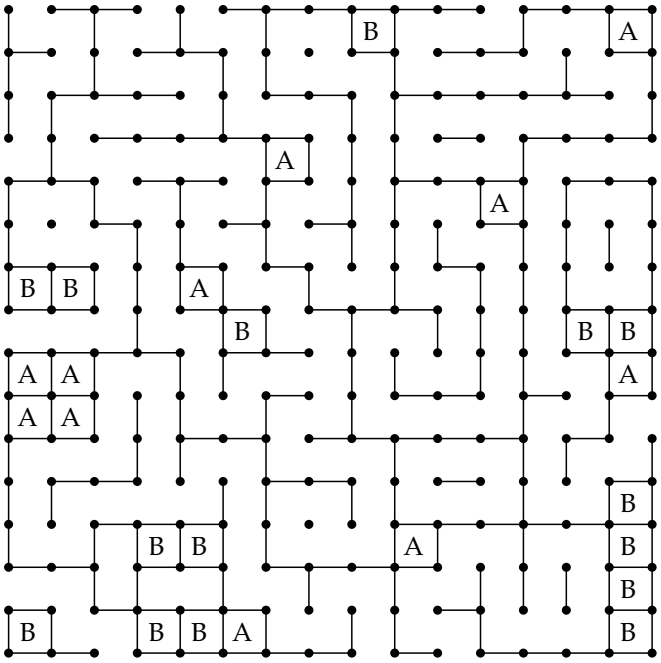
Universität Bremen

Master's thesis

Mastering the game of Dots and Boxes with deep neural networks and tree search



Author: Tom Vincent Peters

Date: November 11, 2018

Advisors: Prof. Dr. Rolf Drechsler
Dr. Hans Meine

Abstract

This thesis presents a novel approach to implementing the AlphaZero algorithm that has been the driving force behind the success of AlphaGo. Instead of Go, Chess or Shogi, this incarnation plays the game Dots and Boxes, which resembles Go in terms of complexity. Motivated by the central research question of how to reach a high playing strength while using far less resources than AlphaGo, this work explores various techniques to reduce resource requirements. Reduced computational demand is reached by training a scaled down, fully convolutional neural network on very small instances of Dots and Boxes. Experiments show that the network is able to transfer learned strategies to far larger boards without any significant loss of playing strength. Further experiments prove that it is possible to find new strategies and improve playing strength by using self-play reinforcement learning. A final evaluation against other AI opponents for Dots and Boxes shows that AlphaDots does not reach state of the art performance with the prescribed amount of training.

Contents

1. Introduction

This thesis is about applying the methods employed in AlphaGo, AlphaGo Zero and AlphaZero to the game Dots and Boxes. First an overview of Dots and Boxes, machine learning and AlphaGo establishes a basic understanding of the major topics.Afterwards section 2 presents the research questions which look for ways to replicate the success of AlphaGo while using far less resources. The research questions are followed by an in-depth discussion of related work, such as machine learning techniques, advanced strategies for Dots and Boxes and the inner workings of AlphaGo and its successors. Subsequently the thesis presents the methods that were used to adapt AlphaZero to Dots and Boxes, including a novel approach to designing neural networks that can play on boards of arbitrary size. Another major part of this thesis will be the evaluation of the newly developed AI by comparing it with various existing implementations. Answers to the posed research questions conclude the thesis.

1.1. Dots and Boxes

Dots and Boxes is a simple game for two players. It is played on a grid of dots. Players alternately connect two adjacent dots with a line. When a player draws the last line of a box, it is captured by that player, who then has to draw another line. The game ends when all lines are drawn. The player who captured the most boxes wins the game. Figure 1 shows an example game.

Dots and Boxes can be played on boards of arbitrary size. Due to the many possible actions in each turn, the game offers a large search space on bigger boards. For example a 14×14 Dots and Boxes game has about the same average branching factor as the game of Go on a standard 19×19 board. Simple rules and a large search space in Dots and Boxes make for a good test bed to replicate AlphaGo's methods. In comparison to Chess or Go the game received little attention by researchers – for example there is no proof for complexity of finding an optimal strategy on a board of arbitrary size.

There are a few publications on Dots and Boxes that describe strategies for the game, the most notable being a book [Beroo] by Elwyn Berlekamp. Among other things, he describes the typical course a game takes and details strategies that might be employed in optimal play. In the beginning of the game players mostly avoid to give away boxes to their opponent. As a consequence *chains* of connected, not yet captured boxes form on the field. A chain is made of connected boxes that can be captured in one go after the chain was opened, which means that all boxes can be captured when one line is drawn inside the chain. When no more lines are available that do not give away boxes, players are forced to open chains for their opponent. Usually they count the number of boxes in the chains and give away the shortest one.

