Bachelor Thesis

Reinforcement Learning for 3-player Chinese Checkers

David Schulte

August 27, 2019

Mathematisches Institut
Mathematisch-Naturwissenschaftliche Fakultät
Universität zu Köln

Betreuung: Prof. Dr.-Ing. Gregor Gassner

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 make 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 the training of programs to make decisions. In the past, board games have been used to test training techniques and demonstrate the state-of-the-art.

The idea of a machine that could defeat humans in a board game, especially in chess, has sparked people's curiosity for several centuries. Because of the popularity and complexity of the game its players had a high reputation. The game seemed to be not solvable, so a chess playing machine would have to think like a human. (komischer Satz, umschreiben)
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. Just like a human would, the puppet moved its arm, moving the chess pieces. 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.
The idea of chess playing machines and algorithms was picked up again in the 21st century. 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. Lacking the sufficient hardware, their ideas stayed logical constructs without application.
Computer chess is also a common theme in science fiction, like in the 1968 science-fiction 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 viewer 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 chess 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 machine learning researchers worldwide. 2016 the program AlphaGo invented by the research company Google DeepMind won four out of five matches against 18-time world champion Lee Sedol. In December 2018 DeepMind published a paper in Science Magazine that describes AlphaZero and its evaluation. It is built upon AlphaGo but was generalized in such a way, that it was applicable to not only Go, but also also two-player games with perfect information. Another difference to its inferior predecessor was that the program learns these games by itself giving only the rules but no examples of games played by humans.

The research in this field is not only conducted to gain insights about these board games. These games are suitable training ground in Machine Learning. In contrast to most other real-life problems, they can easily be defined as a closed system and allow easy evaluation. They help illustrate
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 by Surag Nair used in his paper "Playing Othello without human knowledge". It was significantly modified to fit a 3-player game with possible repetitions in game states, using a different neural network and a modified tree search. I also enabled it to play several games in self-play silmultaneously.

section 2 will explain the rules of the game. section 3, section 4 and section 5 will present the components of the program and how they are combined. Lastly, the results of the training process will be presented in section 9.

!BEZUG AUF PROBLEMSTELLUNGEN AUSSERHALB VON GAME DOMAINS!

2 Chinese Checkers

Chinese Checkers is a modified version of 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. The board size was reduced from the standard 121 field board to 37 fields.

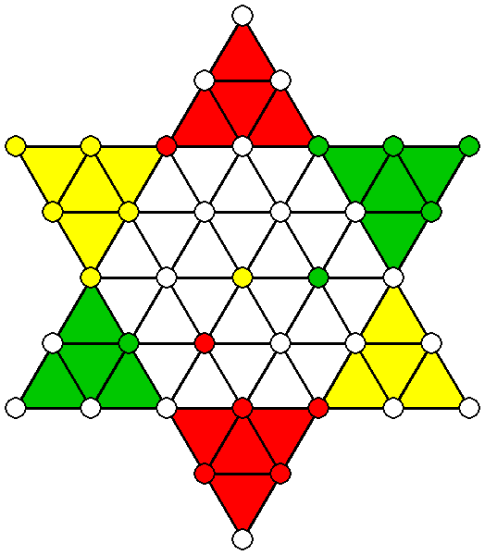

Figure 2.0.0: Board state after two moves each

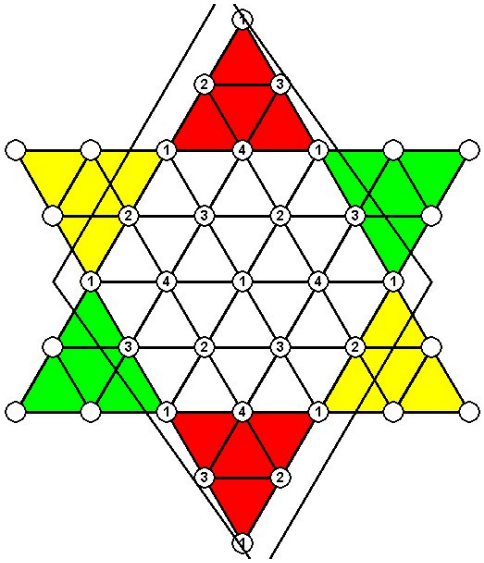

Figure 2.0.0: Grid NOCH KORRIGIEREN

2.1 Rules

Every player starts with 6 pieces, placed in one corner of the star-shaped board. The players take turns clock-wise. The goal is to move one's figures across the board to the end zone in the opposite corner before the other players manage to do so. After the winner is decided, the remaining two players continue to win the second place. It is allowed to move one's piece either to a free adjacent field or to jump over an occupied field if the space behind it is free. These jumping-moves can be chained, so that consecutive hops over a large area of the board are possible. Once placed in it a piece cannot move outside of its corresponding end zone. In this modified version, a player is not allowed to move his piece into enemy zones, with the exception of those fields that intersect with the neutral zone in the middle of the board. After a piece is moved it is the next player's turn.
We award the winner, the second place and the loser scores 3, 1 and 0 respectively. Furthermore we introduce a second-winner rule, that is not part of the classic game: While the winner has to move his own pieces into his end zone, it is just required to have a fully occupied end zone, possibly also by another player's pieces.

The game embodies offensive as well as defensive tactics. A player can move his 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 space.

2.2 Mathematical Description

This subsection will provide a mathematical description of our game. Even though a more compact description is possible, this one is used to stay close to the implementation of the game.

$S \subset \{0,1,2,3,4\}^{9\times 9}$ is the set of all possible board configurations. In a board state $B \in S$ the value of $B_{i,j}$ describes the state of the field in the corresponding position. 0 denotes an empty field. 1,2 and 3 denote a field occupied by the respective player. 4 denotes fields that are unreachable, because the lie outside of the actual board.
There are 37 field on the board and every of the three players has 6 pieces. The board is represented by a 9x9 array.

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

	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.
We describe the moves out of perspective from player one.
The algorithm requires a vector with each move in the game corresponding to an entry. This action vector is logically split into two parts with the first part representing direct steps and the second one representing (consecutive) jumps.

Every of the 25 accessible gets assigned 6 step directions. This results in 150 moves.
To encode jumps the board is subdivided into four grids of different sizes. A sequence of jumps starting in a grid has to also 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 a every figure of a player is blocked, or when a player has won the game and is waiting for his opponents to determine the second place. Thus the action vector contains 320 entries. $|A| = 320$.

