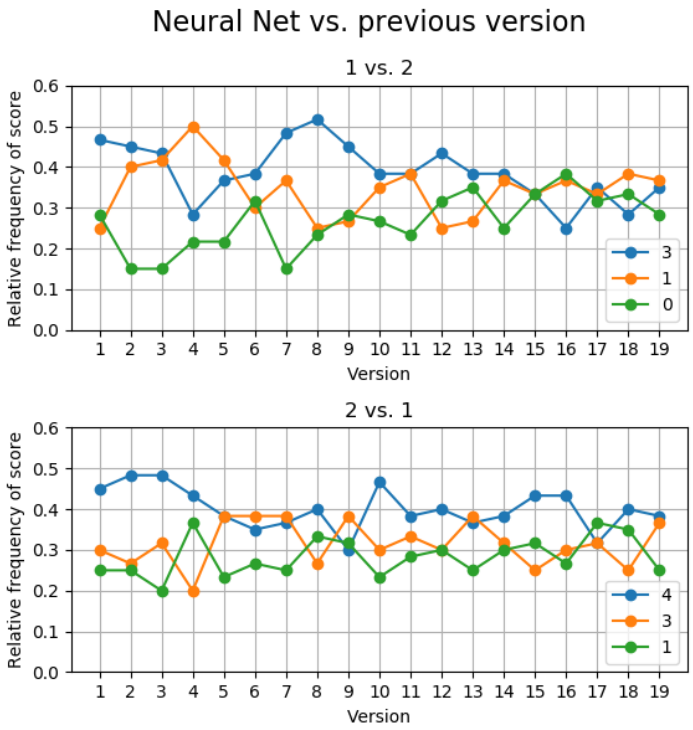


Figure 9.1.66: Neural Network vs. previous version

9.2. Evaluation against the Greedy Agent

The Greedy Agent serves as a benchmark for the Main Agent.
Even though some version updates led to a decline in performance against the Greedy Agent, the improvements are clearly visible.

When comparing the results of different scenarios, one thing stands out. The main agent performs better in double games, especially when using the tree search.
The reason for that lies in the nature of the Monte-Carlo tree search. Before every move, the decision of the agent is influenced by simulations of the game by the tree search. In these simulations, the agents estimate decisions by other players. Hence, these simulations are closer to the actual game development, when another player acts under the same policy.
Because the neural network was trained with help of the tree search, its pre-evaluations still carry the effect.

This is a weak point of the Main Agent that can be exploited. When playing against it, unorthodox moves can increase one's chances of winning. Those are moves that are not necessarily the best ones, and therefore lead to a game development that is unforeseen by the agent.