Berlekamp describes an important strategy to play Dots and Boxes which is called *Double Dealing*. It enables a player to *stay in control* of the game by not fully capturing

chains, but leaving two or four boxes for the opponent, who in turn has to open the next chain after capturing the leftover boxes. Figure 1 shows a full example game that features Double Dealing in turn 15. Section 3 provides an in-depth explanation of the game's rules and strategies, including a more advanced strategy than Double Dealing.

1.2. Machine learning

Recent increases in computational power have given rise to the successful application of machine learning techniques, especially to using neural networks in various domains. The work of Schmidhuber [Sch15] provides a thorough overview of neural network techniques. For this thesis the focus is on using artificial neural networks to play Dots and Boxes. *Artificial neural network* is an umbrella term for machine learning techniques that are loosely inspired by biological neural networks found in brains. They share the principle that there is a network built of neurons that are connected with each other. Input is fed into the network, which processes it to generate an output.

Using an artificial neural network usually works as follows: First the network is trained by *supervised learning* which adapts the network's parameters until its output matches the desired value for a given input. This is done by providing the network with example input data, which it then processes to an output. Subsequently the network's output is evaluated by comparing it to the expected output data. After successful training, the network is deployed to solve its assigned task on new, previously unseen data.

Training a network works by presenting the network with an input and then comparing its output with the expected result. The difference between the network's output and the expected result is used to adapt the network so that later passes match the expectations more closely. Later sections will explain the required principles in more detail.

Training neural networks works best when a large amount of training data is available, because the parameters of the network must be adapted to match a general pattern in the data. If there are just a few examples, there is a high probability that the network *overfits* the data and thus only works correctly for the few given examples but fails to generalize to the actual task at hand.

1.3. Search algorithms

Strong game playing engines are often based on search algorithms, which simulate many possible outcomes of the game to find the best move. Other approaches to implementing computer opponents rely on hand crafted rules that govern the behavior.

Playing a game by a predefined set of rules usually is very fast, but might lack the ability to handle complex special cases. Conversely, searching through all available actions can take a very long time, but will yield optimal results in an exhaustive search.

To improve the speed of search algorithms, it is possible to enhance the search with heuristics, which offer a fast, rule based assessment of a given situation. In the past[1], heuristics often were complex hand crafted functions that estimated the probability to win the game. Crafting these heuristics is specific to a single type of game and typically requires intricate knowledge of the game.

The AlphaGo search algorithm combines a very successful search algorithm with a somewhat automated way to craft a good heuristic by using neural networks. Training a neural network removes the need to manually write complex code to assess games. Nevertheless it is still necessary to design and train a neural network, which introduces other hardships like providing adequate training data. AlphaGo uses neural networks in combination with Monte-Carlo Tree Search – a search algorithm which yields better results than previous approaches to highly complex search spaces like the game of Go. [Gel+12]

In it's original form, Monte-Carlo Tree Search (MCTS) analyses a given situation by repeatedly executing four steps, one of which plays random moves until the game is over. The steps are called *Selection*, *Expansion*, *Simulation* and *Backpropagation*. During the first step, MCTS traverses an internal tree structure that represents the successive states of the game which it already explored until it reaches a leaf node. Selection of the traversed path is governed by a formula that balances exploring new moves with exploiting known strong moves. A node in the tree represents a board configuration, while an edge between two nodes represents the action to get from one state to the other. After a leaf node has been selected, it is expanded, which means that all valid successive states are added to the tree. Then for the simulation step, beginning with the state of the selected leaf node, the game is played to the end by random moves. Finally the score of the simulated game is propagated backwards through the initially selected path, updating each node's value with the new result. [Cha10]

Repeatedly playing games with random moves is the inspiration for the algorithm's name, because of the well known Monte-Carlo Casino in Monaco. By executing sufficiently many games, the algorithm slowly converges to an optimal result and eventually explores all possible options. After a set number of iterations, Monte-Carlo Tree Search selects the action that has been explored most often.