It should be noted that this action vector 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(25) are included (25). Direct moves (8) as well as jumping sequences that lead out of the end zones ($3*6+2*1*5+1*3 = 31$) 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 vector, 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.
$m : S \times A \times P \to S \times P$ returns the next board state given a board state and an action by a specific player.
$m_s : S \times A \times P \to S$ returns the next board state given a current board state, a action and the player that executes it.
$m_p : S \times A \times P \to P$ returns the next player given a current board state, a action and the player that executes it.
$\phi : S \to V$ assigns every board state a state value.
$g : V \times P \to \mathbb{R}$ return the benefit of board a value for a specific player p, For the following implementation, it is assumed that g is linear with respect ot its first argument.
$t : S \to \{0, 1\}$ returns 1, if the game is over and 0 if it is not.

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 to convey the general methodology of tree search algorithms for decision making. Afterwards we will take a look at the Monte-Carlo tree search and see, why it is the algorithm that was used in this project.
For now we will assume that no game loops are possible, meaning that every board state can only occur once in one game iteration. 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

Minimax and the Monte-Carlo tree search both gradually build game trees. The tree representation of the game is following. A board state and current player $(s, p) \in S \times P$ is denoted by a tree node. The two nodes (s, p) and (s', p') can be connected by an edge if there is an action $a \in \tilde{a}(s, p)$ such that $m_s(s, a, p) = s'$ and $m_p(s, a, p) = p'$.
Given a board state s and the current player p, these algorithms 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^* that maximizes $g(v(s^*), p)$.
It is assumed that the evaluation function g is known for each player, and that every player acts by the same policy.

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. $\tilde{V} = \mathbb{Z}$
$\tilde{P} = \{1, 2\}$
$\tilde{A}$ with $|\tilde{A}| = 2$.

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

While this assumption enables a 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 with the current game state and player (s, p) as root node. The tree contains every possible node down to a specified maximal depth. Since both players are taking turns, the decision maker alternates between each level of the tree. Leaf notes are in the deepest layer of the tree or represent terminal game states from which no further actions are possible. 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 parent's level 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.

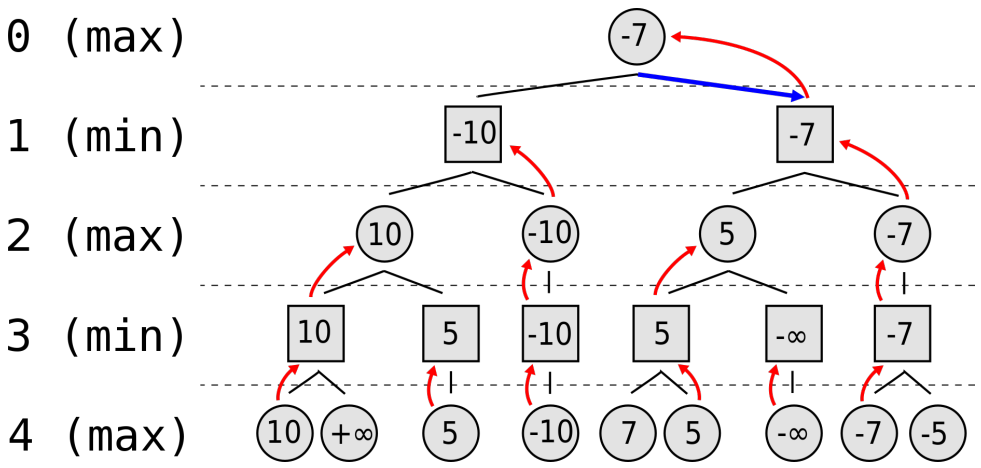

Figure 3.1.2: Minimax Tree
In this tree both players have two available actions for every state. The state values get back-propagated in a Depth First Search. The result is that the max-player will choose the action corresponding to the right edge of the root node.

Algorithm 1 Minimax

```
1:  function MINIMAX(s, p, depth, depth_max)
2:      if t(s) = 1 or depth = max_depth then
3:          v ← φ(s)
4:          return v, 0
5:      g_best ← −∞
6:      a_best ← 0
7:      for each a ∈ α(s, p) do
8:          s' ← m_s(s, a, p)
9:          p' ← m_p(s, a, p)
10:         v', _ ← minimax(s', p', depth + 1, depth_max)
11:         g ← g(v', p')
12:         if g > g_best then
13:             g_best ← g
14:             v_best ← v'
15:             a_best ← a
16:     return v_best, a_best
```

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:

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

This makes Minimax unsuitable for our problem. Although there are techniques like Alpha-Beta-pruning to skip the evaluation of unpromising nodes, modern Machine Learning programs mostly work with another algorithm.

3.2 Monte-Carlo tree search

The Monte-Carlo tree search 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. To do so, 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 that represents the current game state, a path navigating through the tree is selected until a leaf node is reached.
The selection of actions that define paths through the tree is determined by the following formula:

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

$Q_{s,a,p}$ is a evaluation of the profitability of a for player p in state s. It is initialized as 0.
$P_{s,a,p}$ is a pre-evaluation of the action a in state s for player p. The values of P should be taken as given for now. They are outputs of the neural networks, as we will discuss in the next chapters.
$N_{s,a,p}$ is the number of times action a was chosen by player p in state s.
$N_{s,p}$: The number of times that player p chose an action in state s. It is the sum of values $N_{s,a,p}$ over all $a \in A$.
$U_{s,a,p}$ is the upper confidence bound of action a at state s. It describes how important a exploration in direction of the different edges of a node is.
c_{puct} is the exploration parameter. It will be put into context in the next paragraph.

From all possible actions the one maximizing U is chosen.

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

Expansion

The leaf node s' gets evaluated by the state evaluation function.

$$s' = m_s(s, a, p) \tag{3.2.6}$$

$$V_{s',p'} = \phi(s') \tag{3.2.7}$$

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

Back-propagation

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

$$V_{s,p} = \begin{cases} \phi(s) & \text{if } N_{s,p} = 0 \\ \frac{\sum_{a \in A(s)} V_{m_s(s,a,p)} \cdot N_{s,a,p}}{N_{s,p}} & \text{else} \end{cases} \tag{3.2.8}$$

$$Q_{s,a,p} = \begin{cases} g(V_{m_s(s,a,p)}, p) & \text{if } N_{s,a,p} > 0 \\ 0 & \text{else} \end{cases} \tag{3.2.9}$$

Equation 3.2.4 showcases 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 lead to a state with positive reward are investigated further.
The second term describes optimism regarding Exploitation. This term grows everytime, 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 best choice, but a vector with values for every action in A. Those values are proportional to the number of times that the corresponding edge starting at the root node was traversed.

