# Comparing the viability of AlphaZero in games with perfect, imperfect, and no information

Master Thesis

presented by
Peter Truckenbrod
Matriculation Number 1591266

submitted to the
Data and Web Science Group
Prof. Dr. Heiko Paulheim
University of Mannheim

May 2020

# Abstract

AlphaZero has beaten the previous best game AIs in perfect information games. The question is if this success can be reproduced for games with imperfect and no information as well. Three AlphaZero implementations were created that can learn and play the games TicTacToe, Take 6!, and Rock Paper Scissors. AlphaZero then had to compete against a strong comparison AI during different stages of the training. TicTacToe was used to see if the implementation works as intended. AlphaZero can beat the comparison AI in Tic Tac Toe by a margin. Take 6! as representative of games with imperfect information could successfully be learned and played by AlphaZero. That Take 6! is played by more than two players did not influence the performance of AlphaZero. The algorithm ran however more unstable due to the random element of the game. AlphaZero could not successfully learn Rock Paper Scissors as a representative of games with no information. Even long training sessions resulted in completely random play by AlphaZero.

# Contents

# List of Algorithms

# List of Figures

# List of Tables

# Chapter 1

# Introduction

The victory of the program deep blue against the world champion Garry Kasparov in a chess tournament match in 1997 is seen as a milestone in the development of artificial intelligence. Before this event, chess was regarded as one of the few board games where the human mind could outplay a computer program. Even though chess is a game with perfect information [1], the sheer amount of possible moves each turn made it difficult for a program to choose the next move in an acceptable amount of time. [1] A human player can rely on his intuition to select his next move. A computer however, does not have such a thing. This advantage kept human players ahead of computer programs in tournament chess, where there is only a limited amount of time to choose the next move. In 1997 with an IBM built supercomputer possessing 30 nodes, each with a 120 MHz processor and 1 GB of RAM, this hurdle was finally taken. The supercomputer, which was called Deep Blue, used the data of thousands of recorded professional chess games to evaluate the advantage of a possible move. Due to the high processing speed of the hardware, it was possible to evaluate 200 million positions per second and thus to discover strong moves that finally led Deep Blue to victory against the chess world champion Garry Kasparov. [2]

The method that deep blue used is called bute force. [2] Brute force algorithms scale however very badly if the problem that they have to solve gets more complex. The hardware of deep blue had to be so fast because it needs to evaluate each turn millions of positions. The game data that was used for the evaluation had to be recorded and digitalized. Even after the milestone chess was taken by Deep Blue, there was still a popular board game with perfect information that could not be successfully played by a computer. This game is called Go. Go is like chess a board game with perfect information. However, instead of having a 8 x 8 board,

it has 19 x 19 fields. Therefore, each turn the amount of possible moves on a standard-sized Go board is way higher than the possible moves on a chessboard. The larger amount of possible moves makes the game more complex than chess by a margin. The amount of possible game states is so high that even with modern hardware, bute force algorithms can not play this game successfully against professional players.

With research advancing in the field of artificial neural networks, a new approach of creating an AI with superhuman strength for the game of Go appeared. The main idea was to transform the current state of the game in a vector, which then can be used as input for an artificial neural network. [3] The output of the artificial neural network is a probability distribution of the next possible moves with high probabilities being assigned to those moves that will lead more likely to a victory. This approach has a significant advantage: The time it takes to calculate the next move is significantly reduced. The AI gains an 'intuition'. [4]

Before a artificial neural network can produce any meaningful output, it needs to be trained. Training requires input data and the corresponding target labels. So, to train a artificial neural network to predict a strong move in a Go game, one needs to collect game states and moves that led to victory from that given game state. Here lies the main problem: Many matches need to be recorded and digitalized beforehand to train a neural network that can play a complex game like Go. Even worse, only data generated in games with professional players should be used to get strong results. Data collection can easily become a major bottleneck in the development of stronger AIs. However, even with this limitation, an algorithm called AlphaGo, using a neural network trained from recorded games with professional players, could finally beat the strongest human Go player Lee Sedol in 2016. [3]

After the victory against Sedol, the work on AlphaGo continued and led 2018 to the creation of AlphaZero. AlphaZero is an algorithm that also uses a artificial neural network tree to predict its next move in a game. The main difference is that the neural network of AlphaZero is solely trained by self-play. No more recorded games are required for this algorithm to run. Instead, AlphaZero records its game history by playing a game against itself, recording all the game states, and the final result of the game. Afterward, the artificial neural network is trained using the created data, and the self-play is repeated. By this, given enough training time, AlphaZero was able to become even stronger then AlphaGo. [5] AlphaZero produces strong moves with little processing time, and it does not even require any data besides the game rules as input. It is no surprise that the public has paid much attention to AlphaZero since its release. [6, 7] There are many speculations if Alp-

haZero can learn to play any game with superhuman strength if it just has enough training time. This thesis will examine the limits of the use of the algorithm.

## 1.1 Problem Statement

Besides the game of Go, AlphaZero also achieved very strong results in chess and shogi, being only supplied the rules of those games. [5] With the development of AlphaZero, a new level of game AI was reached. It seems like AlphaZero can learn to play a game with superhuman strength just by knowing its rules. This would mean that developing an AI for a game from now on is very simple: Just provide the rules and let AlphaZero create a strong AI. Here the question arises: Is this true for any board game? Go, chess and shogi are all rather complex 2-player games with perfect information and are all played similarly. Can AlphaZero also achieve superhuman strength if the game has a very low amount of turns or has imperfect information? Is the algorithm also applicable to games that have more than two players? Answering these questions will be the subject of this thesis.

## 1.2 Contribution

This thesis will focus on the question if the algorithm AlphaZero can be used to learn any game with perfect or imperfect information. For this, three games are selected: Tic-Tac-Toe, Take 6! and Rock Paper Scissors. These games have different grades of information. While Tic-Tac-Toe is like chess or Go a game with perfect information, Take 6! is a game with imperfect information. Rock paper scissors is an extreme case as it provides no information about the game state to the player at all. In all these three games, this thesis will focus on the following research questions:

- How to transform the game state such that it can be used to train AlphaZero?
- Can AlphaZero without any adaptation learn to play this game in a meaningful way?
- How do the characteristics of the game influence the performance of AlphaZero?
- Which adaptations can be made to increase the performance of AlphaZero for the given game?
- Which strategy is used by AlphaZero to play this game?

As the code of AlphaZero is not open source, for this thesis, the algorithm will be implemented according to David et al. [5]. Furthermore, the game routines of the three mentioned games will be implemented and, an evaluation method for the strength of AlphaZero in the given game will be developed.

## 1.3 Related Work

AlphaZero is a fairly new algorithm. Never the less, there are already many publications that examine the viability of AlphaZero in different board games. Most notably, the original paper from Silver et al. [5, 8], which presented AlphaZero and showed that it could play the games Go, chess, and shogi with superhuman strength. There have been reimplementations of AlphaZero, most notably Tian et al. [9]. The question if AlphaZero can also perform well in games with imperfect information, is examined by Charlesworth for the game Big 2 [10] and Jiang et al. for the game of Doudizhu [11]. Both Charlesworth and Jiang et al. have proven that AlphaZero can also play games with imperfect information with superhuman strength. Jiang et al. was published while this thesis was already in work. Petosa et al. [12], as well as Charlesworth [10], proved the viability for games with more than two players. There is no publication available that covers an AlphaZero implementation for the game of Take 6!. Xu et al. [13] tried to prove the viability of AlphaZero for Rock Paper Scissors; the paper was however withdrawn before review.

# Chapter 2

# Theoretical Framework

## 2.1 Artificial Neural Networks

While computers can outperform humans effortlessly in mathematical tasks, such as multiplication or addition, they are outclassed in things like image recognition. The reason for this is that in order to recognize the content of an image, the computer needs to have an abstract understanding of the content. Before artificial neural networks gained popularity, such tasks were solved by highly specialized algorithms that often only performed well in a controlled environment. However, in recent years, the development of artificial neural networks made significant progress. Now computers can perform tasks that before were considered 'to abstract'. [14][15] This section gives a rough overview of anything artificial neural network-related, which is used by AlphaZero.

An artificial neural network is a machine learning tool that, conceptually, resembles how the human brain works. It consists of different layers: at least one input layer, at least one output layer, and 0 to n hidden layers. The layers are ordered in a sequence with the easiest variant being a simple linear neural network where one layer follows another. However, the architecture of an artificial neural network can be arbitrarily complex. Each layer consists of 1 to n neurons that have connections to the neurons of the previous and following layers. [14] There can be either connections to all neurons of the previous and following layers (dense neural network) or just to some (sparse neural network). Each connection has a weight, indicating how strong the connection between two neurons is. An example of a linear dense neural network with one input layer with three neurons, two hidden layers with four neurons and an output layer with one neuron is shown in figure 2.1. For any given input, a neural network will produce a corresponding output. However, first,

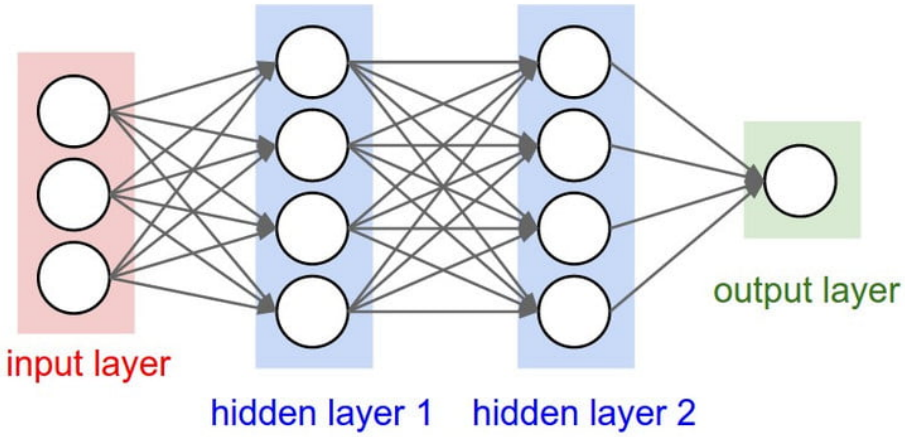


Figure 2.1: The concept of an artificial neural network. Source: [16]

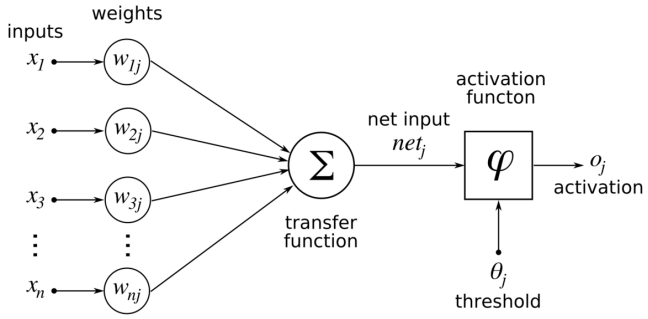


Figure 2.2: Concept of a neuron in an artificial neural network that sums all incoming values and applies an activation function. Source: [17]

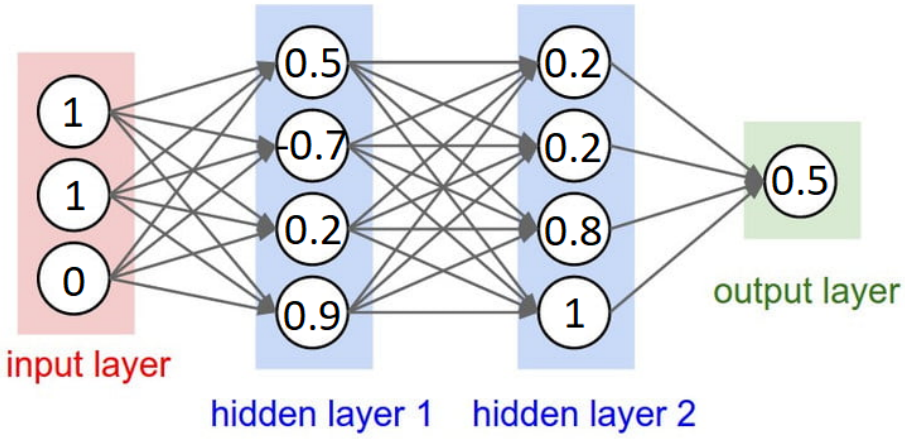


Figure 2.3: The concept of an artificial neural network with one hot encoded input values and tanh activation function. Source: [16] (modified)

the input data needs to be converted so that it has the same dimension as the input layer. Now, each neuron of the input layer gets assigned a value from the input data. As shown in figure, 2.2 those values are then 'send' to the next neuron via the connections, multiplied by the weights. In the neurons of the next layer, all the incoming values are summed up, and an activation function is applied to the sum. This procedure is done for all layers until the output layer is reached. [14] This process is displayed in figure 2.3. The values get passed through the neural network until they reach the output player. In other words: the input values get mapped to the output values. This can be described as mathematical equation $f(x) = y$ where the function $f$ is the neural network, $x$ the input values, and $y$ the output values. Given that there are enough neurons, the function $f$ can take any shape. This allows the neural network to model even the most complex relationship between input and output data. [18]

### 2.1.1 Activation Functions

As shown in figure 2.2, an activation function gets applied to the net input of a neuron. There is a huge variety of activation functions and covering all of them would exceed the boundaries of this thesis. However, the activation functions TanH and Sigmoid will be briefly covered as they are used in the implementation of the

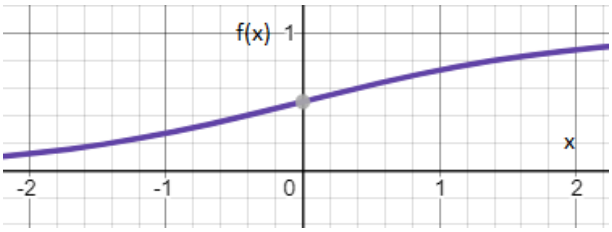


Figure 2.4: The plot shows the sigmoid activation function that can be used to output probabilities. Source: created with [19]

AlphaZero algorithm. [5] Sigmoid has the following equation:

$$f(x) = \frac{1}{1 + e^{-x}}$$

As shown in figure 2.4, the range of this function is $(0, 1)$. For high values, the output will be close to 1, and for low values, the output will be close to 0. [20] The advantage of this is that values from 0 to 1 can be easily mapped to probabilities. Since the AlphaZero algorithm outputs probabilities for the next move in a game, it makes sense that the neurons of the output layer use the sigmoid activation function. The second activation function that is used is called TanH. Its equation is:

$$f(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

The biggest difference to the sigmoid function is that its range is $(-1, 1)$ where high values create an output close to 1, and low values create an output close to -1. A net input of 0 will create an output of 0. The plot of the TanH function is depicted in 2.5 TanH has the advantage of enabling quicker learning for neural networks. [20] Furthermore it is simple to express the result of a game (win = 1, loss = -1, draw = 0) with the output of a TanH function. The AlphaZero algorithm uses this fact. [5]

### 2.1.2 Loss and Backpropagation

Now that it is clear how an artificial neural network produces an output for a given input, the question rises how it is trained so that it produces the desired output for a given input. To 'train', a neural network means to adjust its weights in the connections between the neurons. While the number of neurons, layers, and the activation functions usually stay the same during the training, the weights can take any value

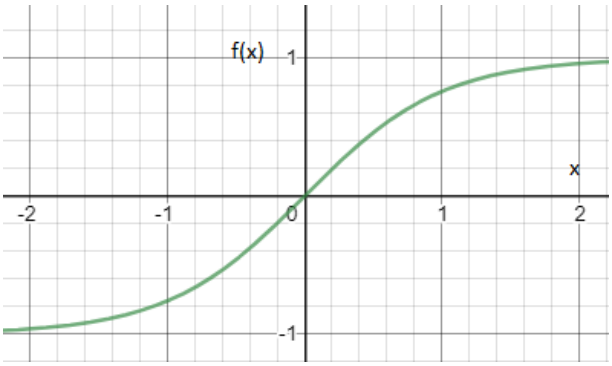


Figure 2.5: The plot shows the TanH activation function that produces an output in the range (-1, 1). Source: created with [19]

and thus determine which output will be produced for which input. The process of changing weights is called backpropagation. [21] In order to perform a back-propagation, the so-called 'loss' needs to be calculated. The loss is the difference between the actual output of the neural network and the desired output. The actual output is calculated from a given input. The desired output or target data however, needs to be supplied. After a collection of input data and target data was created, training can start. For each pair of the input- and target data, the neural network calculates the actual output. Afterward, the loss between the predicted and actual data is calculated. There are many different functions, so-called 'loss functions' that can perform this task. [21] AlphaZero uses two loss functions: mean squared error and the cross-entropy loss. [5]

Mean squared error is a straightforward loss function. Its equation is:

$$loss = \sum_{n=1}^{N} (x_n - y_n)^2$$

where $N$ is the amount of output neurons, $x$ is the output of the neural network and $y$ is the corresponding target value. This function takes each output value, subtracts the corresponding target value, squares the result, and sums up all the squared values. Huge differences between the output and the target values will create a huge loss, while no difference will lead to a loss of 0. Mean squared error is a good choice if the output of the neural network is a regression or any other non-probabilistic value.