Instead of the original MCTS algorithm, AlphaGo employs a modified version that does not play games with random moves to get a final score to propagate through the tree, but uses a neural network to predict the result. AlphaGo's neural network does two things at once – given a board configuration as input, it provides a probability distribution over all legal moves and an assessment of the chance to win the game. The modified version of MCTS uses the network's output to guide the search towards

[1] See for example this paper from 1989 about using heuristics in alpha-beta search for chess [Sch89]

profitable moves while it also explores new moves that were not recommended by the network. As a result it finds far stronger moves than the original network. [Sil+16; Gel+12]

A key ingredient to the success of AlphaGo is its ability to improve itself. It does so by utilizing self-play reinforcement learning where it plays against itself and simultaneously trains its neural network to reflect the improved strategies found with MCTS. There are three parts of AlphaGo that operate in parallel: One module plays the game against itself with the best available neural network, another module trains neural networks with the data that was generated in self-play while a third module evaluates neural networks to determine the best one. To create a strong instance of AlphaGo, the authors let the algorithm train the neural network in self-play for three weeks. They distributed the task to 50 graphics processing units (GPU) on multiple machines. [Sil+17a]

Later sections will discuss the algorithm in more detail. There will also be further explanations of how heuristics and neural networks tie into search algorithms.

1.4. Related work

Some parts of this thesis build upon the related bachelor's thesis [Pet15] by the same author. The thesis is about applying the minimax search algorithm with alpha-beta pruning and various other enhancements to Dots and Boxes. Besides minimax, the thesis also covered a basic version of Monte-Carlo Tree Search that used the simple, rule based Easy, Medium and Hard AIs in KSquares for the rollouts. The framework for evaluating the playing strength of various implementations against each other is reused in this thesis.

QDab as described in [Zhu14] is the best currently available opponent in Dots and Boxes according to the evaluation in [Pet15]. It is built using Monte-Carlo Tree Search and a fully connected neural network which requires a smartly selected set of features extracted from a given board state to play Dots and Boxes on a 5×5 board. QDab is restricted to playing Dots and Boxes on 5×5 boards because the neural network and other parts of the implementation can not handle different board sizes. Its neural network is fed with a set of features that are extracted from the board. Among other things, the features represent 12 different types of chains. The internal representation of the game utilizes the Strings and Coins format that is described in section 3.2.

In March 2016, AlphaGo beat the world champion in Go and thus achieving a historic breakthrough that can be compared to the game of chess between Garry Kasparov and Deep Blue in 1997. Before AlphaGo, computer programs that play Go were only able to beat amateur players. The methods that were used in AlphaGo are described in the paper *Mastering the game of Go with deep neural networks and tree search*. [Sil+16] Since this thesis is about replicating the methods of AlphaGo and its direct descendants [Sil+17b; Sil+17a] for Dots and Boxes, this thesis is named after the initial paper about

AlphaGo with only the name of the game changed. As the title suggests, the main components of AlphaGo, AlphaGo Zero [Sil+17b] and AlphaZero [Sil+17a] are neural networks and Monte-Carlo Tree Search. This thesis will go into detail on the concepts that form the basis for the application of AlphaGo's methods to Dots and Boxes. A key aspect of AlphaGo is that during training the algorithm improves itself with self-play reinforcement learning.

AlphaGo Zero is a streamlined version of AlphaGo that does not rely on human training data, while AlphaZero is the general form of the algorithm that was evaluated on Chess and Shogi. Consequently, this thesis calls its implementation *AlphaDots* or *AlphaZero for Dots and Boxes*.

In 1998 Weaver et. al. tried to train and evolve a simple feed forward neural network with one input, one hidden and one output layer to play Dots and Boxes with little to no success. [WB98]

There is a noteworthy book about Dots and Boxes written by Elwyn Berlekamp. [Beroo] It covers strategies for the game and offers many examples of challenging board configurations, some of which were used as tests when implementing the AlphaDots algorithm. Besides the book about Dots and Boxes, Elwyn Berlekamp also contributed a chapter about Dots and Boxes to the book *Winning Ways for Your Mathematical Plays: Volume 3* that explains strategies for Dots and Boxes and provides mathematical insight into the game. [BCG03]

Dots and Boxes was solved for small board configurations up to 5×4 boxes by Barker et. al. in 2012. [BK12] The authors relied on symmetric properties and other features of Dots and Boxes to reduce the search space when they computed all possible moves. For a special type of endgame in Dots and Boxes the authors of *Playing simple loony Dots and Boxes endgames optimally* provide an optimal strategy. [BC14]

2. Research questions

The overarching goal of this thesis is to research an opponent for Dots and Boxes that is based on Monte-Carlo Tree Search combined with deep neural networks as described in the papers about AlphaGo, AlphaGo Zero and AlphaZero. [Sil+16; Sil+17b; Sil+17a] This thesis' research questions are aligned along that goal. Although this work seeks to apply the methods of AlphaZero, its emphasis is to keep the hardware requirements to an affordable minimum.