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

Algorithm 2 search

function SEARCH(s)
  if s is a terminal node **then**
    return $eval(s)$
  if s is a leaf node **then**
    $v, p = nn(s)$
    for each a in $A(s)$ **do**
      $N(s,a) \leftarrow 0$
      $Q(s,a) \leftarrow 0$
    return v
  $u_{best} \leftarrow -\infty$
  $a_{best} \leftarrow 0$
  for each a in $A(s)$ **do**
    $u \leftarrow Q(s,a) + c_{puct} * P(s,a) * \frac{\sqrt{N(s)}}{1+N(s,a)}$
    if $u > u_{best}$ **then**
      $u_{best} \leftarrow u$
      $a_{best} \leftarrow a$
  $s' \leftarrow m_s(s,a,p)$
  $p' \leftarrow m_p(s,a,p)$
  $v \leftarrow search(s', p')$
  $Q(s,a) \leftarrow \frac{N_{s,a,p} * Q_{s,a,p} + g(v,p)}{N(s,a,p)+1}$
  $N_{s,a,p} \leftarrow N_{s,a,p} + 1$
  $N_{s,p} \leftarrow N_{s,p} + 1$
  return v

The second function calls the search-function multiple times, constructing a tree. Then it counts how often each edge starting at the root node was chosen. It constructs a normalized vector C containing these counts. The higher the value of C_a for $a \in A$ is, the better it is scored by the tree search. In section 5 we will have a more detailed look at how it is used to choose an action.

Algorithm 3 mcts

function GETACTIONCOUNT(board, player, numSimulations)
  for $i \leftarrow 1$ to $numSimulation$ **do**
    search(board, player)
  for each a in A **do**
    $C(a) = N(s, a, player)$
  $C \leftarrow \frac{C}{len(C)}$
  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 a 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 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 his problem, the values would be back-propagated in circles and the skewed. To prevent this, we back-propagate not v, but the actual score $\tilde{v}$ of the state. $\tilde{v}$ does not described 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. So 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 in the tree. Loops in actual game-play can not be prevented this way. subsection 7.1 further addresses this problem and how it was handled.

4 Neural Net

In this section we will take a look at the neural network and how it is trained.
A 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. The activation functions are needed. Without them the linear functions would simplify into one linear function. This would limit the complexity of the network quickly.
The functions of the network contain parameters that we denote with the vector $w \in W \subset \mathbb{R}^n$. Therefore the output depends on the function input and those parameters. Training the network means changing its 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. 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-estimate probability vector π and the board value v.
We denote these two predictions as following functions:

$$f_\pi : S \times W \to [0,1]^{|A|} \tag{4.0.10}$$

$$f_v : S \times W \to V \tag{4.0.11}$$

Combined they are formulated as following function.

$$f : S \times W \to [0,1]^{|A|} \times V \tag{4.0.12}$$

4.1 Network architecture

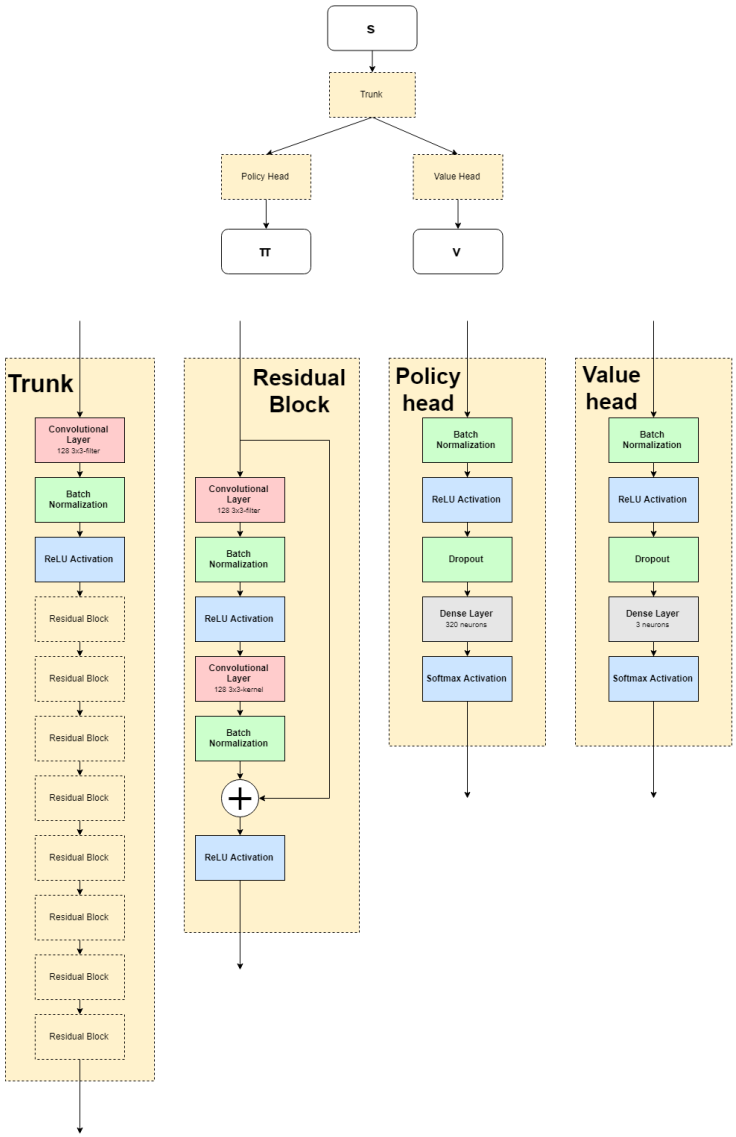

Figure 4.1.13: PLACEHOLDER Neural Network Architecture

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 investigate the layers that the network is built of.

4.1.1 Dense layer

Dense layers are the most basic layers used in neural networks. They take the signals of a previous layer as input and return linear combinations of them. Inspired by biology the entries of its output are called neurons.

$$d : \mathbb{R}^n \times \mathbb{R}^{n*m+m} \to \mathbb{R}^m$$
$$(x, w) \mapsto \begin{pmatrix} x_1 \cdot w_1 + ... + x_n \cdot w_n + w_{n*m+1} \\ \vdots \\ x_1 \cdot w_{n*(m-1)+1} + ... + x_1 \cdot w_{n*m} + w_{n*m+m} \end{pmatrix} \tag{4.1.14}$$

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

4.1.2 Convolutional Layer

Convolutional neural networks are often used for pattern detection. They analyze transform multidimensional arrays, mostly 2-dimensional ones. For example, a convolutional layer can detect edges and corners in an image, and several layers in combination are able to detect complex patterns. We use convolutional layers to detect abstract patterns in the game, like ladders and blocks of figures.
One layer contains a number of filters with an uneven kernel size k. These filters are $k \times k$ matrices. We slide them across the input matrix and multiply the overlaying indices. The corresponding output to the center of the current input window is the sum the of these products.