The cross-entropy loss function can be applied to probability distributions only. The equation is:

$$loss = -\sum_{n=1}^{N} p_n * \log q_n$$

where $N$ is the amount of output neurons, $q$ is the probability distribution that is created as output of the neural network and $p$ is the target probability distribution. The parameters $q$ and $p$ both need to be probability distributions that sum to 1. This loss function is useful if the neural network should be trained such that the similarity between predicted probability distribution and target probability distribution is maximized. The loss will be bigger, the more the distributions differ and 0 if $q = p$.

The goal of the backpropagation is to minimize the loss and thus to make the output of the neural network more similar to the target values for a given input. [14] The hope is that after the training of the neural network, it produces outputs that are close to the target values, and it can abstract its gained 'knowledge' even for inputs that were not in the training data. [18] The adjustment of the weights is made according to their contribution to the loss. For this, the connections of the neurons are followed back from the output to the input layer. The weights are adjusted using the gradient decent algorithm. The adjustment rate is also called 'learning rate'. The learning rate is an important parameter in a neural network. Choosing a learning rate that is too high might result in getting stuck in a local optimum, while a too low learning rate will result in a time-consuming learning process. [18] To solve the problem of choosing the right learning rate, dynamic learning rates were introduced. They usually start with a high rate in the beginning to achieve a high convergence and lower the rate during the training to achieve loss overfitting. The dynamic learning rate used in the practical part of this thesis is called 'Adam'. Adam is a sophisticated adaptive learning rate optimization algorithm or short optimizer that, as of now, achieves the best performance of all optimizers in most of the use cases. [22]

## 2.2 Reinforcement learning

Usually, neural networks are trained with data that was collected and labeled before the actual training process. For example, to create a neural network that predicts the weather for the next day, one could collect data such as humidity, temperature, day of the year, and how the weather actually was the next day. Now, this input- and target data could be used to train a neural network, and afterward, predictions for the future could be made. This process is called supervised learning. [18] While

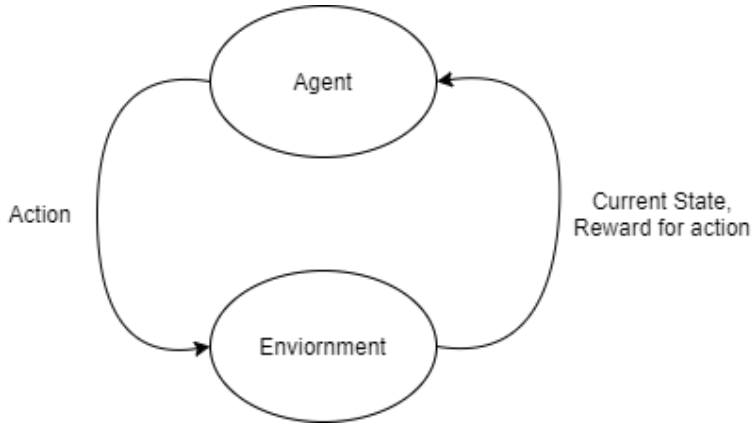


Figure 2.6: The figure shows the concept of reinforcement learning. An agent invokes some action in an environment and receives the current state of the environment and the reward for the action, which then is used to train the agent. Source: self created from [23].

supervised learning can be theoretically applied to all problems; it is not always useful to do so. To predict, for example, which move in a game of chess will most likely lead to victory, it is possible to collect data from professional games and label whether or not a certain move led to victory. In order to produce good results, however, only professional games should be recorded, and much data needs to be collected. For complex games like chess and Go, it might simply not be possible to get enough data and label it in order to generate strong predictions. With reinforcement learning, however, it is possible to overcome this problem as the input data and target values get generated on the fly. [23]. Reinforcement learning one of the three machine learning paradigms: supervised learning, unsupervised learning, and reinforcement learning. Prerequisite for reinforcement learning is the existence of an environment that an agent can interact with. The environment can change its state according to actions taken by the agent. While interacting with the environment, the agent produces the data it trains itself with and thus making it needless to supply training data from extern. [23].

In figure 2.6 the concept of reinforcement learning is shown. An agent receives the current state of an environment as input data. From this data, an action of the actor is calculated, which then gets applied to the environment. The environment then outputs the updated state of itself and eventually, a reward for the applied action. The reward can now be used to examine whether or not the agent needs

to change its action for the previous state. The new state is used to create a new action, and the cycle continues. [23] In the case of the AlphaZero algorithm, the agent is a neural network. The output it produces is a probability distribution over the next possible moves in a game. One of those moves is chosen according to the generated probabilities. This is the action in the process. The environment is the game that AlphaZero tries to learn. [5] [9] It can be seen as a game board that has a certain state e.g., white king at d3, black queen at c8 and so on. In this state, the action now gets applied e.g., white king to d4. This action will create a new state of the game and eventually, a reward. A positive reward could be given if, after this move, the game concluded in a victory, a negative reward if it led to a loss. With the old state of the game as an input and a target value that is created with the help of the reward, it is possible to train the neural network. With this process, the neural network will start the produce 'better' actions over time that will lead to a higher reward.

## 2.3 Monte Carlo Search Tree

It is beneficial to look some turns ahead to pick a good move in a game. Theoretically, the most reliable algorithm for this would be to check all possible moves for all possible turns each turn and select the move that led to the subtree with the best possible move. The Minmax algorithm, for example, plays a perfect game if its depth is so large that it covers all possible moves of the game. [24] The problem is that in most games, the amount of possible moves increases exponentially for each turn more the algorithm plans ahead. [1] There are only a few games where the amount of possible states is so small that a full computation of the search tree is feasible. For the game of Go, however, the amount of possible states is higher than the number of atoms in the universe. [4] This makes it impossible to compute a full search tree of the game. Here the Monte Carlo search tree can help to find a strong move without the need to calculate every possible state of the game. A Monte Carlo search tree applies the Monte Carlo method to turn-based board games. In the following, the process will be shortly summarized.

A Monte Carlo search tree is built with a process called Monte Carlo tree search. Each round, or expansion of the search tree, consists of four steps:

1. Select a leaf node.
2. Expand this leaf node.
3. Simulate the game from the newly created leaf node.

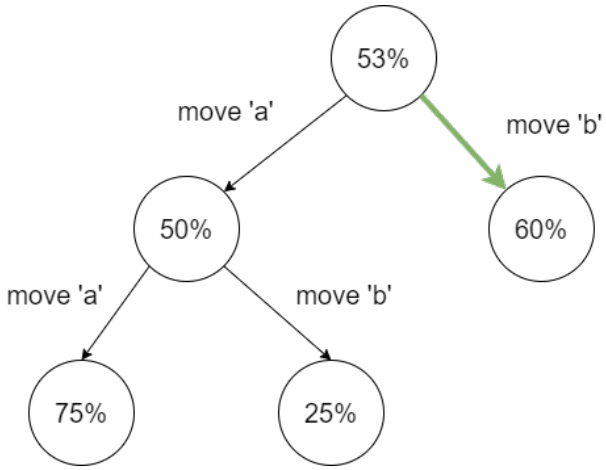


Figure 2.7: A Monte Carlo search tree for a game with two moves 'a' and 'b'. Each node contains the win probability of the player from that node. The Monte Carlo tree search selects the node with the highest win probability for expansion. Source: self created from [25]

4. Backpropagate the result of the simulation.

Figure 2.7 shows a Monte Carlo search tree during the phase of selection. The tree is used to play a hypothetical game where the player has each turn two possible moves 'a' and 'b'. Each node of the Monte Carlo search tree contains the win probability of the player from the state of that node. Note that all win probabilities depicted are from the perspective of the active player at the root node of the tree. In case it is a two-player game, the probabilities have to be inverted each consecutive turn as the player wants to maximize his win probability and minimize the one of the enemy. In the tree, the algorithm follows the nodes with the highest probability of winning. In the case of the given example, move 'b' is selected as it has 60% instead of 50% win rate. Because the node after move 'b' has no consecutive nodes, the selection step ends here. [25]

Now follows the expansion of the selected node. A new subnode is created and attached to the selected node. This process is shown in figure 2.8. The Monte Carlo tree search first needs to decide if a new subnode will be created for move 'a' or move 'b'. There are different implementations of how this is done. In the case of the AlphaZero algorithm, this decision is made by a neural network. In the example, the algorithm decides to expand the tree with the move 'a'. The new subnode

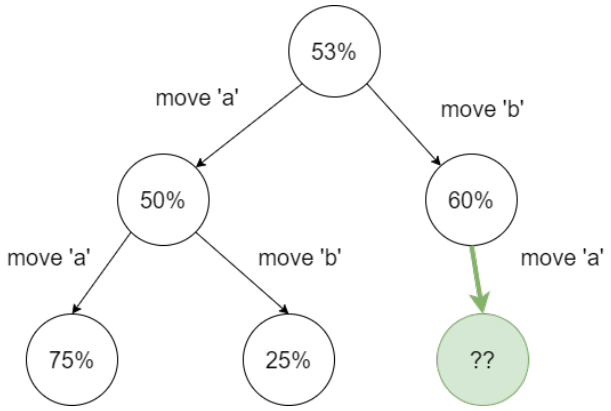


Figure 2.8: A Monte Carlo search tree during the expansion step. The selection of the move that will be performed during the expand, here move 'a', depends on the concrete implementation of the Monte Carlo tree search. Source: self created from [25]

does not have a win probability yet. However, in the next step, this is changed. [25]

After expansion follows the simulation, the goal of the simulation is to evaluate 'how good' the node is. After the simulation, the node got a win probability assigned. Figure 2.9 depicts this. Usually, a simulation is done by random playout. This means that from the given state, random moves are picked until a final state of the game is reached. If the simulation ends in a victory, a win probability of 1 is assigned to the node; in case of a loss, the Monte Carlo tree search assumes that the win probability is 0. There are however other possible simulation algorithms. AlphaZero, for example, uses a neural network to predict the win probability from a subnode. As in the example, the probability is then directly assigned to the node without any playout needed. [5].

The final step of a Monte Carlo tree search round is the backpropagation of the win probabilities. In case a node is not a leaf node, its win probability is always the average of the win probabilities of its leaf nodes. The probabilities are updated back to the root node. Afterward, one iteration is finished, and the next can start. [25] The process of backpropagation is depicted in figure 2.10.

How often the Monte Carlo search tree is expanded depends on several factors. It makes sense to have a larger search tree if the game is complex. The AlphaZero

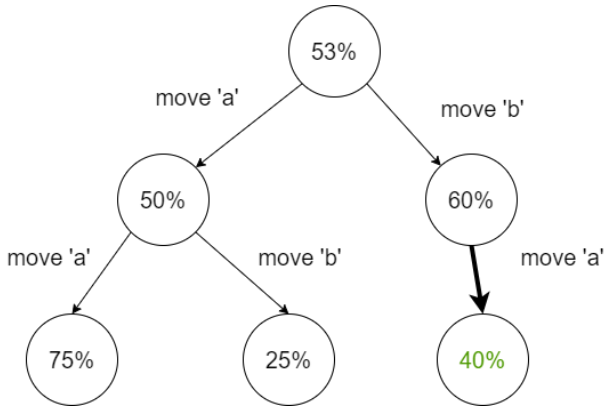


Figure 2.9: The Monte Carlo search tree after the simulation. The new node got a win probability assigned. The simulation is usually done by random playout until a final state of the game is reached. Source: self created from [25]

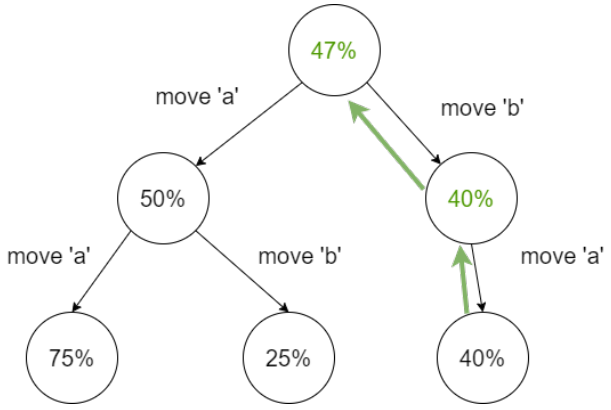


Figure 2.10: The Monte Carlo search tree during backpropagation. The win probability of each node is the average win probability of all its leaf nodes. The backpropagation is done up to the root node. Source: self created from [25]

implementation that learned the game of Go, for example, made 1600 Monte Carlo tree search iterations before deciding which move to take. More expansions demand more computing power and thus take a longer time, while also increasing the accuracy of the prediction.

After the Monte Carlo search tree is fully build-up, it is time to decide which move to play. The algorithm selects the move that led to the biggest subtree. In other words: the node is selected that has collected the most visits in all select steps. In the case of the in figure 2.10 depicted example, this would be move 'a'. The subtree under move 'a' is with 2 nodes bigger than the subtree under move 'b'. The actual win probabilities play no role in selecting the move. The selected move can now be played in the game. Afterward, and with a new game state, a new Monte Carlo tree search can be conducted to select the next move. [5]

## 2.4 AlphaZero

AlphaZero is an algorithm that can learn a wide range of games solely by self-play and can play them after enough training time with superhuman strength. It combines the concepts of artificial neural networks, reinforcement learning, and the Monte Carlo method into a single algorithm. As of now, it is the most advanced game AI for the games of Go, chess and shogi and can beat all other algorithms like stockfish. Besides the good performance are the rather easy implementation and the independence of external data the most significant advantages. As already mentioned, the algorithm learns to play a game solely by self-play. This means that no expert data, opening books, or any form of labeled data is needed to train the AI. This has been mentioned as the main motivation behind the development of AlphaZero. [5, 26, 9, 8] This section will give a quick overview of how AlphaZero works.

At the core, AlphaZero consists of a neural network $f_\theta$. The architecture concept is depicted in figure 2.11. It has an input layer that expects the state $s$ of a game as input. The state is a representation of the game at a certain turn or timestamp $t$. There are 1 to $n$ hidden layers. The amount and size of the hidden layers depends on the complexity of the game. The neural network has two output layers: one for the move probabilities $p$ and one for the win probability $v$. The move probabilities are a probability distribution over the next possible moves where the move that will lead most likely to the best result has the highest value. The win probability or simply called 'value' is an estimation, how likely it is to win the game from the given state. To handle games that do not only end in a victory or loss, but can also

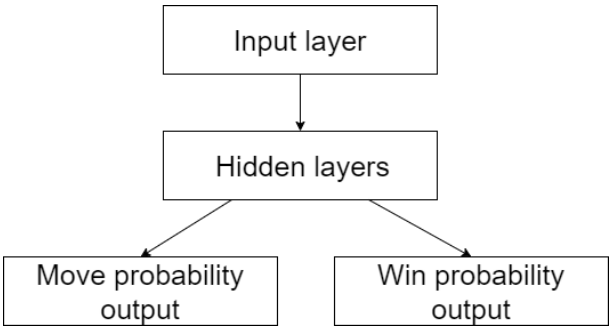


Figure 2.11: Conceptual architecture of a neural network used by AlphaZero. The input layer takes the game state. It has two output layers, one for the move probabilities and one for the win probability. Source: self-created

be a draw, the value $v$ has a range of $(-1, 1)$, where -1 stands for a loss, 1 for a victory and 0 for a draw. The output laver of $p$ has a Sigmoid activation function and the output layer of $v$ a TanH activation function. For further explanation take a look at chapter 2.1.1. The output of $p$ and $v$ always happens as a pair for the same state $s$. Put in a formula:

$$(p, v) = f_\theta(s)$$

Where $\theta$ are the parameters of the neural network or, in other words, the weights and biases. [5, 9]

AlphaZero does not pick its next move according to $p$. Instead, it conducts a Monte Carlo tree search for each state $s$ and uses $(p, v)$ for selection and simulation. The output of the Monte Carlo search tree are the probabilities $\pi$ that are used by AlphaZero to pick the next move. [5, 26] Each node in the Monte Carlo search tree gets a visit count $N$. Each time a node is visited during selection, this counter increases by 1. Furthermore, each node has an action value $Q$, and an upper confidence bound $U$. The calculation of the upper confidence bound goes as follows: Let there be a state $s$. From that state we can calculate a move probability $p$ by using the neural network $(p, v) = f_\theta(s)$. There is a probability $P \in p$ that symbolized the likelihood of taking a move $a$ from $s$ to $s'$. Then the upper confidence bound of the node that stands for $s'$ is:

$$U(s') = \frac{P(s, a)}{1 + N(s')})$$

[5, 27]The upper confidence bound is used as first indicator which nodes to expand. As $N$ usually gets bigger, the more the Monte Carlo search tree is expanded, $U$ gets