Beginning with the implementation of different neural networks, the thesis will try to replicate a policy-value network that is able to play Dots and Boxes. Before designing and training neural networks, it is necessary to provide suitable training data on which the networks can operate. Training a neural network should not take longer than three days on a modern GPU so that it is possible to replicate the results within an acceptable timeframe. After training, the network needs to be easily accessible within a usable interface.

After implementing a neural network that can play Dots and Boxes, it has to be tied into an implementation of Monte-Carlo Tree Search to create an integral part of the AlphaZero algorithm. By using Monte-Carlo Tree Search (MCTS) that utilizes the output of a neural network to find better moves than the network itself, it should be possible to create better training data for the neural network. Further training on the improved data should result in a network that makes better moves. Consequently it should then be possible to implement a training loop where the network steadily improves by self-play, thus constituting a working AlphaZero algorithm for Dots and Boxes.

The following research questions are to provide the major topics of this master's thesis.

1. How can the results of AlphaGo Zero be replicated for Dots and Boxes with far less resources?
2. How well does a convolutional neural network lend itself to playing Dots and Boxes on varying board sizes?
3. Is it possible to find new strategies (e.g. Double Dealing) by training a neural network with reinforcement learning in self-play?
4. What is the playing strength of the newly implemented AIs?

2.1. Is is possible to implement AlphaZero with less resources?

Can good results be reached with less resources than those of AlphaGo?

According to the publications on AlphaGo, AlphaGo Zero and AlphaZero, the authors used a humongous amount of computational resources to train their neural networks. The policy network for AlphaGo was first trained on 29.4 million positions from 160,000 games on 50 GPUs for three weeks. Afterwards the policy network was trained by self-play reinforcement learning using 50 GPUs for a week. [Sil+16] For AlphaGo Zero there were two versions: one "small" version with 20 blocks that was trained on 64 GPUs for 3 days and a large version with 40 blocks that was trained on 64 GPUs for 40 days. [Sil+17b] The authors of the AlphaZero paper state that they used 64 TPUs[2] [Sil+17a]

For this thesis there are far less resources available. Consequently, the central research question is if comparable results can be reached with just one GPU and a maximum of three days for training. Furthermore using less resources will benefit the reproducibility of the presented results. Reducing the amount of required resources will guide the design decisions and direction of research of this thesis.

2.2. Is it possible for the same network to play on varying board sizes?

How well does a convolutional neural network lend itself to playing Dots and Boxes on varying board sizes?

Humans play Dots and Boxes on board of many different sizes and shapes. It is common to mark an arbitrary area on quad paper and use it as game board. Consequently an algorithm that plays Dots and Boxes should ideally be able to play on arbitrary boards. If a neural network could play on boards of various size, would it be able to transfer the strategies it learned on one particular type of board to another one?

To answer the research question, a neural network must be devised and trained to play Dots and Boxes on different board sizes. The network should be able to play the game on differently sized boards without adapting the weights for the given environment. If the network is able to play well on unknown board sizes, it probably has found some strategies that apply to Dots and Boxes in general. It should at least be able to avoid giving away boxes prematurely and furthermore it should ideally open short chains first before it opens long chains. Section 3.3 defines the different types of chains in Dots and Boxes.

Before any training can commence, two things are required first:

- A network architecture has to be defined that is able to play on various boards.

[2] TPU is short for Tensor Processing Unit – a specialized piece of hardware for operating on neural networks. [Jou+17]

- There has to be training data on which the network can be trained.

The design of the network's architecture and the composition of the training data are key issues for this research question. Besides these design decisions, the evaluation results will provide answers to the posed question.

2.3. Do new strategies emerge from self-play?

Is it possible to find new strategies (e.g. Double Dealing) by training a neural network with reinforcement learning in self-play?

To answer this question, a self-play reinforcement learning method must be implemented for Dots and Boxes. It should be an algorithm like AlphaGo that is able to improve its playing strength by competing against itself. Ideally it should not be as resource intensive as AlphaGo so that it is possible to reproduce the results with just a single GPU in an acceptable time.

In Berlekamp's chapter on Dots and Boxes in the book Winning Ways Volume 3, he describes a list of insights into Dots and Boxes that human players can gather. His first example is "you don't just open up any boxes unless you have to and then you open as few as possible." [BCG03, p. 569] This basic level of proficiency in Dots and Boxes could be used as a starting point from which new strategies can be learned in self-play under the assumption that it is possible to train a neural network to play Dots and Boxes in the first place.