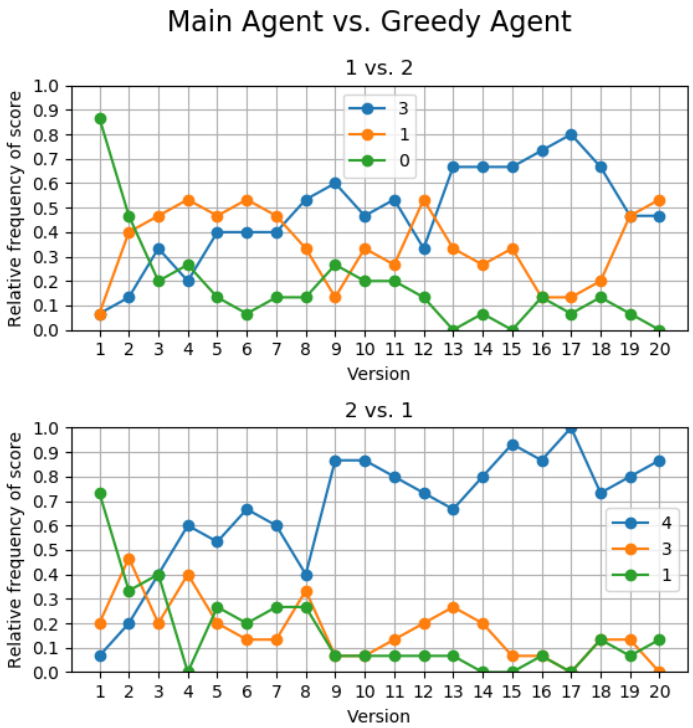


Figure 9.2.67: Main Agent vs. Greedy Agent

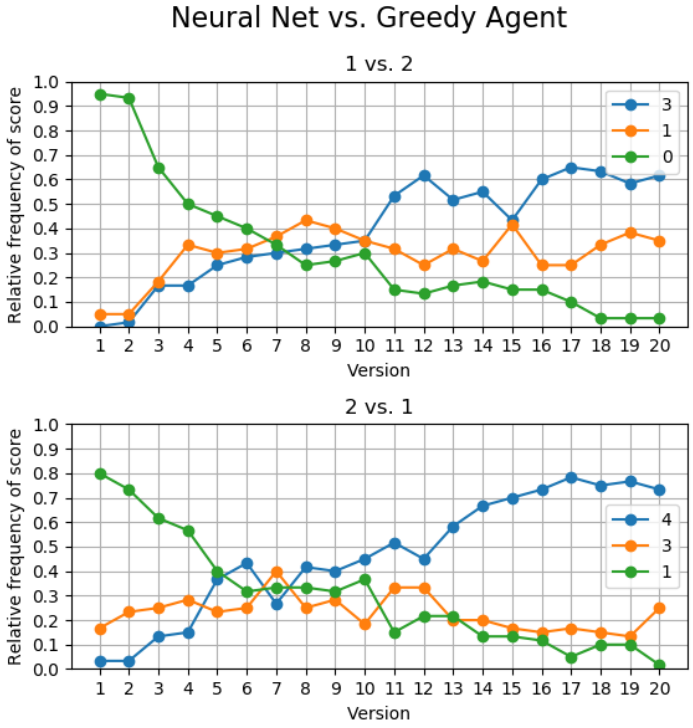
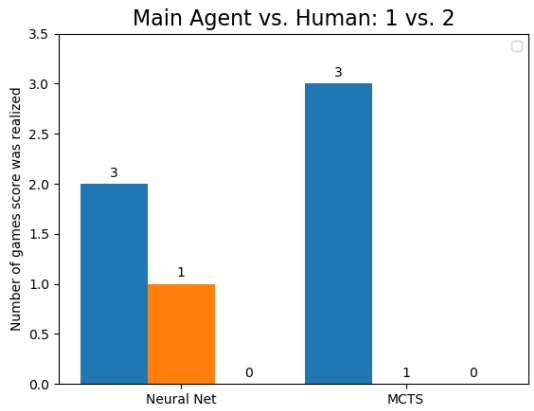


Figure 9.2.68: Neural Network vs. Greedy Agent

9.3. Evaluation against a Human Player

The Main Agent was also evaluated against two friends of mine, who are familiar with the standard version of Chinese Checkers.
They played a set of 6 games each against the Main Agent with and without the tree search.



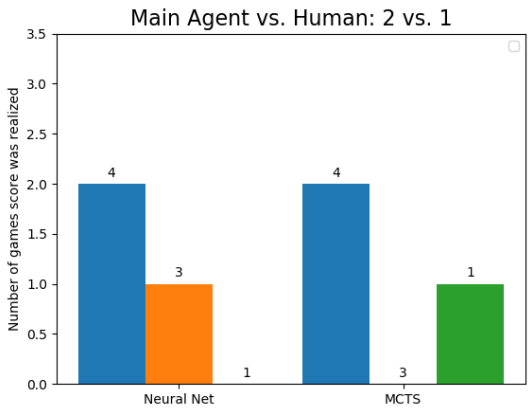


Figure 9.3.68: Main Agent vs. Human: 1 vs. 2 Figure 9.3.68: Main Agent vs. Human: 2 vs. 1

The Main Agent proved to be a strong opponent.
Surprisingly, the agent showed the worst performance in double games with the use of the tree search. That is the situation that led to the best results against the Greedy Agent.

9.4. Main Agent vs. Neural Network

We evaluate the Main Agent against the neural network without Monte-Carlo tree search. It can be observed that the Monte-Carlo tree search significantly contributes to the performance of the Main Agent. This assures that the exploration parameter is not too low, as discussed in subsection 7.3.