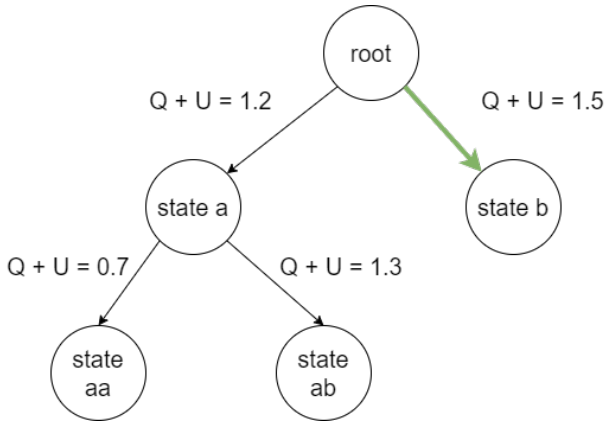


Figure 2.12: The Monte Carlo search tree of AlphaZero during the select phase. The select follows the path of the highest $Q + U$ value. Source: self-created

smaller. The calculation of the action value $Q(s)$ is done by dividing the summed up win values of all child nodes $v(s')$ by the visit counter $N(s)$. Put in a formula:

$$Q(s) = \frac{\sum v(s')}{N(s)}$$

[5, 26] $Q(s)$ will be high if the average win value in the child nodes is high. If the node has no child nodes then $v$ is the win value from $(p, v) = f_\theta(s)$. The process of selection in the Monte Carlo search tree of AlphaZero can be seen in figure 2.12. The selection follows the path with the highest $Q + U$ value until a leaf node is reached. Each node that gets passed during selection has its visit counter increased by 1. This ensures that $Q + U$ does not grow each expansion, which would lead to an imbalanced search tree. After selection follows the expansion. As shown in figure 2.13, the expansion is also done according to the $Q + U$ value. As $v$ for non existent nodes is 0, $Q$ is also 0. Because the visit counter $N$ of non existent nodes is 0, the calculation for $U$ is:

$$U(s') = \frac{P(s,a)}{1 + N(s')}) = \frac{P(s,a)}{1}) = P(s,a))$$

This means that a new node is created if $P(s, a)$ for a non-existing node during selection is higher than $Q + U$ of the existing nodes or if during selection, a leaf node is reached. [5] Then the node with the highest $p$ value is created. The simulation of the newly created node is done by calculating $(p, v) = f_\theta(s)$. With the $(p, v)$ value-pair, it is possible to calculate the $Q$ and $U$ value of the new node. A

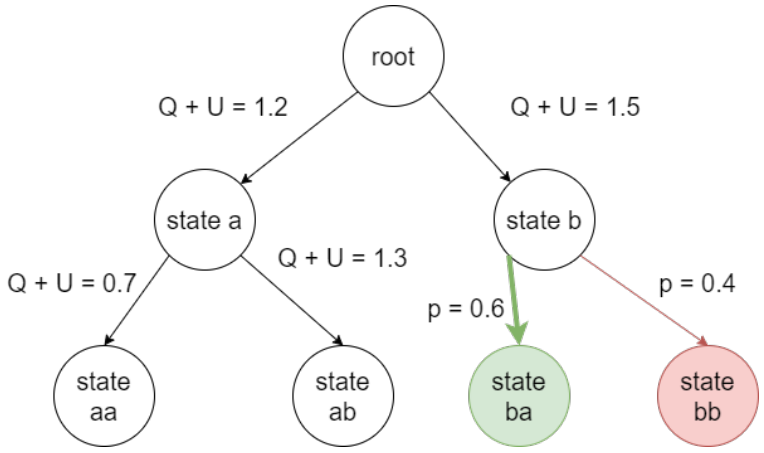


Figure 2.13: The Monte Carlo search tree of AlphaZero during the expand phase. In case a leaf node was reached during select, a new node with the highest move probability - here 'ba' is created. Source: self-created

random playout is not needed for AlphaZero. After the simulation, all $Q$ values are updated in the backpropagation step. Afterward, one Monte Carlo iteration is finished, and the cycle can continue. After a certain amount of iterations, the final move probabilities $\pi$ are extracted from the Monte Carlo search tree. With the final move probabilities, the actual move for the current turn is selected. The Monte Carlo search tree is executed for each player, each turn. [5]

AlphaZero plays the game it wants to learn against itself. This means that the same neural network simulates all players on the board. The schematic operation of AlphaZero is depicted in figure 2.14. Each turn and for each player, the state $s$ and the final move probabilities $\pi$ are saved for training. Furthermore, a variable $z$ get defined that states whether or not the game concluded in a victory. When a game is finished, a value is assigned to $z$ for each turn and each player. If the game concluded in a victory, $z = 1$, loss $z = -1$ and draw $z = 0$. Now, the training of the neural network can start. The target values for the move probabilities $p$ are $\pi$. The error is the cross-entropy between the two probability distributions. The target value for the win value $v$ is $z$. The error is the squared difference of the values. To put all together in a formula:

$$(p, v) = f_\theta(s)$$

and

$$loss = (v - z)^2 - \pi \log p$$

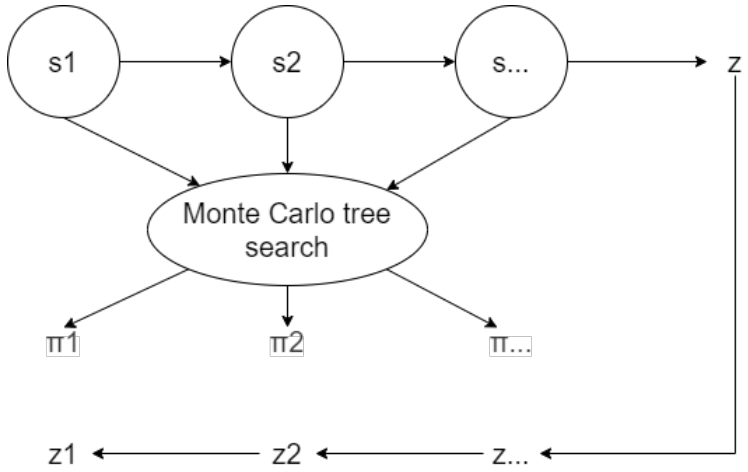


Figure 2.14: The schematic operation of AlphaZero. Each state $s$ a move probability $\pi$ is created with a Monte Carlo search tree. When the final state of the game is reached, the feedback $z$ then is backpropagated to all moves. With $s$ as input, $\pi$, and $z$ as target values, the neural network is now trained. Source: self created from [5]

[5] [26] With the loss now calculated, the parameters $\theta$ of the neural network can be updated. In the long run, this will lead to the selection of strong moves by AlphaZero. Figure 2.15 shows the Elo rating of AlphaZero in the game of tic tac toe during training. For the first iterations, the strength is AlphaZero is rising monotonously. At a certain strength level, however, the increase of strength gets slower, and sometimes the strength even decreases a bit. To avoid a decrease in strength during training, AlphaZero checks each iteration, if its new version has at least a 50% win rate against its current version. If not, all the changes made are discarded. [5] It can still happen that compared to another AI, the strength of AlphaZero decreased during training. This is however pretty rare. In the long run, more training results also in a better AI. After the training is finished, AlphaZero can be used against 'external' opponents. Further training does however not take place. The algorithm will not adapt after the training is finished.

The process of developing AlphaZero, its strengths and weaknesses, and its playstyle are comprehensively described in [4] - an interview with the creator of AlphaZero, David Silver.

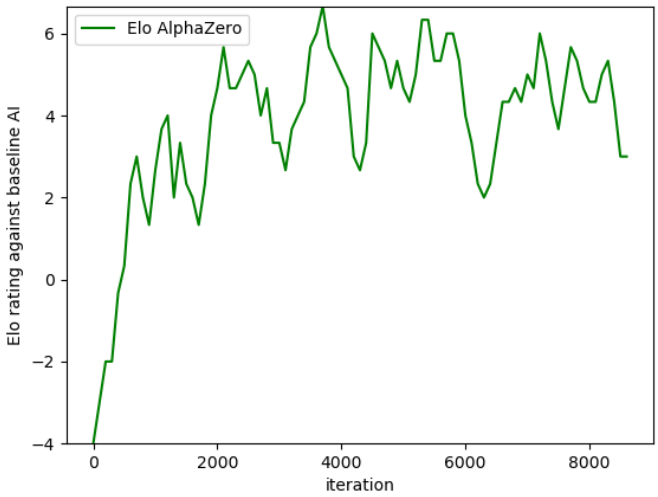


Figure 2.15: The Elo rating of AlphaZero against a baseline AI in tic tac toe during training. At the beginning of the training, the score rises more quickly then later. After some iterations, no more improvement is visible. Source: self created

# Chapter 3

# Algorithms

This chapter covers how AlphaZero was adapted for the games Tic Tac Toe, Take 6! and Rock Paper Scissors. The rules of the games are shortly summarized, and the implications for AlphaZero are pointed out. It shows how the state of the games is converted to an input vector and how the output of AlphaZero is translated into a move made in the game.

The concrete programming of AlphaZero for the games is not part of this thesis. This chapter rather explains conceptually how the AlphaZero was implemented. The implementation of all three games was done in Python. The neural network, activation functions, loss calculation, and optimizers are done with the TensorFlow library. TensorFlow offers great flexibility and for the developer and is easily customizable. Meanwhile, it is still easy to use. [28] These are the reasons why TensorFlow was chosen as framework. The implementation of the Monte Carlo search tree is done completely from scratch. This was done because adaptions needed to be made for games with imperfect information. Overall was the implementation of AlphaZero done just with the information provided by the original paper. AlphaZero is unfortunately not open source. There is however, literature about the reimplementation of AlphaZero. [9] The original paper is also detailed enough to allow a good reimplementation. The goal of this thesis is to stay as close as possible to the original. [5]

## 3.1 Tic Tak Toe

The game of Tic Tak Toe is well known around the world. The reason why it was chosen as one of the tree games covered by this thesis is its simplicity. A Tik Tak Toe board consists of only nine fields. Because a field can only be occupied one

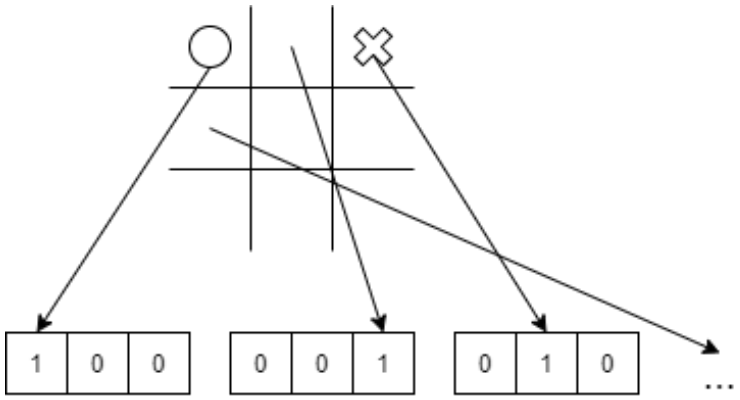


Figure 3.1: The conversion of a Tic Tak Toe game state to a one-hot encoded input vector. Each field can either be taken by player 1, or player 2, or it can be empty. This results in a $9 * 3 = 27$ bit long input vector. Source: self-created

time, this results in only $9! = 362880$ possible states of the game. Considering that the game can be mirrored each axis, the number of possible states can be even reduced further. The most important advantages of experimenting on a simple game are:

- Easy implementation of the game rules and AlphaZero.
- It is possible to comprehend the results of AlphaZero.
- A human can come up with the strongest possible strategy in the game. Thus it is possible to create a strong AI to compare AlphaZero to.
- The training times are shorter, which allows us to execute more experiments quicker.

Tic Tak Toe is an excellent way to find out if the AlphaZero implementation used by this thesis works fine. Meanwhile, it is already clear that AlphaZero will work on the game. Tik Tak Toe is a two-player game with perfect information just like Go or chess, and as such, there is no need for adapting the algorithm. The goal of this thesis is to find out of AlphaZero also works on games with imperfect information. Tic Tak Toe will not help here, but it can be seen as a baseline for the other experiments.

A Tic Tak Toe board consists of nine fields, ordered in a 3x3 square. Each turn,

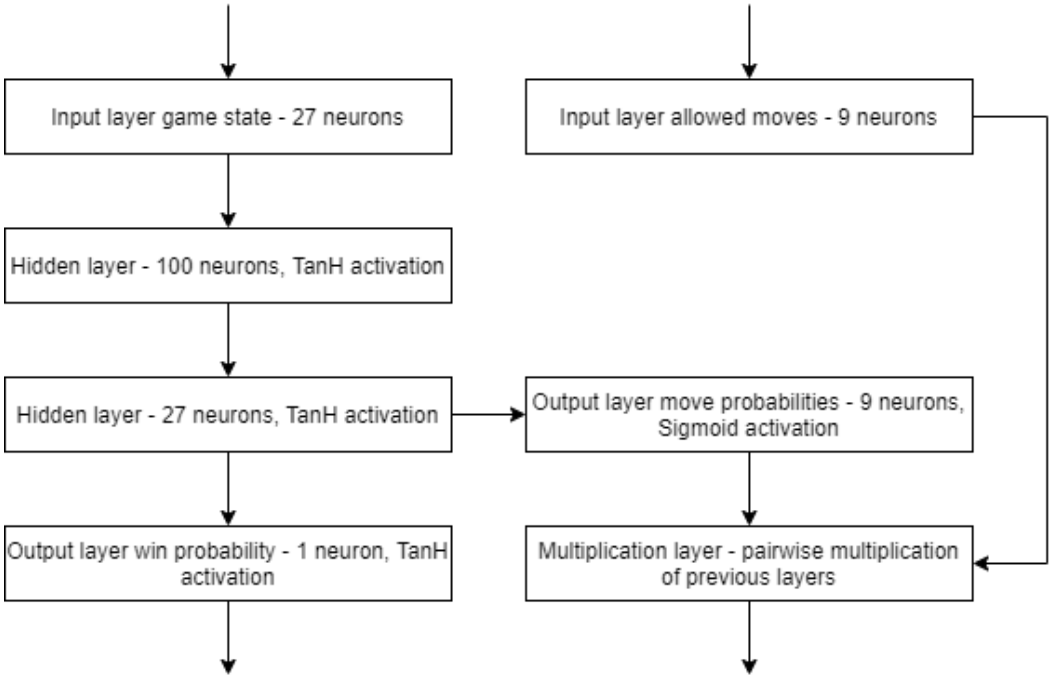


Figure 3.2: The architecture of the neural network that is used by AlphaZero for the game of Tik Tak Toe. From the game state, the neural network predicts the win probability $v$ and the move probabilities $p$. $p$ is then pairwise multiplied with the values of the second input layer to only allow moves that are not illegal. Source: self-created

one of the two players picks a field and marks it with his symbol. After the field is marked, the turn is finished. Now it is the turn of the next player and so on. The goal of the game is to collect three own symbols in a row anywhere on the field. The game ends if one player got three symbols in a row or if all fields are taken. In the first case, the game will end with a victory for the player who filled the row; in the second case, the game will conclude in a draw. It is a common strategy in the game to actively prevent the enemy from forming a row of three. If both players play 'perfect', each game will result in a draw. The player that starts the game has a much easier time to win as he is one turn ahead.

As simple as the game is converting its state into an input vector that can be con-

sumed by a neural network. Each field can either be empty, taken by player 1 or taken by player 2. One hot encoding this information takes three bit. Afterward, the three-bit information of each field is concatenated to a single input vector. The process of converting the game is depicted in figure 3.1. The final vector has a length of $3 * 9 = 27$ bit. It contains all information about the game at a given turn $t$. The inspiration for this conversion was taken from Charlesworth 2018 [10]. The original implementation processed the input (for Go) as a 19x19 image. [5] For each reimplementation, the input conversion seems to be different and mostly depending on the game. [11]

The architecture of the neural network used for Tic Tak Toe is depicted in figure 3.2. There are two input layers. One for the game state and one that takes a one-hot encoded vector of the allowed moves. After the game state-input layer come two hidden layers with TanH activation functions. The first hidden layer has 100 neurons, and the second hidden layer has 27 neurons. This amount of hidden layers, neurons, and their activation functions have shown to produce good results in practice. However, no experimental evaluation took place if these parameters are optimal. Parameter tuning might be useful for future work. From the game state $s$, the outputs win probability $v$, and move probability $p$ are calculated. The move probabilities $p$ are then pairwise multiplied with the allowed moves input. The process can be seen in figure 3.3. This ensures that only moves that are not illegal can have a higher probability than 0. If this were not the case, AlphaZero would need to learn which moves are legal besides learning how to play strong moves. [10] This 'trick' decreases the training time needed by AlphaZero. The move probabilities without the illegal moves may not sum up to 1. This is however no problem as these probabilities just get used by the Monte Carlo search tree. The final move probabilities then sum again to 1 and can be used to select the next move.