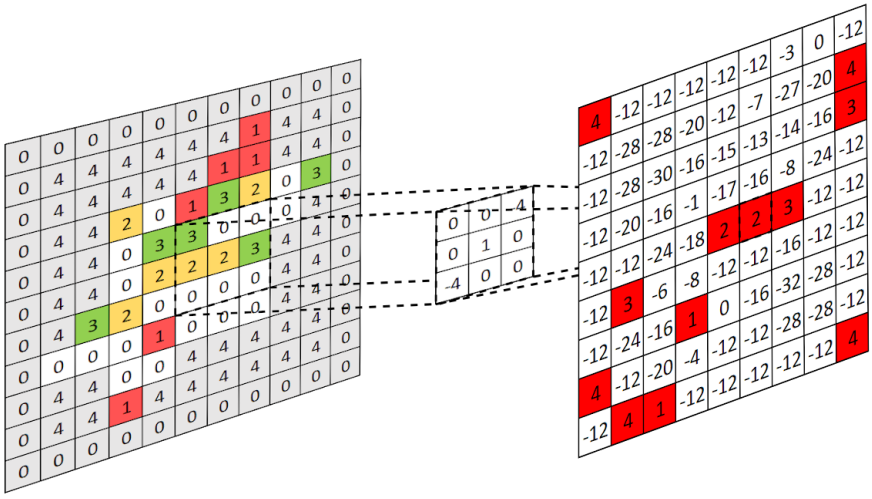

Figure 4.1.15: Convolutional Layer
This image shows an example filter applied to the board state. It detects fields than can be jumped over in the upper right or lower left direction. The positions that have positive indices and do not lie on the border indicate these fields.

$$\tilde{k} := \frac{k-1}{2} \tag{4.1.16}$$

$$\begin{aligned} C : \mathbb{R}^{n_1 \times n_2} \times \mathbb{R}^{k \times k} &\to \mathbb{R}^{n_1 \times n_2} \\ (X, F) &\mapsto Y \end{aligned} \tag{4.1.17}$$

$$\tilde{x}_{i,j} := \begin{cases} X_{i,j} & \text{if } 1 \leq i \leq n_1 \text{ and } 1 \leq j \leq n_2 \\ 0 & \text{else} \end{cases} \quad \text{for } i = -\tilde{k}+1, ..., n_1 + \tilde{k}; j = -\tilde{k}+1, ..., n_2 + \tilde{k} \tag{4.1.18}$$

$$Y_{i,j} = \sum_{l=1}^{k} \sum_{m=1}^{k} F_{k,m} * \tilde{x}_{i-\tilde{k}+l, j-\tilde{k}+m} \tag{4.1.19}$$

The entries of each filter matrix are learnable parameters.

4.1.3 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.

$$\begin{aligned} g : \mathbb{R}^n &\to \mathbb{R}^n \\ x &\mapsto \begin{pmatrix} \max(0, x_1) \\ \vdots \\ \max(0, x_n) \end{pmatrix} \end{aligned} \tag{4.1.20}$$

4.1.4 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. That is the reason why it is usually applied as an activation for outputs that represent probability distributions.

$$\begin{aligned} g : \mathbb{R}^n &\to \mathbb{R}^n \\ x &\mapsto \frac{1}{\sum_{i=1}^{n} \exp(x_i)} \begin{pmatrix} \exp(x_1) \\ \vdots \\ \exp(x_n) \end{pmatrix} \end{aligned} \tag{4.1.21}$$

4.1.5 Batch Normalization

!ERKLAERUNG!

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i \tag{4.1.22}$$

$$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2 \tag{4.1.23}$$

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \tag{4.1.24}$$

$$y_i = \gamma\hat{x}_i + \beta \tag{4.1.25}$$

4.1.6 Dropout

Dropout is another method to regularize the neural network. A dropout layer ignores a randomly sampled fraction of the inputs in every iteration. This prevents one single input from having too much effect on the output.

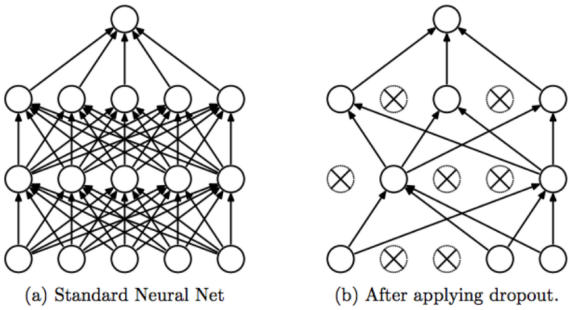

Figure 4.1.26: Network using dropout
This figures shows a network with dense layers and an output. In figure (b) a dropout layer is inserted between each layer. The crossed out neurons represent the ones, that are ignored in the next layer in one specific iteration.

4.1.7 Residual Block

The use of residual blocks makes our network a residual neural network. In the residual blocks there is a skip connection implemented. Through this connection the input of the residual block gets added to the processed data just before the activation function of the block. From a certain point on, neural networks without skip connections tend to stagnate or even decline in quality with every layer that is added. This problem is discussed in detail in this article:

!QUELLE!

4.2 Loss Function

The loss function describes the error of a prediction given labeled outputs. 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.

$$
\begin{aligned}
l_\pi : S \times A \times W &\to \mathbb{R} \\
(s, \hat{pi}, w) &\mapsto -\sum_{i=1}^{920} \hat{\pi}_i \log(f_\pi(s)_i)
\end{aligned}
\tag{4.2.27}
$$

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

$$
\begin{aligned}
l_v : S \times V \times W &\to \mathbb{R} \\
(s, \hat{v}, w) &\mapsto \frac{1}{3} \sum_{i=1}^{3} (f_v(s, w)_i - \hat{v}_i)^2
\end{aligned}
\tag{4.2.28}
$$

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

$$
\begin{aligned}
l : S \times V \times A \times W &\to \mathbb{R} \\
(s, \hat{v}, \hat{\pi}, w) &\mapsto l_\pi(s, \hat{\pi}, w) + l_v(s, \hat{v}, w)
\end{aligned}
\tag{4.2.29}
$$

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.

$$
\begin{aligned}
l_w : W &\to \mathbb{R} \\
w &\mapsto l(s, \hat{\pi}, \hat{v}, w)
\end{aligned}
\tag{4.2.30}
$$

We generally don't train on a single tuple, but on a large number of labeled training tuples at the same time. Given the training batch $b \in (S \times A \times V)^{n_t}$ we can formulate the loss function that we want to minimize.

$$
\begin{aligned}
L_b : W &\to \mathbb{R} \\
w &\mapsto \sum_{i=1}^{n} l_w(s_i, \hat{\pi}_i, \hat{v}_i)
\end{aligned}
\tag{4.2.31}
$$

4.3 Optimizers

During the training process an optimizer redefines weights 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 initial guess and (not always successfully) go towards the minimum of a function.
In this case the Adam Optimizer was chosen. The following section will explain Gradient Descent as its basis. Afterwards, different ideas to increase efficiency 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 Jacobian matrix J of F.