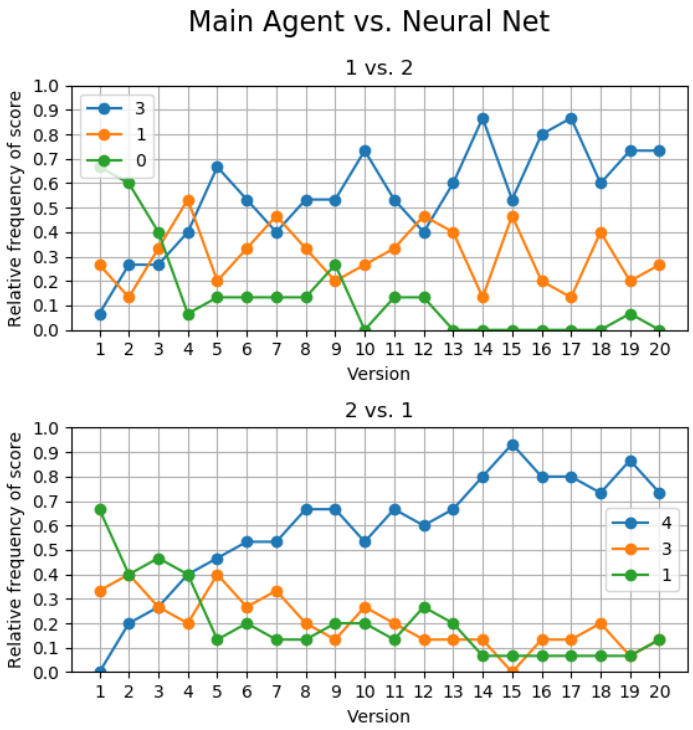


Figure 9.4.69: Main Agent vs. Neural Network

9.5. Evaluation of Illegal Moves

As already explained in section 3, the network also suggests invalid moves that get masked in the tree search. We want to investigate, what share of probabilities in π was assigned to illegal moves on average. We let each version of the network play 120 games against itself without the tree search and analyze the π-vectors returned by their corresponding networks.

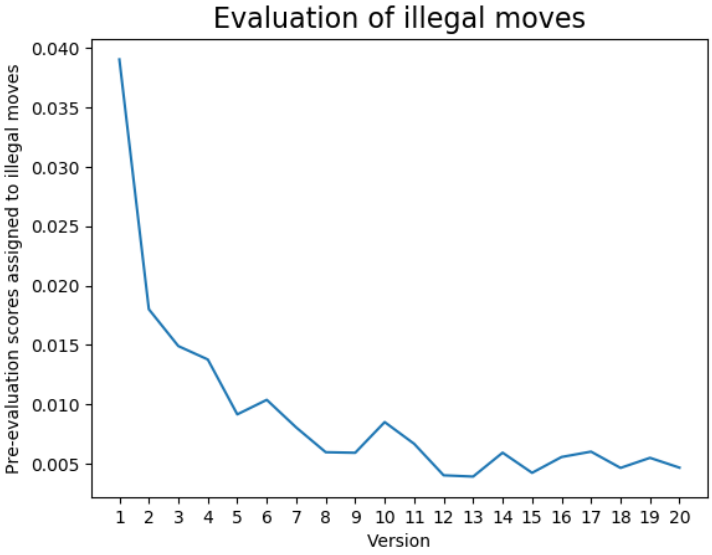


Figure 9.5.70: Pre-evaluation assigned to illegal moves

After the 500 self-play games of the Initialize Agent, the network only assigned less than 4% of the pre-evaluation to illegal moves. In the last version, illegal moves were only assigned 0.45%. It therefore can be said that the network learned to exclude illegal moves from its evaluation.

9.6. Starting order

In previous tests, agents were assigned to every combination of players. In this section, we will investigate, if one of the three players has a (dis-)advantage in the game. In 120 games played by instances of the Greedy Agent and 120 games played by the latest version of the Main Agent without the Monte-Carlo tree search, we analyze the scores of each player.