The Monte Carlo search tree of AlphaZero uses the neural network for its selection and simulation. The tree search works as described in chapter 2.4. It is important to note that the $v$ values are inverted each second level of the tree. This happens because the moves of both players are simulated. This is shown in figure 3.4. The tree search starts with simulating the game from the perspective of the current player. All simulations on the second level of the tree however, are done from the perspective of the opposing player. Because good moves of the enemy are bad for the player and vice versa, the $v$ value has to be inverted. All nodes on the third level are again simulated from the perspective of the current player and so on. The tree is expanded ten times before the final move probabilities are extracted. The number 10 is a good compromise between depth and speed. A lower number might result in a weaker play of AlphaZero, while a higher number will increase the training

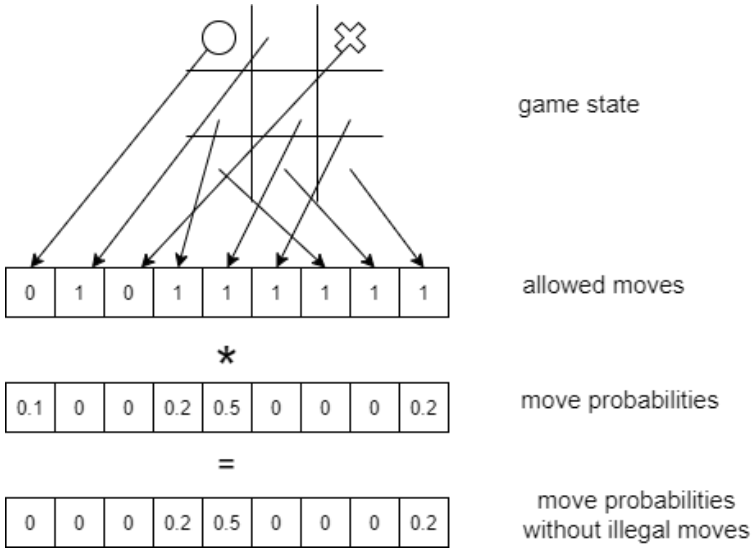


Figure 3.3: The allowed moves are generated from the game state. An input vector is created where empty fields get the value 1 and occupied fields the value 0. After the pairwise multiplication with the move probabilities, all illegal moves are canceled out. Source: self-created

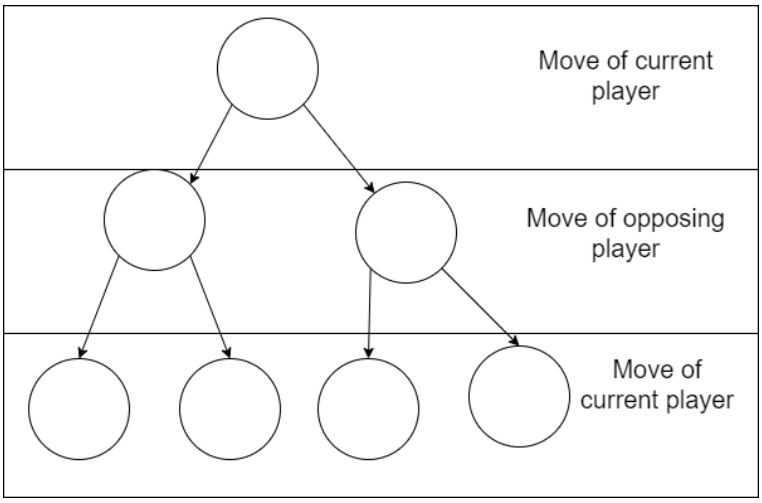


Figure 3.4: Move made from the root state of the Monte Carlo search tree is simulated from the perspective of the current player. AlphaZero assumes that the enemy plays like itself, so it simply simulates the moves of the enemy every second turn. As it is the goal to select the strongest move for the current player, the $v$ value has to be inverted for all simulations form the enemy perspective. This is because good moves of the enemy are bad for the current player and vice versa. Source: self-created

time. There were however, no experiments made if 10 is the best number for expansion of the Monte Carlo search tree for the game of Tik Tak Toe. Tuning this parameter could be helpful in increasing the performance. The Monte Carlo search tree produces a final move probability $\pi$. This probability distribution is then used to determine the actual move played by AlphaZero. Each turn of the training, the state $s$, the allowed moves $x$ the final move probability $\pi$ are saved. At the end of a training game, the game score $z$ is determined for each side of the game party. In case of a victory, $z$ is 1, in case of a draw 0 and after a loss -1. Afterward, the neural network is trained with $s$ and $x$ as input and $\pi$ and $z$ as target values. The calculation of the loss is as described in chapter 2.4

$$loss = (v - z)^2 - \pi \log p$$

No adaptions from the original AlphaZero are needed here. The optimizer used in this thesis is Adam, with a learning rate of 0.0001. Tensorflow recommends this learning rate. Tuning this parameter a bit might be beneficial for the algorithm. However, with 0.0001, good results could be produced.

One training game during training where AlphaZero plays both sides takes on average 300 ms. For 10000 games, a training time of 3000 seconds, so roughly an hour is needed. Ten thousand games are usually more than enough to train a Tik Tak Toe AI fully. The original AlphaZero implementation plays 1000 games, saves the data from these games, and then trains on the batch of data. The implementation used by this thesis however, trains the neural network after each game. First collecting a lot of data and then training the whole batch might decrease the training time needed by AlphaZero. There are however, no numbers on this topic, and the actual time saved will depend heavily on the actual programming of the algorithm.

## 3.2 Take 6!

Learning Take 6! with AlphaZero is an interesting challenge for various reasons:

- Unlike the Tik Tak Toe, Go or chess it is a game with imperfect information
- It can be played with 2, 3, 4, or 5 players. It is in contrast to the games as mentioned earlier that can only be played by two players.
- All players make their turn at the same time
- A turn in the game can require two instead of one decision

Before we explain how these problems were solved, lets fist take a look at the rules and layout of the game to realize where the problems mentioned stem from. Take 6! is a turn-based card game. The game consists of 104 cards, each containing one number from 1 to 104. Each number exists only once in the game. There can be 2 to 5 players in the game. At the start of the game, each player gets ten randomly selected cards. A player can only see his cards. The cards of the other players are hidden. This means that from the perspective of a player of Take 6! it is not possible to know the complete state of the game. Therefore it is a game with imperfect information. Furthermore, four cards are randomly selected at the start of the game. Those cards are visible for all players and each of them forms a so-called batch. Every turn, each player selects one card that he wants to play and puts it not visible for the others in front of him. When all players finished selecting their cards, the selected cards are shown to the other players. Now, starting from the card with the lowest number, the cards are attached to the batch whose last card is smaller then the card played and has the lowest difference to the played card in comparison to the other batches. If a batch exceeds the size of 5, the player who added the 6. card, has to take all cards in the batch as punishment. The 6. card forms the start of the new batch. If a player played a card which is smaller then any of the last cards in the batches, he could choose which batch he will take as punishment. His card forms the start of a new batch. The cards taken as punishment are put to the side and can not be played. Here we can see two properties of the game that make it very different to Go and chess. Fist, all the players take their turn at the same time. This makes the construction of the Monte Carlo search tree difficult. Furthermore, a player may have to make two choices during a turn:

1. Select which card to play.
2. In case this card is lower than the last card in any of the batches: select which batch to take as punishment.

The problem is that these are not simply two consecutive moves but rather two choices on different problems. A neural network can however, only solve one problem at a time.

Besides a number, every card has a certain amount of penalty points. The amount of penalty points is calculated in the following. The penalty value is at least 1 per card. Cards that can be divided by 5, 10, or are a brandy number give extra penalties. The detailed calculation of the penalty points by the number on the card is shown in the algorithm 1. The game es finished when the players played all of their ten cards. After finishing the game, the penalty points are counted. The player with the least penalty points wins the game, the one with the most loses. All other

**Algorithm 1**

CALCULATEPENALTYVALUE($number$)

```
penalty = 1
if number%5 = 0 then
    penalty = penalty + 1
end if
if number%10 = 0 then
    penalty = penalty + 1
end if
if number = ▷ brandy number then
    penalty = penalty + 4
end if
return penalty
```

players are considered neutral. The official rules state however that the winner is only determined when one player reaches more than 100 penalty points, and until this happens, new games are started. For this thesis, the rules were altered here. The reason for this is that there are on average, roughly five consecutive games needed until a player reached 100 points. This means that the target value for the win probability $z$ could only be set for all five games. Because of this, it could happen that a lost game was still considered a victory when the player collected the least amount of penalty points overall. This slowed down learning considerably. For AlphaZero, it makes much more sense to see each game on its own. The information about how many penalty points the player collected overall does not influence the playstyle in a single game, so seeing each game isolated does not even influence the behavior of AlphaZero.

Converting the state of the game from the perspective of a player in an input vector that can be consumed by a neural network is quite a challenge. While Take 6! is a game with imperfect information, there is still much information that the player certainly knows at any point in the game. A player of the game knows for sure:

- The numbers printed on the cards that he holds in his hands.
- The number of cards that he and the other players still hold.
- The penalty points printed on the cards that are on his hands.
- The cards that lay in the batches.
- The last card of each batch.

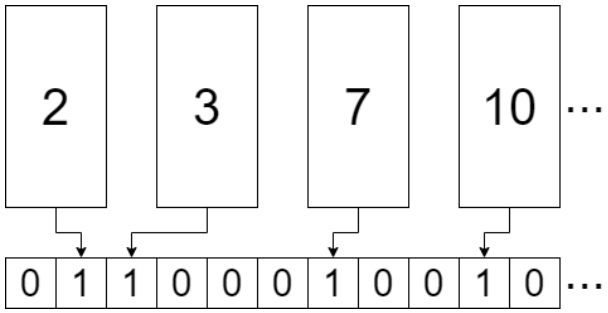


Figure 3.5: The cards on the hand of a player are encoded in a 104-bit vector. Because each number in the game exists only once, the encoding can be done by the simple rule that if a player holds a card with the number $n$, then the $n$th bit is 1. Else the bit is 0. Source: self-created

- The penalty points of each batch.
- The cards that the other players have already played.
- Which cards exist that the other players may or may not have.
- How many penalty points which player has.

This is much information that needs to expressed efficiently and unambiguously. Furthermore, the neural network must get fed with as much information about the game state as possible. AlphaZero can only make a decision based on the information that it has. Leaving, for example, out the information on how many penalty points the opponents have, will lead to a completely different assessment of the current situation and thus skew the Monte Carlo tree search. A strong player considers as much information as possible before making his move. AlphaZero has to do the same. As a model for the conversion of the state of the game to a useable input, Charlesworth 2018 was used. [10]

Encoding the cards that are on the hand of the player can be done without much effort. We know that there are 104 cards with different numbers, and every number exists only once. So encoding the cards on the hand is done by 104 fields that can either be 1 if the player owns this card or 0 if he does not. The process can be seen in figure 3.5. Encoding the player cards this way has the advantage that it always takes 104 bit regardless of how many cards the player has on his hand left. That is important because the shape of the input layer of the neural network is static. One could argue that these 104 bits already contain the information about how many

cards the player still holds on his hands. It has, however, shown to be beneficial to encode the amount of cards extra. This can easily be done as there can only be 0 to 10 cards. Thus the amount of cards is one hot encoded in with 10 bits. Because all players always have the same amount of cards in their hands, we only need this information once.

Converting the state of the batches into a consumable input for the neural network is more complicated. A player can see at any point in the game all batches with all their cards, the sequence of the cards, the numbers on the cards, and the penalty points on the cards. Most of this information however, is completely irrelevant for deciding which card to play next. The important information is:

- What is the last card of the batch?
- How many cards are already in a batch?
- How many penalty points to all cards combined in a batch have?

The last card of the batch is encoded analog to the hand cards. The only difference is that there is only one card instead of up to 10. The number of cards in the batch can vary from 1 to 5. This information can be simply one hot encoded with 5 bits. The amount of penalty points is more tricky. There can be 1 to 26 penalty points in a batch. While it is a very common occurrence that a batch has only one penalty point, it nearly never happens that the number 26 is reached. 26 penalty points would mean that all five cards in the batch are brandy numbers. Because this happens so rarely, AlphaZero has no training examples for this case. If such a situation is encountered during a game, it will most likely negatively affect the performance of AlphaZero. A way around this a smart binning of the possible penalty points. The bins chosen for this thesis are:

- $<= 1$ penalty point
- $1 >= 2$ penalty points
- $2 >= 3$ penalty points
- $3 >= 5$ penalty points
- $5 >= 7$ penalty points
- $7 >= 10$ penalty points
- $>= 10$ penalty points

Like this, it is possible to encode the penalty points of a batch with 7 bits. Furthermore, all of those bins will most likely be used during training. Similar to the penalty points of the batches, the encoding of the already taken penalty points of the player takes place. This information is for AlphaZero crucial as it greatly influences the win probability output. The win probability contributes majorly to the result of the Monte Carlo tree search and, thus, to the actual move taken. Because the number of penalty points that a player has taken can be much higher than the penalty points of a batch, different binning ranges are taken.

- $<= 1$ penalty point
- $1 >= 2$ penalty points
- $2 >= 4$ penalty points
- $4 >= 7$ penalty points
- $7 >= 10$ penalty points
- $10 >= 15$ penalty points
- $15 >= 25$ penalty points
- $25 >= 40$ penalty points
- $>= 40$ penalty points

This has shown to be the best practice during development. There were however, no experiments conducted to verify if this is the best possible binning. Tuning those values might be beneficial for future work. Last but not least, the information about which cards were already played is saved. A played card is any card that is either lying in any of the batches or any card taken as a penalty. While it is for humans not a common tactic to count cards, it is of the highest importance of AlphaZero. The reason for this is that in order to get meaningful results from the Monte Carlo search tree, AlphaZero needs to predict what the other players are going to play most probably. To predict an enemy move, it is required to know which cards the other players have. Having the information which cards the enemies certainly do not have anymore, helps to predict which cards they still hold. Played cards are encoded just like the cards that a player holds on his hand. This takes an additional 104 bit.

After collecting and encoding all this information, the final input can be formed. This is done by merely concatenating all the aforementioned bitstreams. It is crucial to keep the information always in the same order. The neural network can not

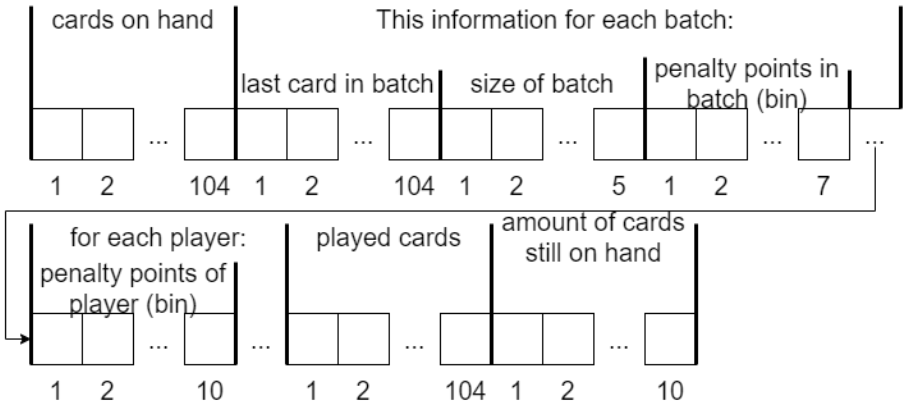


Figure 3.6: The final input given to the neural network. The encoded game state includes the cards on the hand, the last card of each batch, the number of cards in each batch, the penalty points in each batch, the penalty points of each player, the cards played, and the number of cards on the hand left. All the information is encoded with zeros and ones. Source: self-created

realize that, e.g., the information about the cards still on the hand, and the played cards are switched. Thus changing the order of the input will produce rubbish results. The full arrangement of the input data can be seen in figure 3.6.

The architecture of the neural network for Take 6! is more complicated than for Tik Tak Toe. The main challenge in learning Take 6! with AlphaZero is to simulate the move of the opponents in the Monte Carlo search tree. For any game with perfect information, this step is relatively easy. As we saw in Tik Tak Toe, we can take the game state as seen from the opponent's perspective and supply it to the neural network. In a game with imperfect information like Take 6! however, a player does not know the complete game state of the opponent. In case of Take 6! we do not know any card on the hand of the enemies. Without this information, the neural network can not be used for a good simulation and thus, the result of the whole tree search is bad. The initial idea around this problem was to assign randomly chosen cards to the opponents game state for each simulation. This, of course, led to fairly bad results. The randomness could be reduced by only assigning cards to the enemies that were not yet played or on the own hand. While this improved the strength of AlphaZero, it still was not the perfect solution. The solution that was finally implemented by this thesis work was to let the neural network not only predict the win probability and the moving probability from a game state but also

the cards that are most likely played by the opponents next turn. In other words: instead of just having a move probability for the own moves, a move probability for the enemy is computed as well. AlphaZero still does not know which cards the other players have on their hands, but the prediction of which card they could play gets much better. An overview of the architecture of the neural network can be seen in figure 3.7. There is only one enemy move probability output, while there can be multiple opponents. In case there are $n$ opponents, then the $n$ cards with the highest probability to be played by the opponent are selected. It is also possible to have one output per opponent. This would however, result in a much more extensive neural network, which then takes more time to be adequately trained—Thats why this thesis went with the first approach.