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

$$\mathbf{J}\colon \mathbb{R}^n \to \mathbb{R} x \longmapsto \begin{pmatrix} \frac{\partial F}{\partial x_1} \\ \vdots \\ \frac{\partial F}{\partial x_n} \end{pmatrix} \tag{4.3.32}$$

4.3.1 Gradient Descent

Gradient Descent is a numerical iterative algorithm used to find the minimum of a function. Given a 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 $\alpha \in \mathbb{R}_{>0}$. The number of iterations that are executed is called n_{epoch} in the context of machine learning.

$$x^{(i)} = x^{(i-1)} - \alpha * J(x^{(i-1)}) \quad \text{for } i = 1, ..., n_{epoch} \tag{4.3.33}$$

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 to the loss function with motivation to find a local minimum resulting in loss not much higher than the global minimum.
The right choice for α depends on the problem. 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.
Gradient descent is applied to the loss function L_b to find parameters $w^* \in W$ that minimize the loss.

4.3.2 Mini-Batch

When we train a neural network we feed it a large amount of labeled data. Therefore the computation of each gradient is computational expensive. The idea of Mini-Batch is to split the training set into batches of size n_b and execute each gradient descent step optimizing just over one 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 greatly differing from L_b. In theory however these inaccuracies cancel each other out.
We do not iterate over $L_b(w)$, but over the functions $L_{b_i}(w)$.

$$L_{b_i}(w) := \sum_{j=(i-1)*b+1}^{\min\{i*b, n_t\}} l_w(s_j, \hat{v}_j, \hat{\pi}_j, w) \text{ for } i = 1, ..., \left\lceil \frac{n_t}{b} \right\rceil \tag{4.3.34}$$

$$J_{b_i}(x) := \begin{pmatrix} \frac{\partial L_{b_i}}{\partial x_1} \\ \vdots \\ \frac{\partial L_{b_i}}{\partial x_n} \end{pmatrix} \tag{4.3.35}$$

$$x^{(i)} = x^{(i-1)} - \alpha * J_{b_i}(x^{(i-1)}) \quad \text{for } i = 1, ..., b \cdot n_{epoch} \tag{4.3.36}$$

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 method using Batch size 1 is called Stochastic Gradient Descent.

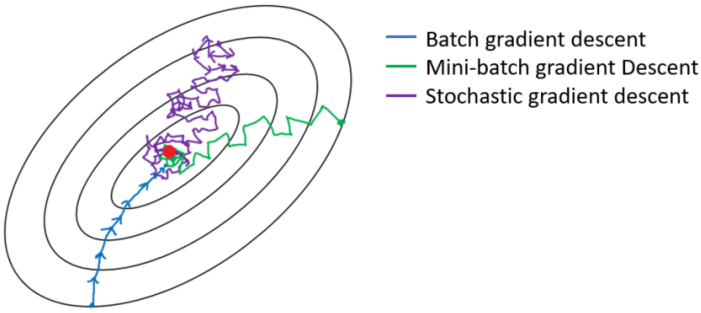

Figure 4.3.37: Mini-batch gradient descent
A 2-dimensional function is minimized with standard gradient descent, Mini-batch gradient descent and stochastic gradient descent. To illustrate this better, different starting points were used.

4.3.3 Momentum

The gradients computed in every step of Mini-batch gradient descend can change directions rapidly, while often alternating around the ideal directional vector leading towards a local minimum. Gradient Descent with momentum takes advantage of that. Just like in physics, were 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 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.

$$M^{(0)} = 0 \tag{4.3.38}$$

$$M^{(i)} = \beta_M * M^{(i-1)} + (1 - \beta_M) * J(x^{(i-1)}) \tag{4.3.39}$$

$$\tilde{M}^{(i)} = \frac{M^{(i)}}{1 - \beta_M^i} \tag{4.3.40}$$

$$x^{(i)} = x^{(i-1)} - \tilde{M}^{(i)} \quad \text{for } i = 1, ..., n_{epoch} \tag{4.3.41}$$

Formula 4.3.40 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.

4.3.4 RMSProp

Another method correcting 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 decide, how far to move in each direction, RMSProp uses a different learning rate for each weight respectively with more respect to the 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.

$$s_j^{(i)} = \beta * s_j^{(i-1)} + (1-\beta) * (\frac{\partial F}{\partial x_j}(x^{(i-1)}))^2 \tag{4.3.42}$$

$$x^{(i)} = x^{(i-1)} - \left(\frac{\alpha}{\sqrt{s_1+\epsilon}} \cdots \frac{\alpha}{\sqrt{s_m+\epsilon}}\right) * J(x^{(i-1)}) \tag{4.3.43}$$

4.3.5 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.

$$M^{(i)} = \beta_M * M^{(i-1)} + (1-\beta_M) * J_{b_i}(x^{(i-1)}) \tag{4.3.44}$$

$$\tilde{M}^{(i)} = \frac{M^{(i)}}{1-\beta_M^i} \tag{4.3.45}$$

$$s_j^{(i)} = \beta * s_j^{(i-1)} + (1-\beta) * (\frac{\partial F}{\partial x_j}(x^{(i-1)}))^2 \tag{4.3.46}$$

$$x^{(i)} = x^{(i-1)} - \left(\frac{\alpha}{\sqrt{s_1^{(i)}+\epsilon}} \cdots \frac{\alpha}{\sqrt{s_m^{(i)}+\epsilon}}\right) * \tilde{M}^{(i)} \quad \text{for } i = 1, ..., n_{epoch} \tag{4.3.47}$$

Adam Optimizer has proven to be a very good off-the-shelf optimizer for Neural Networks.

5 Actors

After explaining the two biggest components of the program, the Monte-Carlo tree search and the neural network, we will now see how they are combined.
Just like the other games that AlphaZero was applied to (Go, Chess, Shogi, Othello), Chinese Checkers is a symmetric game. This means that every player has the same action space and same objective. Therefore we do not have to train a separate network for every player in the game, but rather a single one, that can decide between moves in the role of player one.
Before every decision we rotate the board in such a way that the current player takes the perspective of the player with red figures. This is done by a 120° degree turn of the board. We define following functions:

$$r_s : S \times P \mapsto S \tag{5.0.48}$$

$$r_v : V \times P \mapsto V \tag{5.0.49}$$

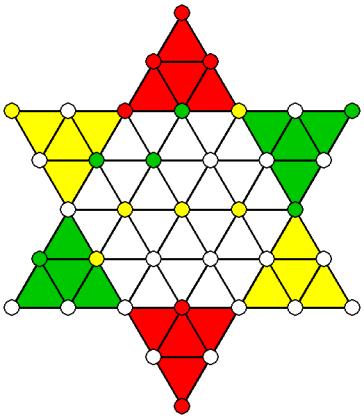

Figure 5.0.49: $r_s(s, 1)$

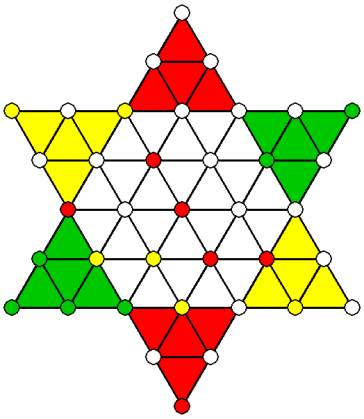

Figure 5.0.49: $r_s(s, 2)$

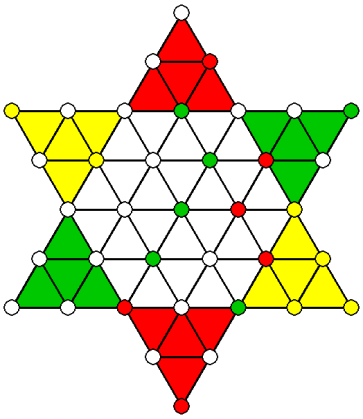

Figure 5.0.49: $r_s(s, 3)$

The decision is always made out of perspective of player one. This does not hurt generality because the board can be rotated in such a way, that the other players takes player one's perspective. This rotation has to take place every time turns are taking, including the look-ahead simulation inside the tree search.
We call the entity, that selects a move given a board state and the current play, an actor. This project contains different actors.

5.1 NN Actor

The NN Actor consists of a Monte-Carlo tree search and a neural network. gets it's state evaluation and probability from a neural network. It is the one that is trained in this project.
It uses the probability vector C obtained from the MCTS-algorithm 3.2.
There are two ways to use the vector.
We can choose the action whose edges in the last MCTS-iterations has the most count. If there are several edges from the root node that have maximal counts, we randomly select one of them.

$$A^* := \arg\max_{a \in A} C_a \tag{5.1.50}$$

$$P(a) = \begin{cases} \frac{1}{|A^*|} & \text{if } a \in A^* \\ 0 & \text{else} \end{cases} \tag{5.1.51}$$

One of the edges with the most visits gets selected. 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 actors, 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.

$$P(a) = C_a \quad \text{for all } a \in A \tag{5.1.52}$$

This will be referred to as varied selection.
Both selection methods are being used in the program.

5.2 Greedy Actor

To provide an analytic evaluation of a move, we introduce progress. We divide the board into rows and record how many rows the figure moved upwards.

$$e : A \mapsto \mathbb{Z} \tag{5.2.53}$$

The greedy actor always chooses one of the moves with highest progress. It is the most simple plausible game strategy.

$$A_G = \arg\max_{a \in \tilde{a}(s,1)} e(a) \tag{5.2.54}$$

$$P(a) = \begin{cases} \frac{1}{|A_G|} & \text{if } a \in A_G \\ 0 & \text{else} \end{cases} \tag{5.2.55}$$

The Greedy actor is used as a benchmark for the performance of our Neural Net Actor.

5.3 Initialize Actor

The Initialize Actor is used to initialize our neural network.
It chooses random moves, but a has high tendency to move figures 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 neutral move to the side. We construct the vector $I \in \mathbb{N}_0^a$ with:

$$I_a = \begin{cases} 1 & \text{if } a \in \tilde{a}(s,1) \text{ and } e(a) < 0 \\ 2 & \text{if } a \in \tilde{a}(s,1) \text{ and } e(a) = 0 \\ 4 & \text{if } a \in \tilde{a}(s,1) \text{ and } e(a) > 0 \\ 0 & \text{else} \end{cases} \tag{5.3.56}$$

$$P(a) = \frac{I_a}{||I||_1} \quad \text{for all } a \in A \tag{5.3.57}$$

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 game agent.