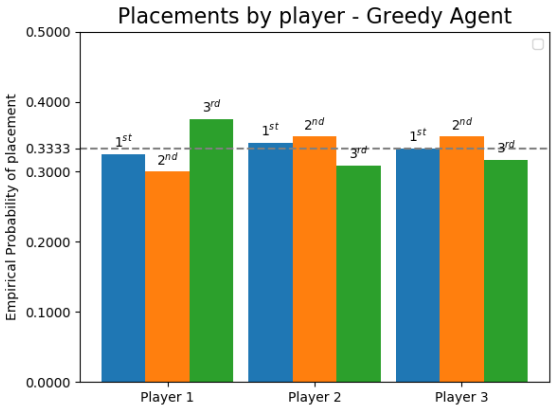


Figure 9.6.70: Placements of players - Greedy Agent

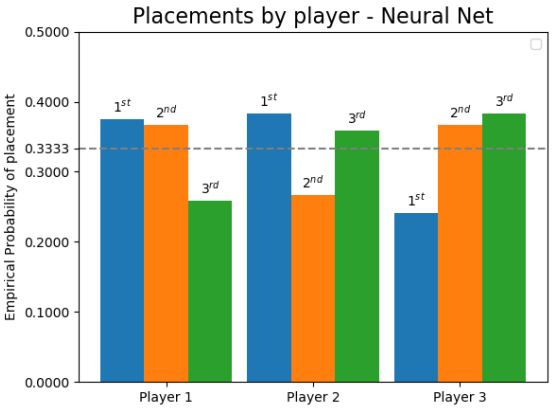


Figure 9.6.70: Placements of players - Neural Network

In a fair game, the placements should be distributed equally. In the games of the Greedy Agent,

player 1 received the worst scores on average. Apart from that, no significant bias can be seen. However, the results of the games played by the neural network show a distinct pattern. The order of the players correlates with their placement.
The state value that the network assigns to the initial board state leads to the same conclusion.

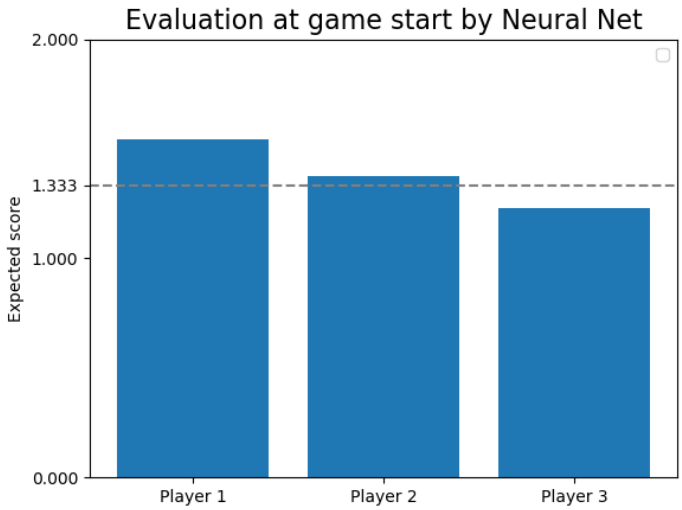


Figure 9.6.71: Evaluation of the game state before the first move

We can conclude that the Main Agent is strongest, when playing for player 1, and weakest, when playing for player 3. Therefore, either the game is unfair, or the Main Agent's tactics are more suitable for specific players.

9.7. Tactics

Our modified version of the game is not being played outside of this project. Therefore, we cannot compare the behavior of the Main Agent with tactics of experienced players.

We want to take a look at a board state, that the Main Agent encountered in every single game, the opening.