Figure 3.7 shows that there are 416 neurons in the output layer of the player move probability and the opponents move probability. There are however, only 104 possible cards that could be played. The reason why it is exactly four times that amount is that there are four possible batches that a player might have to take as a punishment. According to the rules, this can only happen to if a card is played that is lower then the last cards of any batch and no other player plays an even lower card. Here lays however a big problem for AlphaZero. For all the games that AlphaZero mastered so far, each move needs only one decision. In Go, the task is to select one field in the game to put down a stone. In Take 6!, it might be necessary to decide which card is played, as well as which batch take is taken as a penalty. These are however, two separate problems with separate action spaces. AlphaZero has only one neural network that given an input, output values for the same problem. It is not possible to first expect on move probability output and afterward an output that states which batch should be taken as a penalty. One could think about introducing a second neural network for this problem and somehow merge it all in one algorithm. This, however, differs a lot from the original AlphaZero implementation. The solution selected in this thesis is much simpler. Instead of selecting which batch one does take as penalty after everyone revealed his card, each player will have to select a batch that he will take as a penalty if needed, before they reveal their card. This means that besides choosing a card to play, a player at the same time has to choose a batch to take even if there is no chance that he gets in the situation where this information is needed. This allows us to keep the overall structure of AlphaZero as it is. One could argue that because of this approach, AlphaZero loses the information about which cards the other players selected that turn, before selecting which batch to take as a penalty. While this is true, the question rises if this information would influence the decision of AlphaZero. It might be that AlphaZero would play Take 6! a bit better, but if the influence is significant is debatable. There were however, no experiments conducted on this matter in this

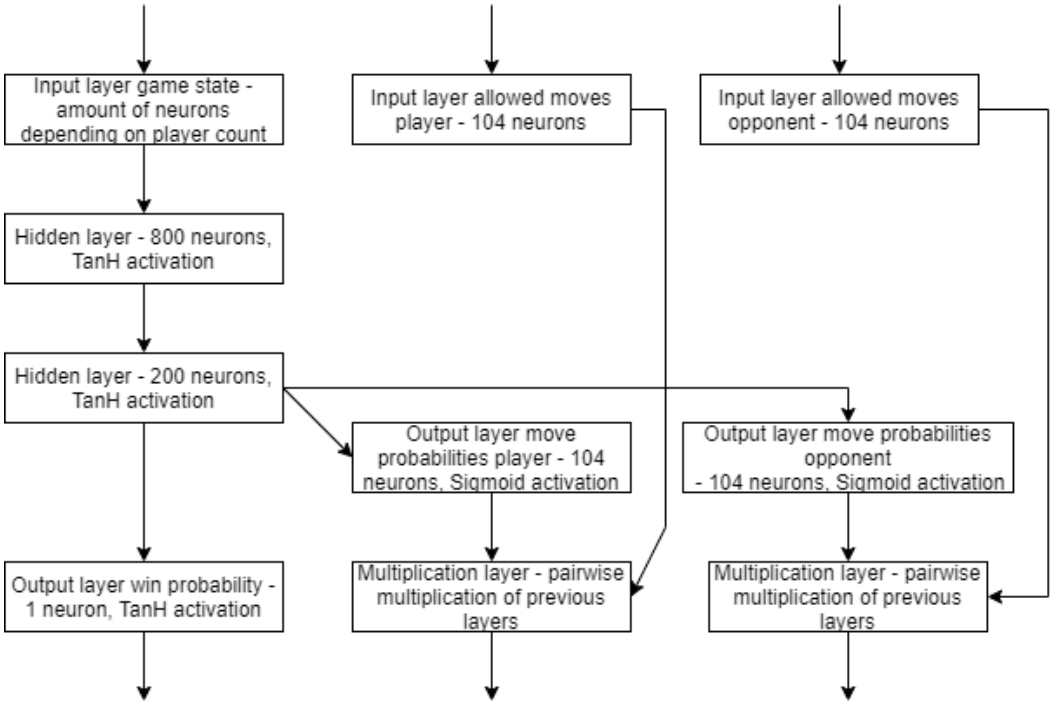


Figure 3.7: The architecture of the neural network that is used by AlphaZero for the game of Take 6!. From the game state, the neural network predicts the win probability $v$ and the move probabilities $p$. $p$ is then pairwise multiplied with the values of the second input layer to only allow moves that are not illegal. This is precisely how the original neural network of AlphaZero works. The main difference is that now also the opponents move probabilities are predicted. Source: self-created

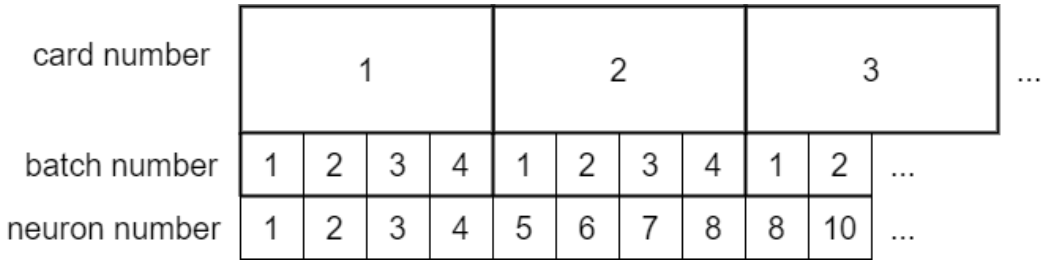


Figure 3.8: The interpretation of the move probabilities output for Take 6!. There are 496 output neurons, labeled with a number from 1 to 496. Conducting a euclidian division by four on the neuron number, taking the remainder, and adding 1 tells for which batch the probability from the neuron stands. The quotient meanwhile stands for the card that is represented by this neuron. By choosing one neuron, a card and a batch get selected at the same time. Source: self-created

thesis. For selecting a card and a batch at the same time, there are two possible technical implementations:

1. One big output layer that covers all possible combinations of cards and batches.

2. Multiple output layers for cards and batches.

This thesis went with the first approach. While this solution requires much more output neurons, it is much cleaner. Having one output layer for the cards and one for the batch is difficult for simulating the opponent. As there can be multiple opponents and we select the cards that they play in a descending order according to their probability, the question would arise to which selected card which batch belongs. All to the batch with the highest probability or some other distribution? To avoid this problem, the first solution is selected. There are 416 possible combinations of 4 batches and 104 cards. This is why the output layer of the move probabilities has 416 neurons. The interpretation of the neuron can be seen in figure 3.8. The neurons are divided into groups of 4. The first neuron in a group stands for the first batch and so on. Furthermore, the first group stands for the card with the number 1 and so on. Converting a neuron number to the card and batch it stands for can be done by conducting a euclidian division by four on the neuron number. The remainder plus one is the batch number and the quotient stands for the number on the card. The output of each neuron is a probability. The neuron with the highest probability points to the card that AlphaZero will most likely play and to the batch

that it will take in case it has to.

One could argue that having all 104 possible cards as output does not make sense as a player can anyway just hold up to 10 cards on his hand. Instead, there could be ten possible cards as output. They symbolize the cards that are on the hand of the player. Furthermore, those cards are always ordered from the smallest card to the highest card. Given four batches, this would require only 40 output neurons. This approach was tried but led to bad results. This happened most probably because the meaning of the output neurons changes each turn and each game. This made it impossible for AlphaZero actually to converge to a strong game play. Thus this idea was abandoned.

The interaction between the Monte Carlo search tree and the neural network is quite different from the example given in the original AlphaZero paper [5] and the Tik Tak Toe example. From a state $s$ the neural network predicts the win value $v$, the move probabilities $p$ and the enemy move probabilities $e$. As stated in the original paper, $v$ is backpropagated up to the root state and used to calculate the action value. $p$ is used to calculate the upper confidence value for the child nodes. Differing from most approaches is the prediction of $e$. In the algorithm used in this thesis, $e$ is used to estimate which cards the opponents will play. This information is then used to influence the state $s$ of the child nodes, which then influences $v, p$ and $e$ itself. Because the enemy moves are predicted with $e$, all nodes in the Monte Carlo search tree are seen from the perspective of the player that wants to calculate the final move probability. So no inversion of $v$ needs to take place. While this differs from the original AlphaZero algorithm, there is no possible way that the unchanged algorithm would work here. $v$ and $p$ can not be predicted for the opponent because we are not entirely sure how $s$ looks for the enemy. The prediction of $e$ can be seen as a substitute for this.

The tree is expanded 16 times before the final move probabilities are extracted. It is again not experimentally proven that 16 is the best possible number of expansions; the results of the experiments presented in the next chapter however, show that AlphaZero can produce strong results with this parameter. As in the original paper described, the Monte Carlo search tree produces a final move probability $\pi$. This probability distribution then is used to determine the actual move played by AlphaZero. Each turn of the training the state $s$, the allowed moves $x$, the opponent allowed moves $y$, the final move probability $\pi$, and the moves played by the opponents $o$ are saved. $o$ is encoded the same way as $p$ or $\pi$ with 496 neurons that represent the probability of a certain card-batch combination being played. At the end of a training game the game score $z$ is determined for each side of the game

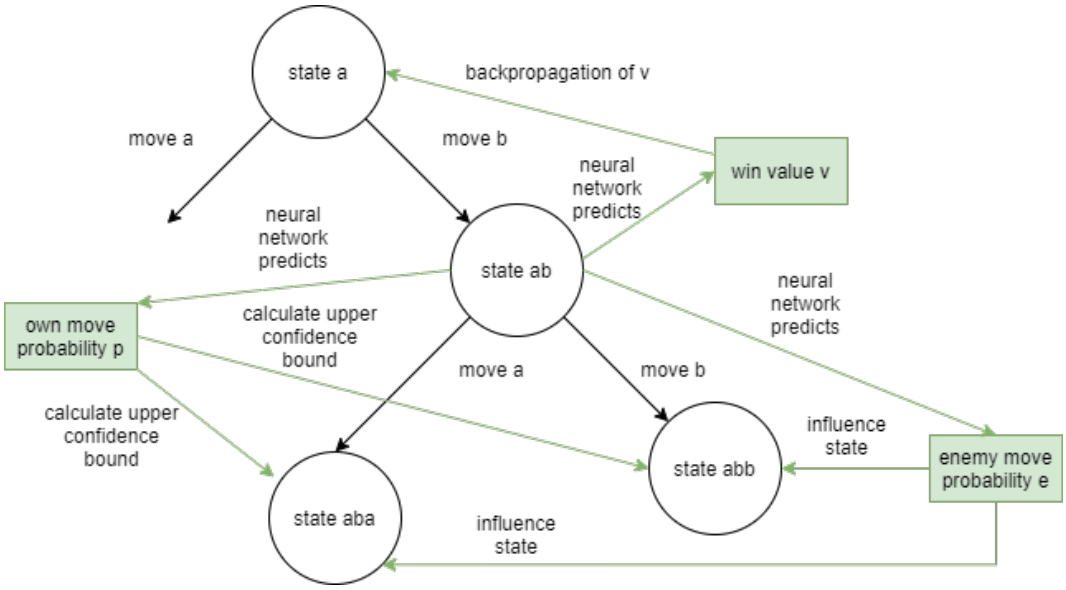


Figure 3.9: A schematic Monte Carlo search tree and the influence of the neural network (in green) for the game of Take 6!. From a state $s$ the neural network predicts the win value $v$, the move probabilities $p$ and the enemy move probabilities $e$. $v$ is backpropagated up to the root state. $p$ is used to calculate the upper confidence value and $e$ is used to predict the moves of the opponents and thus to influence and update the state $s$ for the child nodes. The prediction of $e$ and its influence of $s$ differs from the original paper and is used to play a game with imperfect information. Source: self-created

party. As for Tik Tak Toe, in case of a victory, $z$ is 1, in case of a draw 0 and after a loss -1. Afterward, the neural network is trained with $s$, $x$, and $y$ as input and $\pi$, $z$, and $0$ as target values. The calculation of the loss now differs a bit to the one described in chapter 2.4:

$$loss = (v - z)^2 - \pi \log p - o \log e$$

for

$$(p, e, v) = f_\theta(s)$$

Added to the original error is now the cross-entropy error between $e$ and $o$. The goal is that the predicted enemy moves overtime, match the observed enemy moves. The more both differ, the higher the error. The rest of the calculation is as described in chapter 2.4. While some papers went with a similar approach [11] to solve the problem of the Monte Carlo tree search in imperfect informations games, others left the Monte Carlo tree search completely out [10]. The optimizer used in this thesis is Adam, with a learning rate of 0.0001. This learning rate is recommended by Tensorflow. Tuning this parameter a bit might be beneficial for the algorithm. However, with 0.0001, good results could be produced.

The average training game with three players and the parameters as mentioned above, takes 6 seconds. While this seems to be very long, it should be considered that each turn for each player one Monte Carlo tree search needs to be executed, which needs 16 predictions from a neural network which can not be executed in parallel. This means that each game $3 * 10 * 16 = 480$ predictions from the neural network are needed. This explains the 6 seconds of playtime. For 10,000 training games, this means a training time of roughly 18 hours. So unsurprisingly, the time needed to train AlphaZero for Take 6! is considerably higher than for Tik Tak Toe. The neural network is trained after each training game. As for the other games, no collection of a batch of data and then training on this data all at once takes place.

## 3.3 Rock Paper Scissors

Of the three games examined by this thesis, Rock Paper Scissors is arguably the simplest. This does however not mean that AlphaZero has an easy time to learn this game. The main problems of playing Rock Paper Scissors are:

- It is hard to describe a state $s$ for the game. There is no information about the turn itself. The only information available is the game history.
- The turns or 'games' do not influence each other. Winning one turn does not mean that the winner has an advantage in the next turn.

- It is challenging to predict what the opponent will play. Both players have besides the game history no information to base their move on.

The rules of the game are quickly explained. The game is played with two players. There are three elements: Rock, Paper, and Scissors. Each turn, each player picks one element in secret. When both players finished picking their element, the element of each player is revealed. Rock beats Scissors; Scissors beats Paper; Paper beats Rock. The player who beats the other player gets a point. If both players picked the same element, no one gets a point. For this thesis, the rule is added that 100 consecutive turns need to be played. The player who collected more points in these 100 games, wins. While the actual 'rule' is that an arbitrarily amount of games can be played and the winner can be determined at any point. This was however changed so that AlphaZero can 'learn' the pattern of the enemy in 100 games. The current standard of playing Rock Paper Scissors are AIs that work with conditional probabilities. They usually do not require a fixed amount of games.

Rock Paper Scissors can be seen as the complete opposite of Go or chess, which AlphaZero was developed for. Opposite to a game with perfect information, it can be seen as a game with no information. The only information that describes the current turn is the card that the opponent picked. This information is however not available. Basically, a player misses 100% of the available information - this makes the Rock Paper Scissors an extreme case of a game with imperfect information. This makes the game an exciting challenge for AlphaZero. We already know that it plays Go and chess with superhuman strength. If it also manages to perform well in Rock Paper Scissors, it would show that AlphaZero is a general game AI which can play any game.

Encoding the state of the game is different from the other two games that are part of this thesis. As already mentioned, the only piece of information that exists during a turn, which is not the game history, is the card that the opponent picked. This information is however not accessible and the game would make no sense if it was. So the only data that can be encoded and supplied to the neural network are the moves that the player and the opponent made in the previous games or turns. One question that comes up is if it makes even sense to encode the moves made by the player itself. Looking at the current best performing AIs for Rock Paper Scissors, they use conditional probabilities on the last moves made by the opponent to predict what the opponent will play this turn. [29] The information about the own move history is not regarded. Now it is debatable if or if not this information is useful. One could argue that the own move pattern influences the opponent's moves and by predicting the opponent's move, the own move can be selected. In figure 3.10,

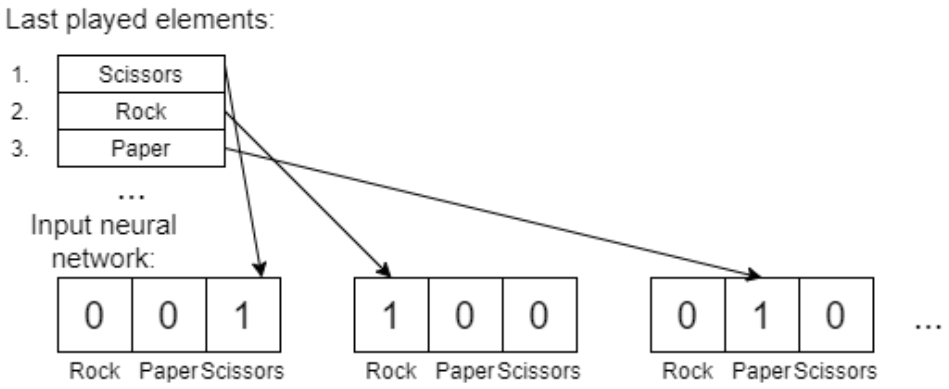


Figure 3.10: The figure shows the conversion of the opponents move history to an input for the neural network. Starting from the last played element by the opponent, the input is created. Each element is encoded in a sequence of one 1 and two 0s. The sequences are then concatenated to one 300 bit input, which can be consumed by the neural network. Source: self-created

the encoding of the move history is depicted. Each element picked is encoded in a sequence of two 0 and one 1. Rock is 100, Paper 010, and Scissors 001. The moves are encoded, starting from the last move that the opponent made. Now, all single sequences are connected, forming one big sequence with 300 fields containing 0 or 1. Because the conversion started from the last card played, the first three fields always symbolize the last move. The sequence can contain the data from up to 100 moves. This is the length of a Rock Paper Scissors game in this thesis. If there are not 100 recorded moves yet, the sequence is filled with 0 till 300 fields are reached. This is done because the neural network always needs an input of the same size.

The architecture of the neural network for Rock Paper Scissors is much simpler than for the other games. This due to two factors:

- There are no illegal moves in Rock Paper Scissors as you can choose each element each turn. This makes the second input layer unnecessary.
- The Monte Carlo tree search can not be conducted and thus, the output layer for the win value can be left out.