Besides a neural network that can play Dots and Boxes at a basic level, there needs to be a policy improvement operator. [SB+98] A policy defines the behaviour of an agent (here: neural network or AI) for any given situation. Accordingly a policy improvement operator is a mechanism that improves the policy in terms of playing strength. So for this case there needs to be an algorithm that improves the output of a neural network. For this the authors of AlphaGo use a variant of Monte-Carlo Tree Search (MCTS) as their policy improvement operator. The employed version of MCTS utilizes "contextual side information" provided by the neural network to find even better moves – the network guides MCTS by providing it with an assessment of the given board position. [Ros11] As this thesis orients itself to the methods of AlphaGo, a similar approach might be viable to create an algorithm that is able to improve its own play and find new strategies.

If the algorithm is able to consistently apply Double Dealing after starting with a network that has not seen a single instance of Double Dealing, it could be considered as a success. To be sure that the reason for the consistent application of Double Dealing lies in the improvement brought by self-play reinforcement learning, the algorithm should be able to use Double Dealing only after training in self-play.

2.4. How well do the new AIs play?

What is the playing strength of the newly implemented AIs?

After investigating various aspects of the AlphaZero algorithm, it would be interesting to compare its performance with the state of the art. As of writing this thesis, the best known AI for Dots and Boxes is QDab according to the evaluation in [Pet15]. Like AlphaZero, QDab is based on Monte-Carlo Tree Search in combination with a neural network to search for the best move. In contrast to the deep convolutional neural network of AlphaZero, the neural network of QDab only has three layers and is fed with hand-crafted features. QDab is restricted to playing on boards of 5×5 boxes. Its playing strength exceeds all other approaches that were investigated in [Pet15].

To answer the question of how the newly implemented AIs compare to the state of the art that is represented by QDab, they play games against each other. There already exists a framework for comparing various Dots and Boxes AIs which was built for the bachelor's thesis [Pet15] of the same author as this thesis. The framework will be extended and used to compare the various neural networks and Monte-Carlo Tree Search algorithms with other AIs.

3. Dots and Boxes

Dots and Boxes is a well known pen and paper game for two players. Despite its simple rules, Dots and Boxes offers a challengingly high complexity on larger boards. This chapter first describes the game's rules and then goes on to describe further aspects that are important for understanding and playing the game reasonably well. Finally there is a short probe into the theoretical properties of Dots and Boxes, including a take on its complexity.

3.1. Rules

The game is played on an $m \times n$ grid of dots where the players alternately draw a horizontal or vertical line between two adjacent dots. Gradually squares are formed, which are the so-called boxes. By drawing the fourth side of a box, a player completes and captures it. If a player completes two boxes with one line, still only one more line must be drawn. When there are no lines left to draw, the game ends. The player with the most captured boxes wins the game. Depending on the size of the board, there can be an even number of boxes so that a game may end in a draw.

Figure 1 presents a full game, played from start to finish by the players A and B on a 3 $\times$ 3 board. The first and third column show the turns made by the first player, while the second and fourth column represent the other player's moves. Turns in the game are numbered from left to right and from top to bottom. During the first 13 moves only one line is added in each turn. Beginning with the 14^{th} turn, the players draw more than one line per turn, because they captured boxes. When a player added more than one line in a turn, the order of added lines is denoted by small numbers next to the newly drawn lines. At the end player A wins the game with 5 captured boxes versus 4 boxes for player B.

Arising from the simple rules of the game, there are a few further aspects to be understood when talking about Dots and Boxes. Consequently, the next three sections will discuss another board representation, chains and phases in Dots and Boxes.

3.2. Strings and Coins

There is a dual, more general representation of every Dots and Boxes game called Strings and Coins. Boxes are represented by coins, which are initially held by four strings. Drawing a line in Dots and Boxes corresponds to cutting a string in Strings and Coins. When all strings of a coin are cut, the player captures the coin. Figure 2 shows a Dots and Boxes game side by side with its Strings and Coins representation.

Strings and Coins might offer a more accessible understanding of chains in Dots and Boxes, which will be explained in the next section.