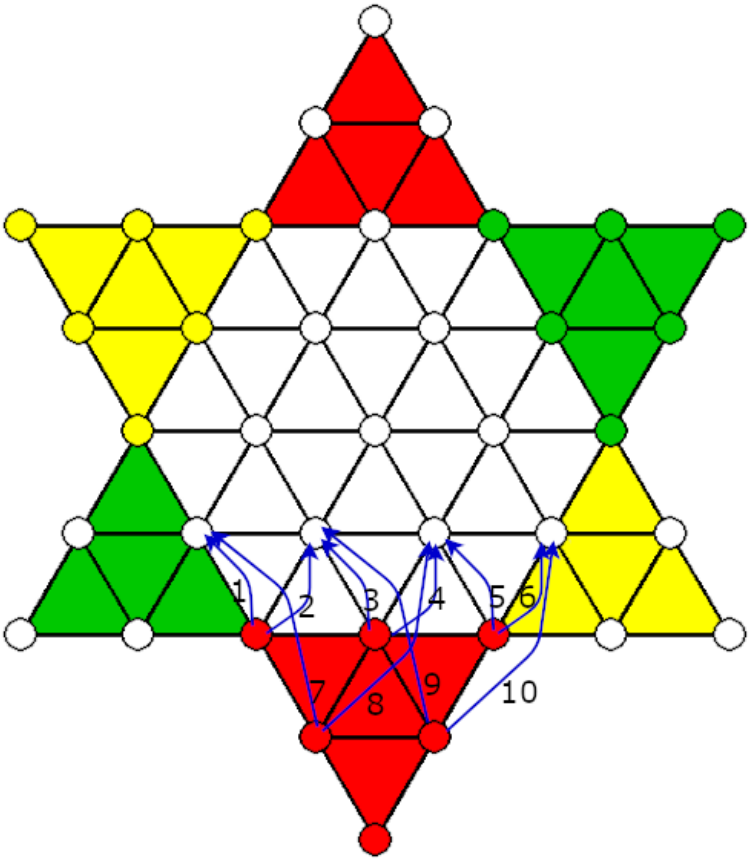


Figure 9.7.72: Opening moves

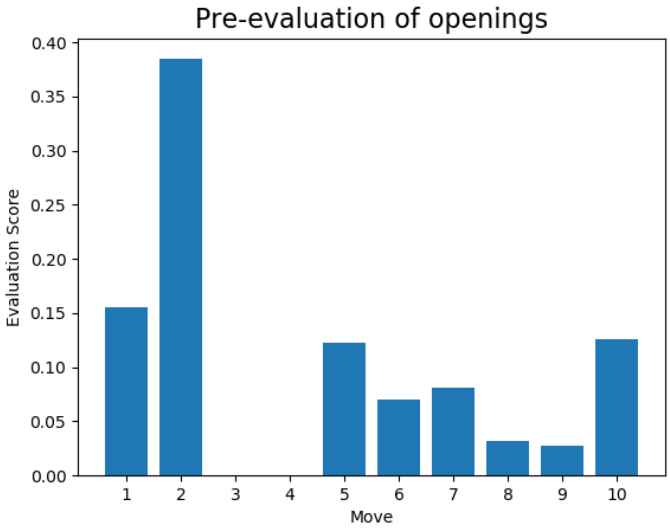


Figure 9.7.73: Pre-evaluation of opening moves

The network clearly suggests move 2 of Figure 9.7.72 as the best opening. It is interesting that it strictly advises against the two direct moves of the middle figure. With pre-evaluation scores

smaller than 10^{-7}, the tree search will not even try them once.

In the middle phase of the game, the Main Agent does not move its pieces forward as quickly as possible. Rather, it prevents other players from jumping, or blocks them completely. When the center of the board starts to clear, it changes its tactics and tries to quickly move its pieces to the end zone.

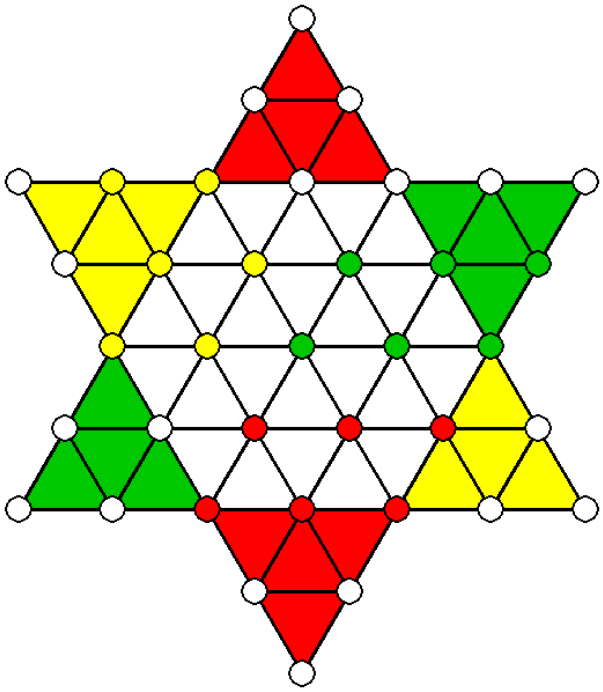


Figure 9.7.73: Example 1
The board is divided into territories by each player.

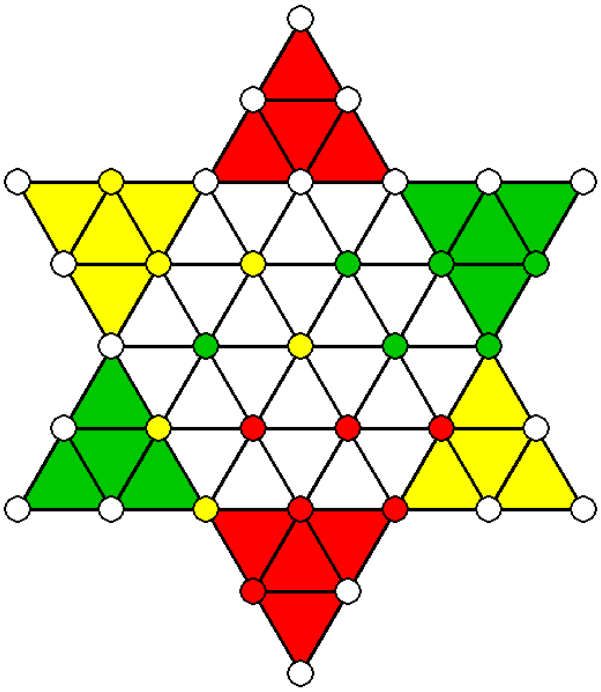


Figure 9.7.73: Example 2
Only player 3 can execute a jump move.