While the first point is quite self-explanatory, point two needs a bit of explanation. As in Take 6!, both players take their turn at the same time in Rock Paper Scis-

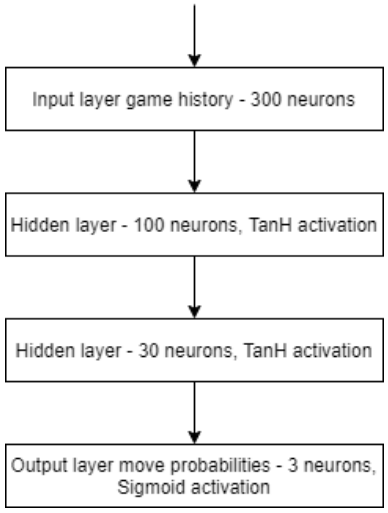


Figure 3.11: The architecture of the neural network that is used by AlphaZero for the game of Rock Paper Scissors. From the game state, the neural network predicts only the move probabilities $p$. Because a Monte Carlo tree search is not conducted, a win $v$ does not need to be predicted. Source: self-created

sors. For Take 6! this problem could be solved by predicting the opponents move probability with the neural network and use this information in the Monte Carlo tree search. Applying this solution to Rock Paper Scissors, however, would not produce any useful results. Because if one could already predict the move of the opponent, he would win in every case. The Monte Carlo search tree would always follow down the path that selects the element that counters the element predicted to be taken by the opponent. So basically, the whole Monte Carlo search tree is useless as it comes down to predicting the enemies move. Instead of taking a detour, one can directly take the prediction from the neural network and base his move decision on it. The architecture of the neural network can be seen in figure 3.11. It is a simple feed-forward neural network with four dense layers. One input layer with 300 neurons, two hidden layers with 100 and 30 neurons using the TanH activation function, and an output layer with three neurons using the sigmoid activation function. There are three output neurons because there are three possible elements a player can choose from. The output of the neural network is directly used to determine which element will be picked next. The neuron numbers of the hidden layers were chosen to scale down form the number of input neurons to the number of output neurons. The TanH activation function is supposed to produce better re-

sults. The encoding of the game history explains the 300 neurons of the input layer.

The output of the neural network $p$ are three values ranging from 0 to 1, representing a probability. Each value stands for one element in Rock Paper Scissors. The probability represents the chance of winning the turn by picking the corresponding element. The neural network directly outputs the recommendation of what to play this turn. Each turn AlphaZero selects one element randomly, using the generated probability distribution and plays this element. This stands in great contrast to the usual AlphaZero procedure, where a Monte Carlo tree search would be conducted to create the final move probabilities. Each turn during training, AlphaZero saves the state $s$, which is, in this case, the game history. Furthermore, the element picked by the opponent is saved. From the element picked by the opponent, it is easily possible to calculate which element should have been picked to win this turn. Converted to a probability distribution where the winning element has a probability of 1 and the other elements of 0, this will be the target value $e$. For training, $s$ is used as input and $e$ as the target value. The loss is calculated by determining the cross-entropy between $p$ and $e$ for the state $s$:

$$loss = -e \log p$$

for

$$(p) = f_\theta(s)$$

The goal is that $p$ is $e$ or, in other words: That the neural network will predict the element that will beat the element selected by the opponent. One 'game' consists of 100 'turns' after which the neural network is trained, and the game history is reset. As for the other games, the optimizer used in this thesis is Adam, with a learning rate of 0.0001. Tensorflow recommends this learning rate.

The average training game with the parameters mentioned above takes 200ms. The seems like a lot. It should however be considered that one game has 100 turns, so 100 predictions from the neural network are needed. For 10,000 training games, this means a training time of roughly 30 minutes.

# Chapter 4

# Experimental Evaluation

In this chapter, the AlphaZero implementations of chapter 3 will be put to the test. For each game, one comparison AI will be implemented. The comparison AIs are based on hard-coded rules. In the first part of this chapter, the rules of the AIs will be presented. Afterward, AlphaZero is compared to the newly created AI during different stages of the training. The goal is to show that the longer AlphaZero trains, the better it performs against the comparison AI and can even surpass it during training. The last section of this chapter will examine the playstyle of AlphaZero.

## 4.1 Base line AIs

The selection of rule set for the comparison AIs is a balancing act. Several factors need to be considered:

- The resulting AI should provide a strong gameplay. Having a too weak AI that e.g., plays cards just by random will not prove that AlphaZero can learn a game to human or even superhuman strength.
- It should be possible to beat the AI. In games like Tik Tak Toe, it is possible to create an AI that plays so strong that the best possible outcome for the game is a draw. Comparing AlphaZero against such an AI would only prove that AlphaZero manages to create a draw himself every game. The goal however, is that AlphaZero plays to win.
- The AI should be able to calculate its move reasonably quickly. AlphaZero already needs a long time for training and inflating this time by conducting hundreds of test games at different training steps will worsen the situation.

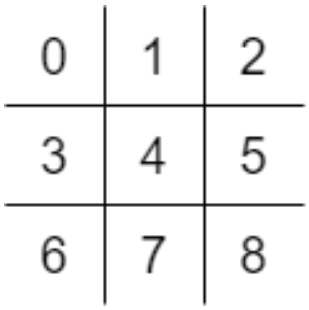


Figure 4.1: The classification of fields on a Tic Tac Toe board. Each area got a number assigned to allow easy and unambiguous addressing. Source: self-created

Therefore the AIs should work without expensive operations such as backtracking and brute force.

- The AI should have a human-like playstyle to have some correlation between the experiment and an actual game against humans.

The following sections will present the rule sets for the comparison AIs and explain their playstyle.

### 4.1.1 Tic Tac Toe

A strong AI for Tic Tac Toe can be made relatively easy. The algorithm 2 shows schematically how the comparison AI works. As a parameter, the AI gets all the information about the fields of a Tic Tac Toe board. This information includes: if the field is empty, if the player occupies the field and if the opponent occupies the field. The expected return value of the algorithm is an integer value from 0 to 8 that stands for the field that the comparison AI will put its mark on. Which field is addressed with which number can be seen in figure 4.1.

First, the AI checks if there is any field on the board that is now empty and would lead to victory if the AI would put its mark there. If there is such a field, then the number of this field will be returned. This means that the algorithm first checks if it can win this turn. If so, it will choose this option. If it can not win this turn, the AI checks if there is any field on the board that is now empty and would lead to victory for the opponent if the opponent would put its mark there. If there is such a field, then the number of this field will be returned. This means that the algorithm checks if the opponent can win the next turn. If so, it will choose the

field the opponent could win with in order to prevent the enemy from gaining a victory. A draw is preferred before a loss. If the AI can neither win nor prevent the opponent from winning this turn, it will check if field 4 is empty and put its mark there. Field 4 is the field in the middle of the Tic Tac Toe board. The middle field provides huge tactical advantages and thus should always be preferred before the other fields. In case the middle is already occupied, the algorithm will select a random number from 0 to 8. Now it will check if the field with the randomly selected number is empty. If so, the number of this field will be returned. If not, then again, a random number is selected till a field is found that is not empty.

One could argue that this algorithm uses brute-force as an expensive operation because it tries out all fields if they could lead to victory or loss. While it is true that this is bute force, the cost of operation is still low. This is due to the fact that a Tic Tac Toe board is nine fields small and the algorithm only checks all possibilities one turn in the future. This is why the calculation time is still acceptable.

In order to prove that the AI has a strong gameplay but is still beatable, a small experiment is conducted. The AI has to play 10000 games against a player who selects a field entirely randomly. The result with the random player starting:

Table 4.1:

| won | draw | lost |
|---|---|---|
| 8308 | 1488 | 204 |

And for starting with the comparison AI:

Table 4.2:

| won | draw | lost |
|---|---|---|
| 9556 | 412 | 32 |

For a simple game like Tic Tac Toe, 10000 games should be more than enough to be statistically relevant. The results show clearly that the comparison AI plays much better than a random player. At the same time, it is clear that it is possible to beat the comparison AI which was one goal formulated at the beginning of this chapter. The experiment proved that the ruleset of the comparison AI can be used for testing against AlphaZero.

**Algorithm 2**

TICTACTOE AI(fields)

```
1:  for all field in fields do
2:      if field.isEmpty() then
3:          field.setPlayerMark()
4:          if playerHasWon(fields) then
5:              return field.number()
6:          end if
7:          field.setEmpty()
8:      end if
9:  end for
10: for all field in fields do
11:     if field.isEmpty() then
12:         field.setOpponentMark()
13:         if opponentHasWon(fields) then
14:             return field.number()
15:         end if
16:         field.setEmpty()
17:     end if
18: end for
19: if fields[4].isEmpty() then
20:     return 4
21: end if
22: randomNumber = random(0, 8)
23: while !fields[randomNumber].isEmtpy do
24:     randomNumber = random(0, 8)
25: end while
26: return randomNumber
```

### 4.1.2 Take 6!

Unlike for Tic Tac Toe, there is not much literature on Take 6!. Also, Take 6! is much more complicated then Tic Tac Toe. This is why many more guesses and assumptions need to be made for a strong Take 6! AI. The algorithm is big to be entirely written here. This is why it is just written conceptually. For the program code, please check the appendix.

1. Select the batch with the least cards.
2. If there are multiple batches with the lowest number of cards, then save them all.
3. For each of the saved batches, get the highest card of the batch.
4. For each of the highest cards, get all the cards on the hand of the player that are bigger.
5. Calculate from all of the highest cards in the selected batches the difference to the cards on the hand that have a bigger number.
6. Select the card that has the lowest difference to any of the highest cards in the selected batches.
7. If there are multiple such cards, then choose the one with the lowest number.
8. If there is no card on the hand that is higher then any of the highest cards in the batches, then play the lowest card on the hand.
9. Select the batch with the lowest amount of penalty points.
10. return the selected card and the selected batch.

Preferring the batch with the least cards will lead to fewer instances where a batch will be full, and the player has to take it. Choosing the card with the lowest difference - thus, the lowest card possible for the batch is preferable for the algorithm. This is done because lower cards get played earlier, which decreases the chance that an opponent plays a card that unfavorably changes the batch. If there is no card on the hand which is higher then any of the highest cards in the batches, the lowest card on the hand is selected. This is done to get rid of very low cards. Very low cards are a problem as they frequently lead to taking a whole batch as a penalty. Getting rid of them when the player has to take anyway a penalty is therefore beneficial. If the player has to take a batch as a penalty and can choose which batch, it will always prefer the one with the least penalty points.

The algorithm is pretty lightweight as it consists merely of some not nested for loops and if-clauses. Neither brute-force or backtracking methods are applied. Performance was especially crucial for the Take 6! comparision AI as AlphaZero took already much time to train for Take 6!.

Unlike as for the Tic Tac Toe AI, for someone unfamiliar with the game, it is not visible at a glance that the comparison AI much measurably better than a player who plays cards randomly. In order to prove that the comparison AI plays much better, an experiment is conducted. The AI has to play 10000 games against a random player. The setup of the game is:

- 104 cards
- 10 cards per player
- 3 players all in all
- 1 player is the comparison AI
- 2 random players
- 4 batches used
- A game ends when all cards are played
- Player gets 1 point if he finishes with the least penalty points
- Player gets -1 point if he finishes with the most penalty points

With this setup, the following results for the comparison AI were achieved:

- Lowest penalty points: 6811 games.
- Neither highest or lowest penalty points: 2221 games.
- Highest penalty points: 968 games.

In the introduced score system, this means that the comparison AI achieved 5843 points on a scale from -10000 to 10000 points. 0 would mean that the comparison AI has the same strength as a random player. The results shows that the comparison AI is measurably stronger than a random player and thus can be taken as a good baseline for AlphaZero.

### 4.1.3 Rock Paper Scissors

There is much research for playing Rock Paper Scissors with an AI. Unlike the more exotic Take 6! there are a lot of already existing solutions available. It is mentioned that using conditional probabilities to predict the next move of the opponent leads to the best result in playing Rock Paper Scissors. [29] Because the game is so easy and lightweight, there are a lot of ready to play implementations on the web. A lot of those implementations use conditional probabilities to predict the move of the opponent and thus the own next move. For this thesis the AI of the university of Stirling [30] was chosen as comparison AI. The reason for this is that it has the most stable API that can handle a multitude of requests in a reasonable amount of time.

The algorithm saves all the moves of the opponent. When selecting a card, it takes the last three elements played by the opponent. It calculates the conditional probabilities for the element that will be played next, given the last three elements. Knowing now what the opponent will play, the algorithm selects the card that will beat the predicted card. The more this algorithm plays against an opponent, the more it will learn his 'movement pattern'. The idea behind this is that most human players do not choose an element completely randomly in Rock Paper Scissors. Most players do have a movement pattern or a slight 'tick' in selecting their next element. If one manages to learn this pattern, he can beat a player in the long run. Thus this algorithm will perform in the best case average in the beginning but will become stronger and stronger each turn. This fits the rules declared in chapter 3, where it is stated that one game of Rock Paper Scissors consists of 100 turns. This should give the algorithm enough time to learn eventual patterns in the moves of AlphaZero and counter them.

Unlike for the other comparison AIs, a test against a random player is not needed. It is already proven that the algorithm works. Furthermore, a test against a random player would prove nothing as the conditional probabilities will not work with a completely random player. A test against a player that has a particular pattern is also not needed, as it is clear that the comparison AI would clearly win.

## 4.2 Experiments

The experiments conducted in this thesis all follow the same pattern. For each game, an untrained AlphaZero version, as introduced in chapter 3, will train itself for a certain amount of training games. During the training, at regular intervals, a certain amount of test games are conducted against the previously introduced

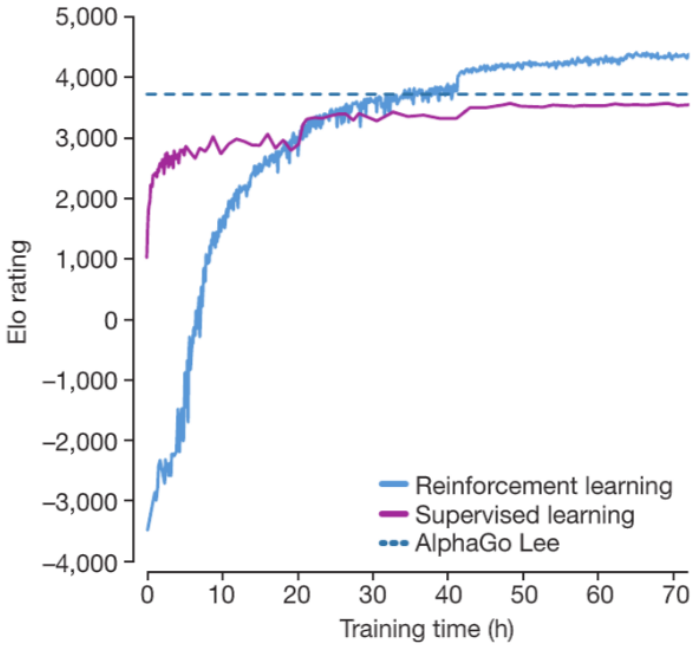


Figure 4.2: Plot of the Elo rating of AlphaZero in Go during the training. This plot is taken form the original paper and is the model plot for the experiments. Source: [5]

comparision AIs. All the results of the test games during one interval are combined into a single score. The score will then be plotted depending on the amount of training games conducted. The plot should prove that:

- The more training is done, the higher the score gets. This will show that the algorithm works as intended.
- AlphaZero will have at some point during the training a higher score than the comparison AI.
- The AlphaZero versions in this thesis behave like the original version whos plot was published in [5] and can be seen in figure 4.2.

After conducting the experiments for all games, the results can be compared to each other. Here the question of this thesis can be answered if AlphaZero can master games with imperfect information the same way as it can master games with perfect information.

### 4.2.1 Tic Tac Toe

The experiment for Tic Tac Toe was carried out with the following parameters:

- 20000 training games overall.
- A test is conducted every 100 training games.
- 100 test games are carried out each test.