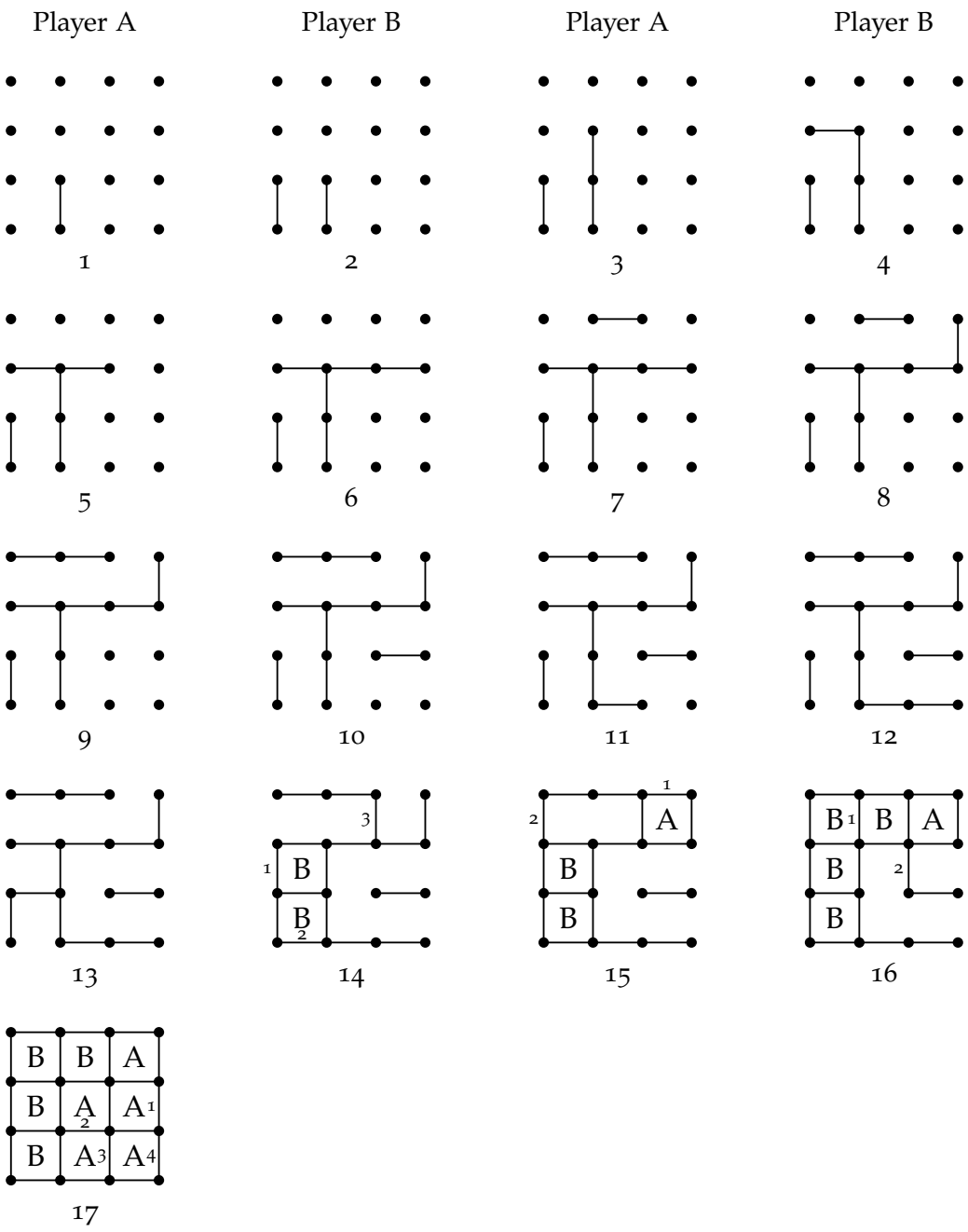


Figure 1: A game of Dots and Boxes, as shown in [Pet15] and based on figure 2 on page 4 in [Beroo]

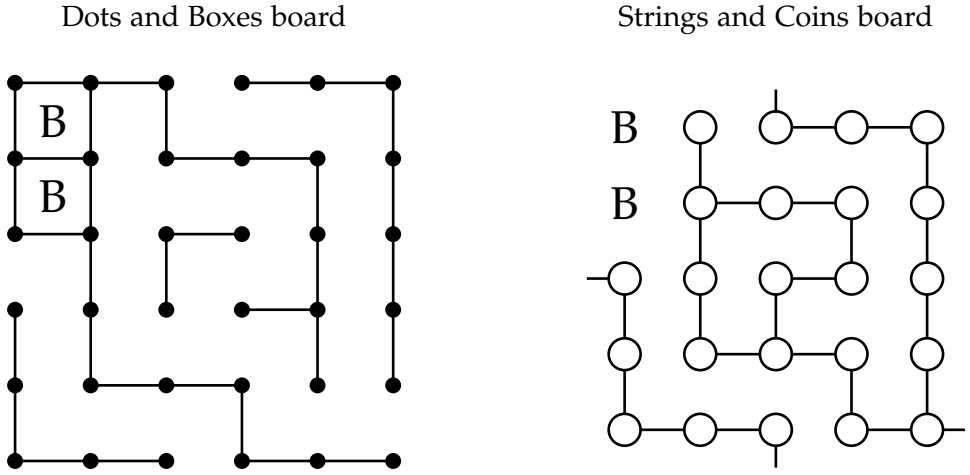


Figure 2: The same game displayed on a normal Dots and Boxes board and on a Strings and Coins board

3.3. Chains

One key aspect of Dots and Boxes are chains: They form during the game and are made of connected boxes which are close to capture. If a line is drawn inside a chain, no box is captured by the current player, but the next player can capture all boxes of the chain.