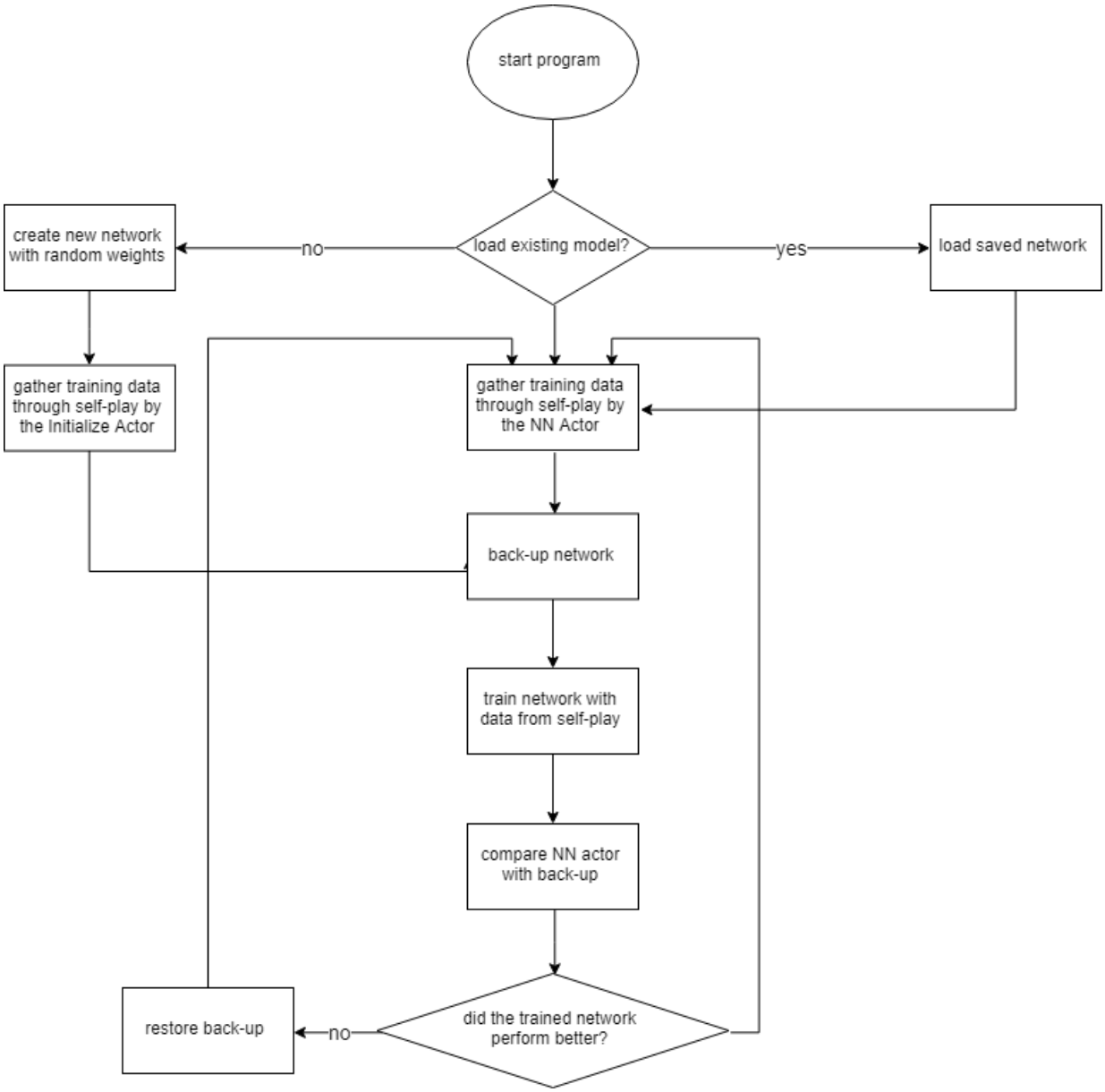

Figure 6.0.58: PLACEHOLDER Flowchart of the training loop

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 actor, 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 $(s, C, p) \in S \times \mathbb{R}^{|A|} \times P$ 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 decided, the final state is evaluated to state value v^*. For every tuple (s, C, p) that we collected, we create a corresponding tuple $(r(s,p), C, r(v^*,p)) \in S \times \mathbb{R}^{|A|} \times V$. 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 Actor is being used. Otherwise the Actor corresponds to the current version of the neural network.

Back-up

We create a back-up copy of the current neural network. If the network does not improve through the training, we can restore the version before training.

Training

We use the training data of maximal the last 10 self-play sessions to train the network, provided they were generated by the NN actor. 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. Of course π and v will not only be altered for input s, but every board state, that the network detects specific similarities in.

If this is the first iteration and the network is being initialized by the Initialize Actor, we use the training examples in this first iteration 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 actors 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 network will play for 2 and which one will play for 1 player. We also rotate between the players, Therefore a set consists of 6 games. We play a fixed number of sets and add up the points scored for 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 difficulties showed up. This chapter will explain them and present their solutions.

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 to gather training examples. These counter-measures help against game loops, but not against quasi-loops. 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. An example. These quasi-loops are especially conducted by players, that are steps away from losing the game, preferring to drag the game over losing.

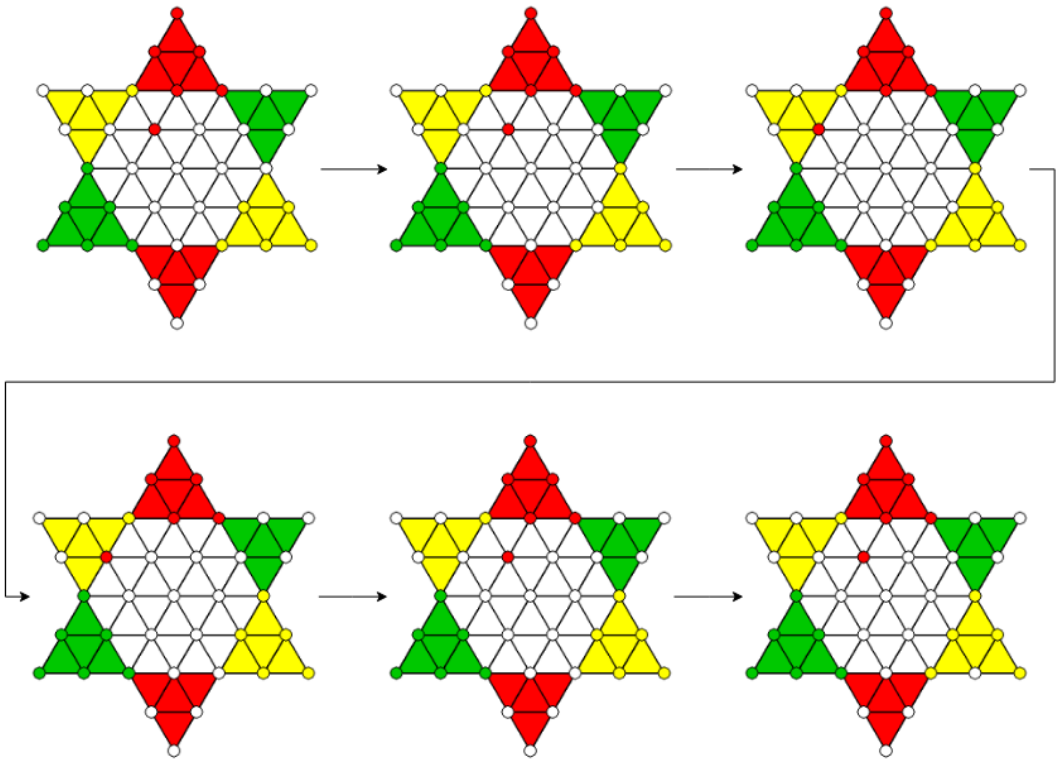

Figure 7.1.59: Example of a quasi-loop before the introduction of the second-winner rule
In the first board state it is the turn of player two. Knowing that a moving his figure out of player one's end zone would result in a loss, he just moves his pieces around.

After the quasi-loop is played a specific number of times, all variations of board states. 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 change the situation to his disadvantage to prevent an actual game loop. 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.
For the case, that all these measurements 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 zero. The actors do not have knowledge about how many steps were taken in game and therefore will not change their behavior. However, after the training process the board states that led to the situation will be evaluated lower.

7.2 Triple Win

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

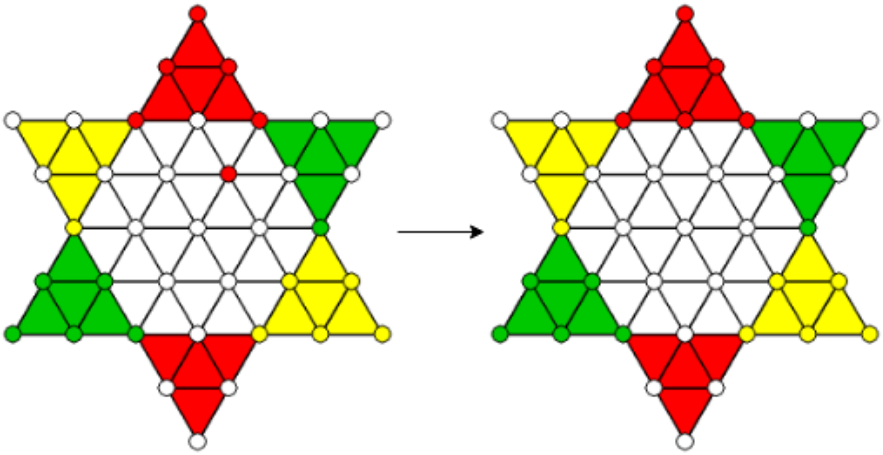

Figure 7.2.60: PLACEHOLDER Flowchart of the training loop

7.3 cpuct

!VERLINKUNG!
The right choice for the exploration parameter c_{puct} is one of the most crucial factors determining the quality of the neural network and therefore the actor. In **??** the mechanics of 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 distinctive and the tree search will focus even more on the paths that were previously favored, neglecting the unpromising ones. In a extreme consequence, the distribution of π can become so distinct, that the tree search does not contribute to the decision at all, because from the root node only the path suggested by the network is explored. On 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 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 use of the agent. One can also change c_{puct} into the opposite extreme for several training iterations with the goal of balancing out the distribution of π. The best solution is to experiment until a exploration parameter is found, and restart the training process using it in all iterations. This however is the most time- and resource-consuming.