As already stated, is AlphaZero for Tic Tac Toe performance wise very efficient. This allows to carry out so many test games so often. 20000 training games is enough for AlphaZero to learn the game perfectly. The following rules calculate the score that AlphaZero gets during a test against the comparison AI:

- All players start with 0 Points
- After a victory, the score is increased by 1
- After a loss, the score is decreased by 1
- Draws are ignored

The advantage of this Elo rating is that a value above means that AlphaZero plays better than the comparison AI. The resulting plot can be seen in figure 4.3. What becomes clear is that after roughly 1000 games, AlphaZero overcomes the comparison AI consistently. After 7000 games, AlphaZero rises to an Elo level of 50. This

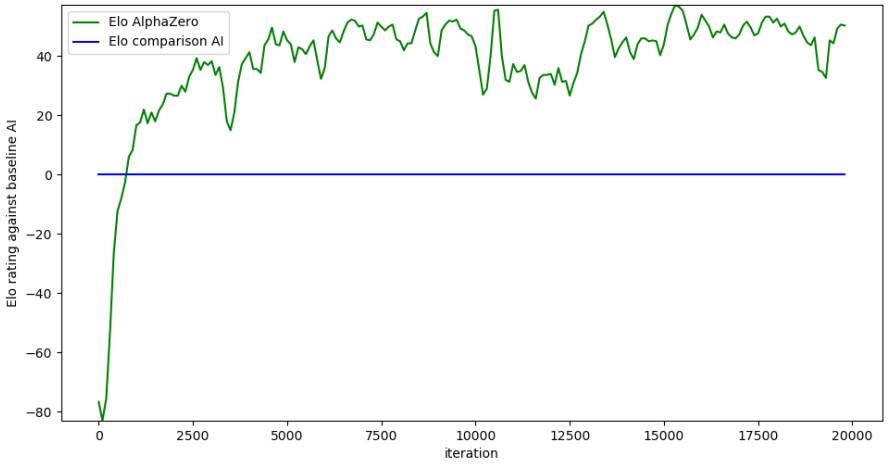


Figure 4.3: The green graph symbolizes the Elo rating of AlphaZero against a baseline AI in Tic Tac Toe, depending on how many training iterations where conducted. The blue line symbolizes the Elo rating of the comparison AI. Being above the blue line means that AlphaZero plays stronger than the comparison AI and vice versa. At the beginning of the training, the score rises more quickly then later. After roughly 1000 iterations, AlphaZero becomes stronger than the comparison AI. Source: self-created

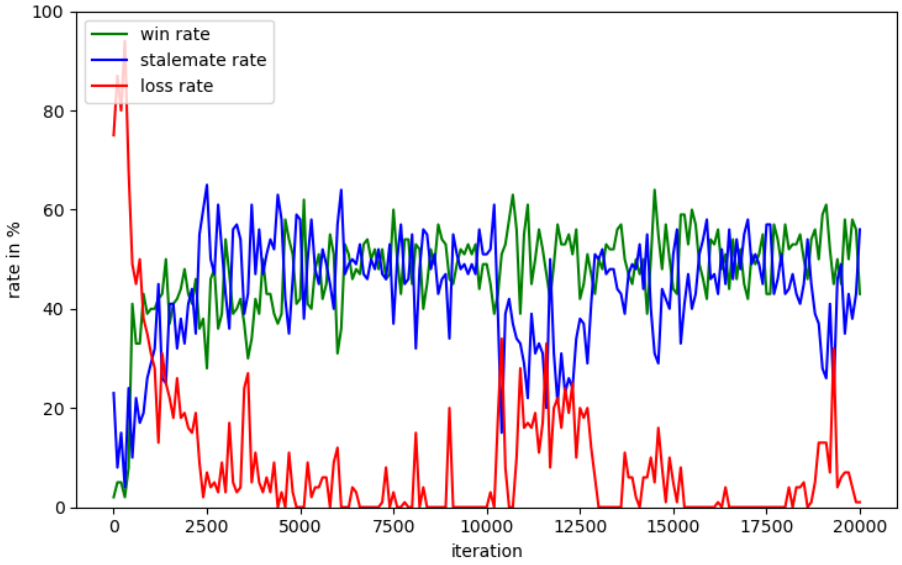


Figure 4.4: The red graph shows the loss rate in percent of AlphaZero versus the comparison AI, depending on the number of training iterations conducted. The green graph shows the win rate and the blue graph the rate of the stalemates. What is visible is that at the later stages of training, the loss rate drops to zero or close to it. This means that AlphaZero does not lose a single game anymore. Source: self-created

seems to be the maximum which can be achieved. With 100 Elo points possible, it seems that AlphaZero does not play perfectly. Investigating the test games in figure4.4 further, shows that roughly 50% of the games end in a draw. This happens since the comparison AI is quite competent, and a Tic Tac Toe game will always end in a draw if two perfect players play against each other. This means that an Elo value of 50 is the maximum that can be achieved against the comparison AI. This means that after 7000 training games, AlphaZero plays a perfect round of Tic Tac Toe. This is a good but also expected result. It proves however, that the AlphaZero implementation developed during this thesis works as the original algorithm.

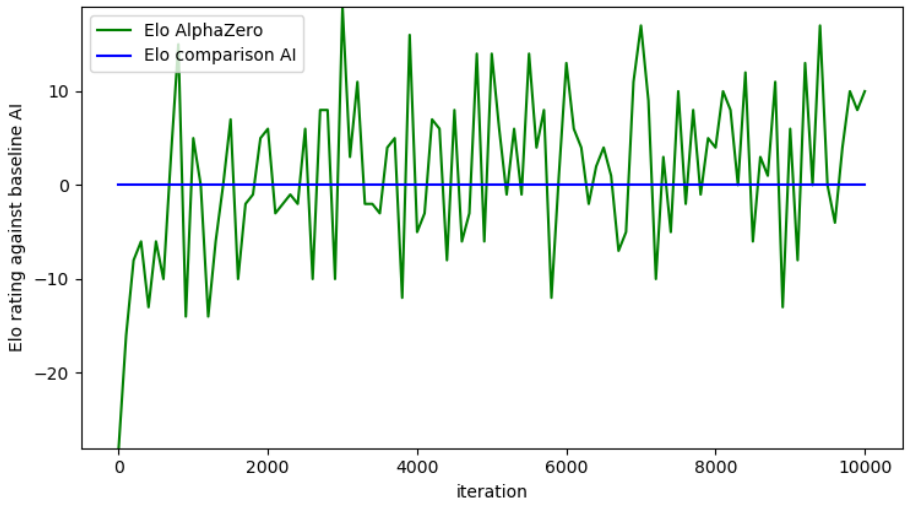


Figure 4.5: The green graph symbolizes the Elo rating of AlphaZero against a baseline AI in Take 6!, depending on how many training iterations where conducted. The blue line symbolizes the Elo rating of the comparison AI. Being above the blue line means that AlphaZero plays stronger than the comparison AI and vice versa. The plot is much fuzzier than the for the original AlphaZero experiment or Tic Tac Toe. The form of the graph is however similar. After roughly 2000 iterations, AlphaZero becomes stronger than the comparison AI. Source: self-created

### 4.2.2 Take 6!

The experiment for Take 6! was carried out with the following parameters:

- 10000 training games overall.
- A test is conducted every 100 training games.
- 100 test games are carried out each test.
- There are three players in the test games, one represented by AlphaZero and two by the comparison AI.
- Training is conducted with three players represented by AlphaZero.

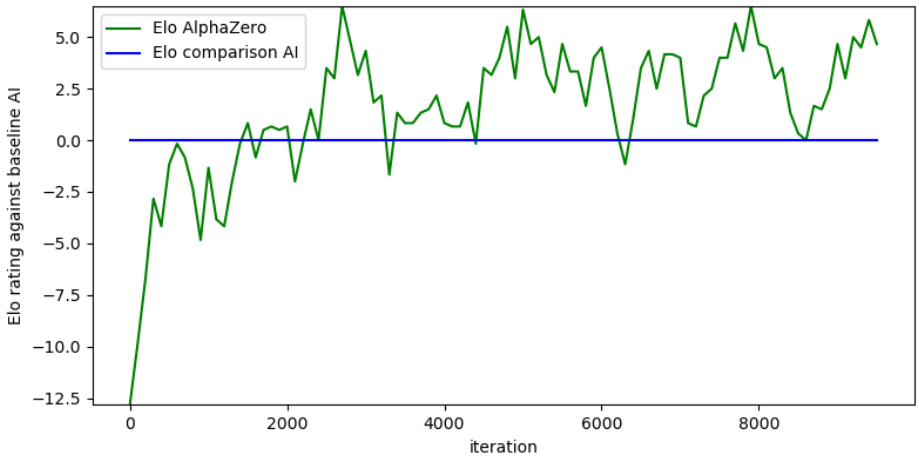


Figure 4.6: The plot shows the moving averages for six values from 4.5. The trend is easier to recognize than in the original. Source: self-created

The setting for the game is the same as introduced in chapter 4.1.2. For the Take 6! experiment, the performance is a much bigger problem then for Tic Tac Toe. The experiment took 40 hours. The Elo value is calculated as in chapter 4.1.2, meaning +1 point if the player got the least penalty points and -1 if he got the most penalty points. The resulting plot can be seen in figure 4.5. Considering that both comparison AIs play with the same 'strength', an Elo value below 0 means that AlphaZero plays worse than the comparison AI and a value above 0 means that AlphaZero plays better. Visible is that the graph is much fuzzier than the graph from Tic Tac Toe. The most logical explanation for this phenomenon is that the random element of Take 6! causes this fuzziness and 100 test games per session are not enough to even out the random effects. However, there were already performance problems for 100 test games per session, so a new experiment with 1000 test games was not conducted. Instead, a new graph is created that uses the moving averages for six values from the raw data. The resulting plot can be seen in figure 4.6. Using moving average values made the graph a lot 'sharper' which allows a better interpretation. AlphaZero needs roughly 2000 training iterations to play better than the comparison AI. What is interesting is that even in the later stages of the training, AlphaZero can drop below the level of the comparison AI. One reason for this can be that the comparison AI is very strong. Another reason

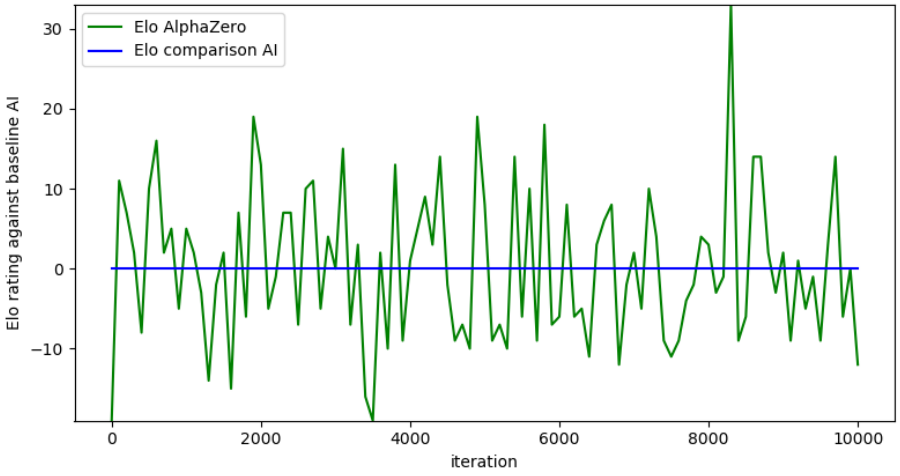


Figure 4.7: The green graph symbolizes the Elo rating of AlphaZero against a baseline AI in Rock Paper Scissors depending on how many training iterations where conducted. The blue line symbolizes the Elo rating of the comparison AI. Being above the blue line means that AlphaZero plays stronger than the comparison AI and vice versa. The plot is very fuzzy, and no trend can be observed. This means that the training does not influence the performance of AlphaZero on Rock Paper Scissors. Also interesting is that the green line always fluctuates around the blue line. This means that AlphaZero and the comparison AI have roughly the same strength. Source: self-created

can be that AlphaZero has a much harder time to optimize its gameplay for a game with imperfect information than for a game with perfect information. The rise of the skill level for Tic Tac Toe was way more consistent and clear. However, it is still observable that the training increased the Elo rating and that a trained AlphaZero plays on average better than the comparison AI. The Elo value has a possible range from -100 to 100. There are however, only values from -25 to 18 recorded.

### 4.2.3 Rock Paper Sissors

The experiment AlphaZero versus the comparison AI for Rock Paper Scissors was carried out with the parameters:

- 10000 training games overall.
- A test is conducted every 100 training games.
- 1 test games is carried out each test.
- One game consists of 100 turns.
- Winner of a game is who won the most turns in a game.

A game of Rock Paper Scissors has 100 turns, which is much more than in a game of Take 6! (10) or Tic Tac Toe (9). Furthermore, the comparison AI is a web-based application that has to be queried every turn. This is performance-wise very inefficient. Because of this, only one test game is carried out every hundred training games. To still receive a useful plot, the points achieved in each of the test games are plotted. The points are calculated similar to the other games, meaning:

- All players start with 0 Points
- After won turn, the score is increased by 1
- After a lost turn, the score is decreased by 1
- Draws are ignored

The resulting plot can be seen in figure 4.7. In this plot is no trend visible. This means that the training does not increase the strength of AlphaZero for Rock Paper Scissors. Furthermore, AlphaZero is not visibly better than the comparison AI. Instead, it looks like a pretty random pattern. The mean of all the test game Elo values is 0.38, which means that during all the training, AlphaZero was a bit better than the comparison AI. This is however hardly significant. What this plot might hide is a high number of games that end in a draw. If all games ended in a draw, the Elo value would be 0. This would however be a very impressive result. To check this theory, the rates of victories, losses, and draws are tracked for all the tests. The resulting plot can be seen in 4.8. What becomes clear is that the ratio of wins, losses and draws roughly stays the same. This is what one would expect if two players play against each other that pick their elements entirely at random.

The experiment has shown that the AlphaZero implementation introduced in this thesis is not suitable for learning and playing the game Rock Paper Scissors at a higher level than the comparison AI. What is however, also interesting is that AlphaZero does not play worse than the comparison AI. What the result does not tell however, is if AlphaZero follows any strategy at all or picks the elements at

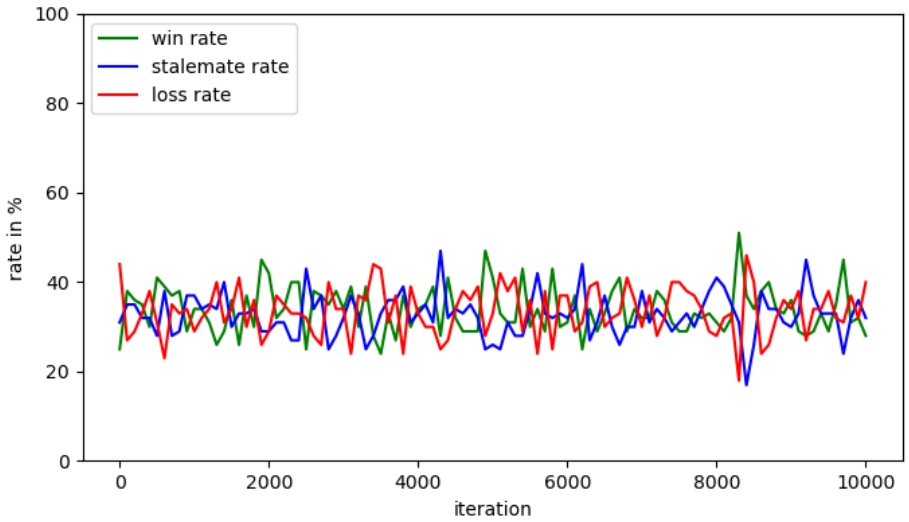


Figure 4.8: The red graph shows the loss rate in percent of AlphaZero versus the comparison AI in depending on the number of training iterations conducted. The green graph shows the win rate and the blue graph the rate of the stalemates. What becomes clear is that at any stage of the game, the win-, loss-, and stalemate rate is always around the 33% line. Source: self-created

random. Picking all the elements entirely at random will always lead to an equal distribution of victories and losses as no opponent can find any pattern in the move history. If AlphaZero has any strategy will be answered in the next section.

## 4.3 Behaviour AlphaZero

This section will answer the question of how the playstyle of AlphaZero is. It will explain if the AI follows specific tactics and gives an explanation of why the plots presented in the previous section look how they look.

### 4.3.1 Tic Tac Toe

For investigating the playstyle of AlphaZero in Tic Tac Toe, 1000 games by AlphaZero, after it finished its training from the former section, versus the comparison AI where conducted. In each of the games, for each turn, the area was recorded where the AI put its mark. This allows us to find out which 'combinations' are played very often. In theory, there are 255,168 different games of Tic Tac Toe. The recorded games showed, however, that there are only ten different courses of the game that therefore appear often:

| Move | | | | | | | | | Game ended with | Times recoreded |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | | |
| 4 | 7 | 6 | 2 | 0 | 3 | 8 | - | - | victory | 131 |
| 4 | 8 | 7 | 1 | 2 | 6 | 5 | 3 | 0 | draw | 128 |
| 4 | 0 | 3 | 5 | 1 | 7 | 2 | 6 | 8 | draw | 125 |
| 4 | 2 | 5 | 3 | 0 | 8 | 1 | 7 | 6 | draw | 125 |
| 4 | 5 | 7 | 1 | 6 | 2 | 8 | - | - | victory | 124 |
| 4 | 1 | 6 | 2 | 0 | 3 | 8 | - | - | victory | 123 |
| 4 | 3 | 7 | 1 | 6 | 2 | 8 | - | - | victory | 116 |
| 4 | 6 | 3 | 5 | 7 | 1 | 0 | 8 | 2 | draw | 102 |
| 4 | 6 | 3 | 5 | 7 | 1 | 2 | 0 | 8 | draw | 14 |
| 4 | 6 | 3 | 5 | 7 | 1 | 2 | 8 | 0 | draw | 11 |

It is visible that AlphaZero always starts in the middle of the board. It seems to evaluate this as the most important position to hold. Furthermore, it becomes clear that AlphaZero never actually loses. It can always at least come up with a draw. The most common course of the game is depicted in figure 4.9. What becomes clear is that the comparison AI lost basically after its first turn as AlphaZero constructs a pinch grinder that leads finally to victory. These are advanced tactics that require

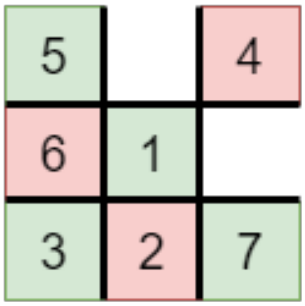


Figure 4.9: The Most common course of game between AlphaZero and the comparison AI. The fields are enumerated in the order they were selected by the AIs. The green fields are selected by AlphaZero and the red fields by the comparison AI. Source: self-created

to think some turns ahead. It shows that the AlphaZero implementation optimally learned to play Tic Tac Toe.