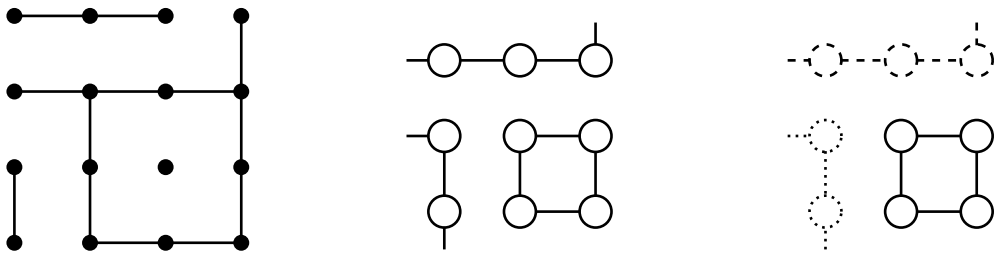


Figure 3: Three types of chains, shown on a Dots and Boxes board, on a Strings and Coins board and on a Strings and Coins board with marked chains

Figure 3 provides an example of different chains. The figure shows the same game three times, but in different representations. The left representation uses the typical Dots and Boxes board, where it is not always easy to discover chains with the naked eye. With the Strings and Coins board that is used for the other two representations, chains are separate entities.

Chains can be categorized in three distinct types:

Short Chain A Short Chain consists of one or two coins / boxes. They bear a strategic difference to long chains, because it is possible to prevent Double Dealing. Figure 3 shows a Short Chain that is marked by dotted lines.

Long Chain A Long Chain is made of at least three coins / boxes. Opening a long chain enables the opponent to take control of the game by Double Dealing. Figure 3 shows a Long Chain that is marked by dashed lines.

Cyclic Chain A Cyclyc Chain is made of an even number of coins / boxes that form a cycle. In terms of control, a Cyclyc Chain behaves like a Long Chain with the distinction that Double Dealing costs twice as many coins / boxes. Figure 3 shows a Cyclic Chain that is marked by solid lines.

3.4. Phases

During a game of Dots and Boxes, there is a point where all moves that are left will give away one or more boxes to the opponent. This event can be seen as the dividing point between two main phases of the game. In the first phase, the players mostly avoid to give away boxes and decide about the shape the game will take. Then, in the second phase, it is no longer possible to avoid giving away boxes and the players usually open short chains first to give away as few boxes as possible.

Since the chains are formed during the first phase, it could be called the forming phase. Correspondingly the second phase might be called endgame phase, because the competitors get their scores. In his book on page 81, Elwyn Berlekamp describes certain typical milestones of a Dots and Boxes game. [Beroo]

3.4.1. The forming phase

Beginning with an empty board, the players make their moves, with which they slowly create chains. It is during this phase of forming chains that the winner of the game is decided. Since the shape of the board will be the base on which the endgame will happen, forming that shape decides who will win.

Due to the fact that there are many possible moves that each player can make, the game has a high branching factor in the forming phase. As a result, the search space is rather large, especially on larger boards.

During the forming phase the winner of the game is decided when both opponents play according to an optimal strategy. In the endgame phase, the final score of the game is usually very clear. To get the upper hand an optimal player might sacrifice a box so that the remaining moves enables the player to later take control of the game.

3.4.2. The endgame phase

When there are no more save lines left to draw, one player will gain control of the game by applying the Double Dealing strategy, which is explained in section 3.5.1. The player who controls the game, will usually win the game, if the board is sufficiently large and there are sufficiently many long chains.