10. Conclusions and outlook

The realization of the project was a success. The concept of AlphaZero was adapted to Chinese Checkers. The results show that the Main Agent significantly improved during training.
There are several interesting questions that built upon the findings in this thesis.

In the results, we can see that the learning progress does not stagnate in later iterations. Given more time and resources, the training could be continued, until the best possible version of the agent is reached.

The Monte-Carlo tree search was successfully modified for a game with recurring states, and loops were prevented. Possibly, a decision algorithm that is designed for graphs from the beginning could improve the results.

Lastly, the project could be enhanced with more focus on cooperation between players. Currently, every player tries to optimize his score and ignores those of his opponents. An agent could be trained that plays in a team and tries to optimize the scores of his team. Another way to approach this question is to train an agent that does not only aims for the maximum score in one game, but in a set of games, in which the scores of each game are added together. Games are played until one player's score reach a specific threshold. In a scenario, where one player takes the lead, it would be reasonable for the other two players to cooperate, even at the expense of individual scores. This adds a lot of complexity to the game and requires more resources, but the dynamic of temporary cooperation promises an interesting research topic.

A. Notation

A	Action space
a	Action
c_{puct}	Exploration parameter in the Monte-Carlo tree search
$f(s)$	Neural network function assigning π and v to s
$f_v(s)$	Neural network function assigning v to s
f_π(s)	Neural network function assigning π to s
$g(v, p)$	Rating of a board state s for player p
$m_s(s, a, p)$	The board state resulting in the move a by player p in state s
$N_{s,a,p}$	Number of times, action a was taken by player p in state s in the tree search
$N_{s,p}$	Number of times, node (s, p) was visited in the tree search
n_B	Size of a training batch
n_b	Size of a mini-batch
n_{epoch}	Number of times the optimizer uses each training example in one training iteration
n_{eps}	Number of games played in one self-play phase
n_{varied}	Number of moves each that are chosen by varied selection
num_{sim}	Number of iterations in the Monte-Carlo tree search
P	Set of players
$P_{s,a,p}$	Pre-evaluation value of action a by player p in state s
p	Player
$r_s(s, p)$	Board rotation
$r_v(v, p)$	Permutation of board state
S	Set of board states
s	Board state
$U_{s,a,p}$	Upper confidence bound for action a by player p in state s
V	Set of state values
$V_{s,p}$	Node value for tree node (s, p)
v	State value
$Q_{s,a,p}$	
$\alpha(s, p)$	Legal moves for player p in board state s
$\phi(s, p)$	State value of board state s with current player p
$\rho(s, p)$	Function returning pre-evaluations for every move by player p in state s

References

[1] Dabbura I., (2017, December 21), Gradient Descent Algorithm and Its Variants. Retrieved from https://towardsdatascience.com/gradient-descent-algorithm-and-its-variants-10f652806a3

[2] He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 770-778).

[3] Ioffe, S., & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. arXiv preprint arXiv:1502.03167.

[4] Nair S., Alpha Zero General (2017), GitHub Repository, https://github.com/suragnair/alpha-zero-general

[5] Nogueira, N. (2006). A minimax tree example. Retrieved from https://en.wikipedia.org/wiki/Minimax

[6] Thakoor S., Nair S. & Jhunjhunwala M. *Learning to Play Othello Without Human Knowledge*

[7] Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., ... & Dieleman, S. (2016). Mastering the game of Go with deep neural networks and tree search. nature, 529(7587), 484.

[8] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., ... & Lillicrap, T. (2018). A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. Science, 362(6419), 1140-1144.

[9] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. The journal of machine learning research, 15(1), 1929-1958.

Eigenständigkeitserklärung (Declaration of Authorship)

Hiermit versichere ich, David Schulte, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, dass alle Stellen der Arbeit, die wörtlich oder sinngemäß aus anderen Quellen übernommen wurden, als solche kenntlich gemacht worden sind, und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt wurde. Köln, den 29. September 2019