### 4.3.2 Take 6!

To find out how AlphaZero plays Take 6!, 1000 games between 3 players, all played by the trained AlphaZero implementation, are conducted. Depending on the turn number, the following data is recorded:

- The average number on the card that is played.
- The average amount of penalty points per player.
- The average amount of cards in a batch.

The following result was recorded:

| Category | Turn | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| av. number on card | 18 | 25 | 32 | 41 | 64 | 68 | 71 | 72 | 74 | 76 |
| av. penalty points | 0.5 | 1.1 | 2.1 | 3.4 | 4.8 | 6.5 | 8.2 | 9.7 | 11.5 | 13.1 |
| av. card in batch | 1 | 1.8 | 2.5 | 3.1 | 3.8 | 4.2 | 4.3 | 4.3 | 4.2 | 4.3 |

What becomes clear from the data is that:

- Cards with lower numbers get played first; higher numbers are kept for the late game.

- AlphaZero takes penalty points early. It throws away low cards and gets penalty points for it right at the beginning, even when it could avoid this.
- The average amount of cards in a batch stays for the most time close to the maximum amount of cards in a batch. This means that AlphaZero tries to 'fill' up all batches to avoid penalty points.

This gives a good overview of how AlphaZero plays. Putting all this information together, one can easily see what AlphaZero tries to achieve with its playstyle: It tries to create a situation where all batches are full. The player who has the lowest cards left now will most likely have to take a whole batch as a penalty. This 'empty batch' will then be filled again by the players who had higher cards left. Next turn, the player who has only low cards left, has the same problem. This is why AlphaZero first gets rid of its lower cards when its still 'cheap' penalty points-wise. Once the batches are all filled up, AlphaZero gains its ground back. This is an impressive tactic for an AI as it requires a proper long term planning and the right balance of short term penalties and long term benefits. Mastering Take 6! with the presented AlphaZero implementation can be seen as a success.

Below, two tables show the course of a Take 6! game. One human player and two AlphaZero AIs played for victory. The game was played with the settings: 10 cards per player, 61 cards overall, and three batches. The human player is an experienced Take 6! player. The two AIs are trained in 10,000 self-play games. Limiting the number of cards to 61 helps AlphaZero to predict the cards on the opponent's hand and speeds up training by a margin. AI 1 played a nearly perfect game, behaving as expected. It got rid of its low cards early in the game, taking penalty points right from the start and then overcoming its opponents in the late game. With this play style, it was able to beat even the experienced human player. AI 2 shows a not expected behavior by not getting rid of its low cards early and amassing many penalty points towards the end of the game. This unusual behavior might be caused by only having one low card. Furthermore, Batch 1 was 'blocked' by the highest card through most of the game, which led to the game revolving around the two remaining batches. Maybe the 10,000 training games were not enough to train AlphaZero in a way that it plays perfectly every game. David Silver describes that AlphaZero can succumb to so-called 'illusions' where it completely overestimates its advantage and thus skewing the selection of the next move. [4]

| | Card played | | | | | |
|---|---|---|---|---|---|---|
| Turn | Human | AI 1 | AI 2 | Batch 1 | Batch 2 | Batch 3 |
| 1 | 46 | 1 | 52 | 42 | 11 | 51 |
| 2 | 4 | 56 | 43 | 1 | 11, 46 | 51, 52 |
| 3 | 48 | 6 | 50 | 1, 4, 43 | 11, 46 | 51, 52, 56 |
| 4 | 60 | 2 | 58 | 1, 4, 43, 48, 50 | 6 | 51, 52, 56 |
| 5 | 23 | 14 | 59 | 58, 60 | 6 | 2 |
| 6 | 13 | 19 | 22 | 58, 60 | 6, 14, 23, 59 | 2 |
| 7 | 30 | 24 | 49 | 58, 60 | 6, 14, 23, 59 | 2, 13, 19, 22 |
| 8 | 37 | 41 | 29 | 58, 60 | 6, 14, 23, 59 | 30, 49 |
| 9 | 33 | 44 | 0 | 58, 60 | 29, 37, 41 | 30, 49 |
| 10 | 18 | 57 | 20 | 58, 60 | 29, 37, 41, 44 | 0, 33 |
| End | – | – | – | 18, 20 | 29, 37, 41, 44, 57 | 0, 33 |

| Turn | Penalty Human | Penalty AI 1 | Penalty AI 2 |
|---|---|---|---|
| 1 | 0 | 0 | 0 |
| 2 | 0 | 1 | 0 |
| 3 | 0 | 1 | 0 |
| 4 | 0 | 7 | 0 |
| 5 | 0 | 10 | 7 |
| 6 | 0 | 10 | 7 |
| 7 | 0 | 10 | 7 |
| 8 | 9 | 10 | 7 |
| 9 | 9 | 10 | 11 |
| 10 | 9 | 10 | 15 |
| End | 13 | 10 | 15 |

### 4.3.3 Rock Paper Scissors

For Rock Paper Scissors, the question remained last section if AlphaZero learned anything during the training or if it just kept playing cards at random. To answer this question, a small experiment is conducted. AlphaZero will play against a 'dumb' comparison AI that picks 'Rock' every turn. If AlphaZero has any strategy, a clear result can be expected. The experiment will take place with the same parameters, as introduced in the previous section. AlphaZero will play 10000 training games against itself and conduct a test game against the dumb comparison AI every 100 games. The result can be seen in figure 4.10. The graph looks very much like figure 4.7 from the first experiment where AlphaZero had to compete

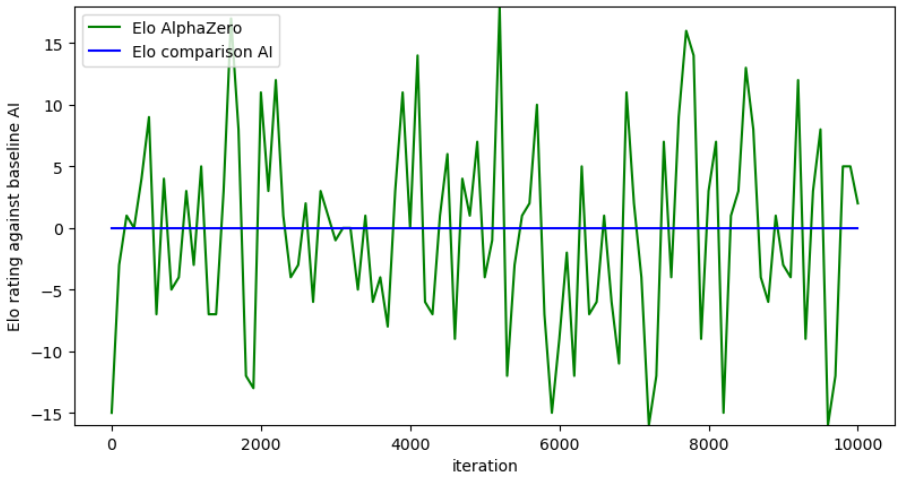


Figure 4.10: The green graph symbolizes the Elo rating of AlphaZero against a dumb AI in Rock Paper Scissors depending on how many training iterations where conducted. The blue line symbolizes the Elo rating of the comparison AI. Being above the blue line means that AlphaZero plays stronger than the comparison AI and vice versa. The dumb AI just plays 'Rock'. No progress during training is visible. Source: self-created

against a 'clever' AI. This can only mean that AlphaZero chooses the element it plays completely randomly regardless of the opponents tactic. This also happens at every stage of the training. This is a hint that AlphaZero is not able to learn and play a game that has 'no information'.

A possible reason why this happens is that playing completely random is a strong strategy against a player who tries to predict you. AlphaZero learns to play the game by playing against itself. It tries to predict the opponent and to choose the counter element. By doing so, it becomes however predictable for the opponent who is represented by the same AlphaZero version. This leads to a situation where AlphaZero tries to play more random not to be predicted rather than trying to predict the opponent. It prefers a more 'defensive' strategy as it seems to be more rewarding. This is however just a theory and not proven by experiment.

# Chapter 5

# Conclusion

This chapter will summarize the findings of this thesis and give an outlook on possible improvements to the AlphaZero implementations presented.

## 5.1 Summary

As expected, AlphaZero performed flawlessly in learning and playing the game Tic Tac Toe. This was however, no surprise as Tic Tac Toe is a perfect information game and rather simple too. AlphaZero was specifically made with a perfect information game in mind and proved already that it can achieve superhuman strength in this type of game. The excellent results in mastering this perfect information game showed however, that the AlphaZero implementation works as intended. Tic Tac Toe can be seen as the calibration of the AlphaZero algorithm. It became clear that the implementation created in this thesis is calibrated nearly perfectly. This allowed us to move the actual research question of this thesis: Can this success be repeated in imperfect information games? Moreover, if so, does this even apply to games with no information such as Rock Paper Scissors?

The results show that for imperfect information games, AlphaZero can perform well. In Take 6! it is able to beat a quite strong comparison AI. Training however, becomes much less linear, and the skill level starts to fuzzy out. Due to the random element in the game, AlphaZero has a much harder time to learn and to beat the opponent consistently. Even in the late stages of the training, losses will still happen regularly. The less reliable information there is, the harder it becomes for AlphaZero to find a consistent strategy. Take 6! however, seems to provide enough information for AlphaZero to develop a game plan that it sticks to. This game plan can even help with the development of other AIs as basic guidelines.

A game with no information, besides the historical game data of the opponent, seems too much to chew for AlphaZero. In Rock Paper Scissors, the success from the other two games could not be repeated. No amount of training changed the play style of the algorithm, which was just a completely random selection of the next element. This resulted in AlphaZero not being able to beat the most simple opponent one can come up with in Rock Paper Scissors. It was however, also not losing against stronger opponents as the random play style of AlphaZero prevented that the conditional probability tactic could work. Why this happens is undoubtedly an interesting question for further research. One possible explanation is that in order to win a game of Rock Paper Scissors against yourself, you have to play in such a way that you do not know what you will play next. This could explain the random play style of AlphaZero.

AlphaZero proved to be a flexible, easy to implement, and a strong solution to create a game AI. The implementation part of this thesis showed that regardless of the game, the code base of AlphaZero remains the same. The only thing a developer has to think about is how to convert the state of the game to an input for the neural network and how to convert the output from the neural network back to an action in the game. For future game developers, it might even be time-saving to use AlphaZero to create a game AI instead of developing something new from scratch.

## 5.2 Future Work

As already mentioned in previous chapters, there are several unanswered research questions left out. First and foremost, the parameters used in the experiments could be less arbitrarily. Parameters include, for example, the layout of the neural network, the depth of the tree search, or the optimizer used by AlphaZero. Unfortunately, the original paper gave no hint of how these parameters could be chosen depending on the game. Also, other papers that did a reimplementation on AlphaZero used rather arbitrary selected parameters. [5, 11, 10, 9] An interesting topic would be, for example, if there is a correlation between the average amount of turns in a game and the optimal depth of the search tree. The correlation between the input size and the optimal amount of neurons in the hidden layer and many more would surely help future research.

Another topic that could be covered by future research is the conversion of the state of a game to an input for the neural network. Can consistent rules be formu-

lated for this?

This thesis has shown that AlphaZero can learn to play games with imperfect information; games with no information however, are not suited for the algorithm. The question is: Where is the border between games suited for AlphaZero and games not suited for the algorithm? Maybe it is possible to find a degree of 'available information' that marks this border.

AlphaZero was developed very recently. The algorithm may be improved and extended. With this new algorithm, it is maybe possible to tackle problems outside of the world of games. If this happens, AlphaZero might even start to influence our daily life.

# Bibliography


[1] T. J. Schaefer, "On the complexity of some two-person perfect-information games," vol. 16, no. 2, pp. 185–225. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0022000078900454

[2] M. Campbell, A. J. Hoane, and F.-h. Hsu, "Deep blue," vol. 134, no. 1, pp. 57–83. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0004370201001291

[3] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of go with deep neural networks and tree search," vol. 529, no. 7587, pp. 484–489.

[4] L. Fridman, "David silver: AlphaGo, AlphaZero, and deep reinforcement learning | AI podcast #86 with lex fridman." [Online]. Available: https://www.youtube.com/watch?v=uPUEq8d73JI

[5] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. P. Lillicrap, K. Simonyan, and D. Hassabis, "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," vol. abs/1712.01815. [Online]. Available: http://arxiv.org/abs/1712.01815

[6] S. Strogatz, "One giant step for a chess-playing machine." [Online]. Available: https://www.nytimes.com/2018/12/26/science/chess-artificial-intelligence.html

[7] S. Parsch, "Vergesst AlphaGo – der neue held heißt AlphaZero." [Online]. Available: https://www.welt.de/wissenschaft/article185109198/Vergesst-AlphaGo-der-neue-Held-heisst-AlphaZero.html

[8] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis, "A general reinforcement learning algorithm that masters chess, shogi, and go through self-play," vol. 362, no. 6419, pp. 1140–1144, publisher: American Association for the Advancement of Science Section: Report. [Online]. Available: https://science.sciencemag.org/content/362/6419/1140

[9] Y. Tian, J. Ma, Q. Gong, S. Sengupta, Z. Chen, J. Pinkerton, and C. L. Zitnick, "ELF OpenGo: An analysis and open reimplementation of AlphaZero," vol. abs/1902.04522. [Online]. Available: http://arxiv.org/abs/1902.04522

[10] H. Charlesworth, "Application of self-play reinforcement learning to a four-player game of imperfect information." [Online]. Available: http://arxiv.org/abs/1808.10442

[11] Q. Jiang, K. Li, B. Du, H. Chen, and H. Fang, "DeltaDou: expert-level doudizhu AI through self-play," in *Proceedings of the 28th International Joint Conference on Artificial Intelligence*. AAAI Press, pp. 1265–1271.

[12] N. Petosa and T. Balch, "Multiplayer AlphaZero." [Online]. Available: http://arxiv.org/abs/1910.13012

[13] H. Xu, K. Paster, Q. Chen, H. Tang, P. Abbeel, T. Darrell, and S. Levine, "Hierarchical deep reinforcement learning agent with counter self-play on competitive games." [Online]. Available: https://openreview.net/forum?id=HJz6QhR9YQ

[14] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," vol. 521, no. 7553, pp. 436–444. [Online]. Available: https://www.nature.com/articles/nature14539

[15] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., pp. 1097–1105. [Online]. Available: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[16] L. Dormehl. What is an artificial neural network? here's everything you need to know | digital trends. [Online]. Available: https://www.digitaltrends.com/cool-tech/what-is-an-artificial-neural-network/

[17] S. Tiwari. Most intuitive explanation of activation functions - good audience. [Online]. Available: https://blog.goodaudience.com/most-intuitive-explanation-of-activation-functions-fdf6d9b4a53a

[18] A. Scherer, *Neuronale Netze: Grundlagen und Anwendungen*. Springer-Verlag, google-Books-ID: 3irVBgAAQBAJ.

[19] Desmos | graphing calculator. [Online]. Available: https://www.desmos.com/calculator

[20] B. Karlik and A. V. Olgac, "Performance analysis of various activation functions in generalized MLP architectures of neural networks," vol. 1, no. 4, pp. 111–122.

[21] R. Hecht-nielsen, "III.3 - theory of the backpropagation neural network**based on "nonindent" by robert hecht-nielsen, which appeared in proceedings of the international joint conference on neural networks 1, 593–611, june 1989. © 1989 IEEE." in *Neural Networks for Perception*, H. Wechsler, Ed. Academic Press, pp. 65–93. [Online]. Available: http://www.sciencedirect.com/science/article/pii/B9780127412528500108

[22] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization." [Online]. Available: http://arxiv.org/abs/1412.6980

[23] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey." [Online]. Available: http://arxiv.org/abs/cs/9605103

[24] D. E. Knuth and R. W. Moore, "An analysis of alpha-beta pruning," vol. 6, no. 4, pp. 293–326. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0004370275900193

[25] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of monte carlo tree search methods," vol. 4, no. 1, pp. 1–43, conference Name: IEEE Transactions on Computational Intelligence and AI in Games.

[26] G. Marcus, "Innateness, AlphaZero, and artificial intelligence," vol. abs/1801.05667. [Online]. Available: http://arxiv.org/abs/1801.05667

[27] L. Kocsis and C. Szepesvári, "Bandit based monte-carlo planning," in *Machine Learning: ECML 2006*, ser. Lecture Notes in Computer Science, J. Fürnkranz, T. Scheffer, and M. Spiliopoulou, Eds. Springer, pp. 282–293.

[28] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, pp. 265–283. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

[29] A. Ricciardi and P. Thill, "Adaptive AI for fighting games."

[30] T. D. of Computing Science {and} Maths at the University of Stirling. Rock paper scissors machine learning. [Online]. Available: http://www.cs.stir.ac.uk/~kms/schools/rps/index.php

# Appendix A

# Program Code / Resources

The source code and the documentation are available at:

https://github.com/ThraexDev/masterarbeit

There is also a copy on a CD attached to the end of this thesis.

## Ehrenwörtliche Erklärung

Ich versichere, dass ich die beiliegende Master-/Bachelorarbeit ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen. Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Mannheim, den 31.05.2020 Peter Truckenbrod