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. That 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 network versions. Because of the recurring states in Chinese Checkers a game, consisting of randomly chosen moves, is very long. A figure can be moved just outside of the end zones and then back across the board, requiring many turns. Therefore the first iterations of self-play would be too time-consuming. The Initialize Actor was designed to nudge the network in the right direction without influencing it to much.

The most costly 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 silmultaneously. A more detailed description can be found in the next chapter.

8 Implementation

The project builds upon a open-source project I found on Github. This project by Surag Nair implements the training process described in the chapter ??? and is a simplification of the structure used in AlphaZero. The game used is Othello, but it is generalized in such a way that it can be used for other two-player board as well. The programming language used is 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 altered to allow 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 close 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 silmultaneously.

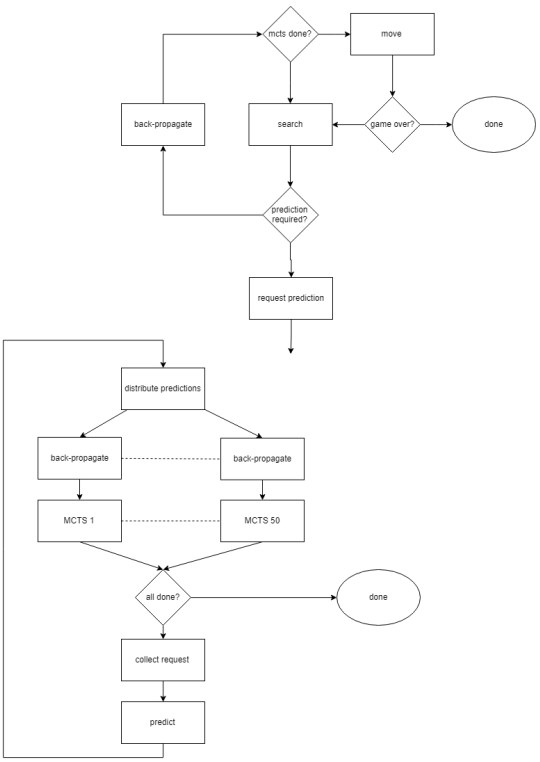

Figure 8.0.61: PLACEHOLDER

In every game instance the actors 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 fed in the network. Then 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 exploration and the back-propagation were separated into two functions. Because the path that traversed in the tree can not longer be retrieved from a recursion stack, it has to be saved in a variable. Because this path, the board states and all variables from the Monte-Carlo tree search have to be saved for all games, this solution is very memory-intensive.
The training was executed on Google Cloud Virtual Machine with 4 virtual CPU's, 24GB RAM, SSD hard drive and one Tesla K80 GPU. The training process took over one week.

9 Results

Tests:
Graph:
NNetMCTS vs NNetMCTS previous version
NNetMCTS vs Initialize Actor
NNetMCTS vs Greedy Actor

Graph
NNet vs NNet previous version
NNet vs Initialize Actor
NNet vs Greedy Actor

Graph
NNet vs NNet same version step counts
NNet prediction of illegal moves percentage

Scenarios:
Block mit Zugzwang
"Timer" durch Player 3

(Evaluation of two 2NNets-1Greedy vs 1NNet-2Greedy games)
(Evaluation which starting position is best)

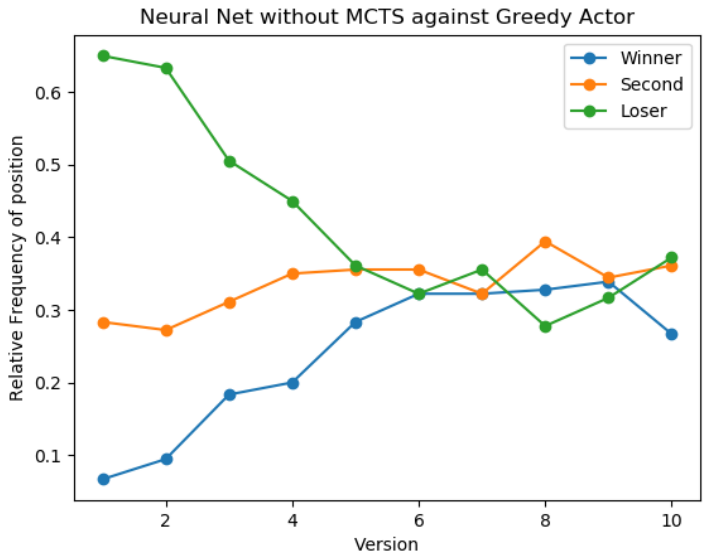

Figure 9.0.62: NNet vs. Greedy

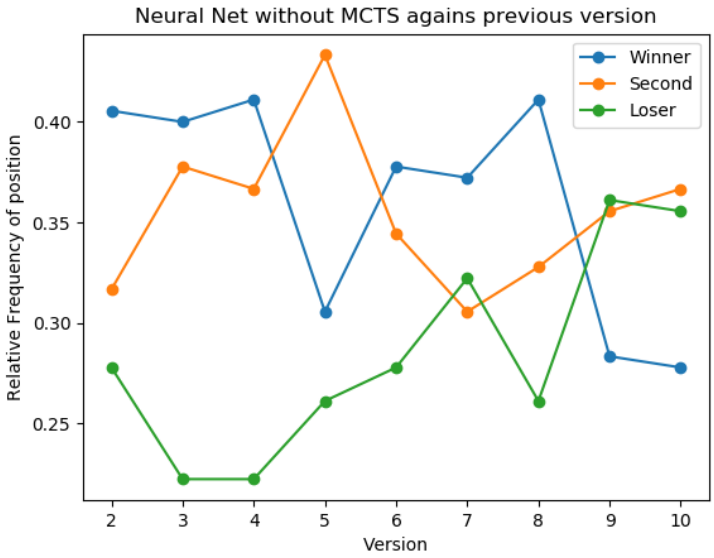

Figure 9.0.63: NNet vs. Previous

10 Conclusions and outlook

The realization of the project was a success. The methodology of AlphaZero was adapted to Chinese Checkers. The results show that the actor 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 the later iterations. Given more time and resources, the training could be continued with the goal of creating a neural network so strong that it reliably defeats human players without the use of a tree search.

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 actor 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 actor 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.

References

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

[2] [SI] 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.