3.5. Strategies for Dots and Boxes

For playing an optimal strategy in Dots and Boxes as a human, it can be beneficial to understand the concepts of Double Dealing and Preemptive Sacrifices. Both strategies are explained in Elwyn Berlekamp's book on Dots and Boxes. [Beroo]

3.5.1. Double Dealing

With Double Dealing a player declines to capture the last few boxes of a chain to force the opponent to open another chain. Double Dealing usually happens in the endgame phase of Dots and Boxes when no save lines are left. A player who can capture a long or cyclic chain is able to do Double Dealing by declining the last two or four boxes respectively. [Beroo]

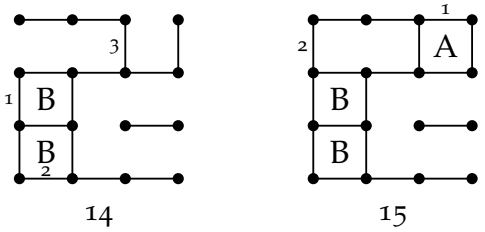


Figure 4: Move 14 and 15 from figure 1

Figure 4 shows an example of Double Dealing. The player B in move 14 captures two boxes and is then forced to open a long chain. In move 15 the opponent A captures only one box and then leaves two boxes for the enemy, who has no other option than taking the two boxes and then opening the other long chain. As a result of Double Dealing, player A is able to win the game with a score of five to four. Without applying Double Dealing, player A would have had to open the last long chain, thus leaving it for capture by player B who would then win the game six to three.

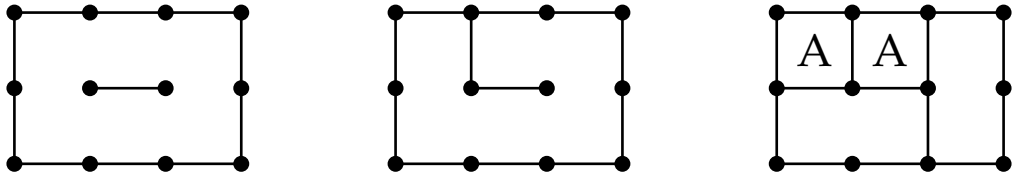


Figure 5: Double Dealing in cyclic chains

Figure 5 shows an example in three steps where Double Dealing is applied in a cyclic chain of six boxes. In the beginning a simple cyclic chain with six boxes is shown. Then the cyclic chain is opened. The last part shows two captured boxes and the Double Dealing move which leaves four boxes to the opponent. Depending on the overall structure of the Dots and Boxes board it can be better to not apply Double Dealing because the opponent will score too many boxes. [Beroo]

3.5.2. Preemptive Sacrifices

During the forming phase of Dots and Boxes, the players decide what structure the board will have – they decide how many chains of what type there will be. Arriving at a beneficial final structure before entering the endgame phase is crucial for each player. Usually it is especially important to influence the game so that the player can take control of the game in the endgame phase with Double Dealing. To achieve that goal a player might need to sacrifice a box instead of drawing a save line that does not give away any boxes. An example for such a situation is shown in the first board of figure 6 that is taken from Elwyn Berlekamp's book on Dots and Boxes. [Beroo]

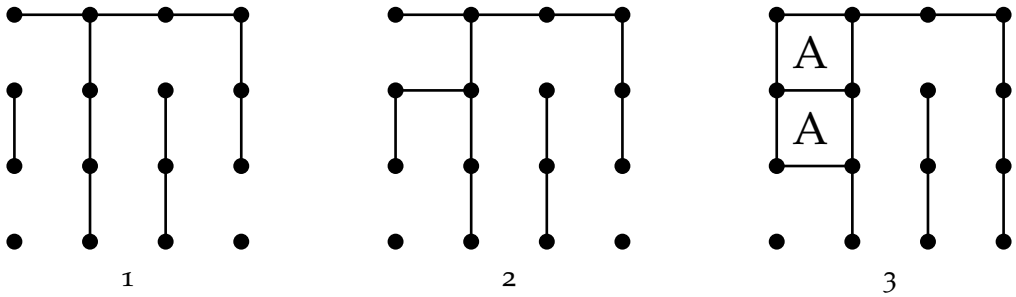


Figure 6: Problem 3 from Elwin Berlekamp's book on Dots and Boxes [Beroo] with the correct solution to make a preemptive sacrifice

In the first board in figure 6 the only option to win the game is opening the short chain

with a *hard hearted handout*[3] as seen in the second part of figure 6 and thus giving away two boxes to the opponent. When the current player preemptively sacrifices the two boxes, the opponent will then have to capture the two boxes and draw the last save line (or open the single box, leading to the same result). Thanks to the sacrifice, the initial player is then faced with a situation where the board has a short chain and a long chain. That situation is desirable because the initial player opens the single box for the opponent, who then has to capture it and open the long chain.

3.6. Game theoretical classification

Dots and Boxes is an impartial game. This means that moves are not specific to players – all players are allowed to make all available moves. Chess for example is not impartial because players own pieces on the board and are not allowed to move the opponent's pieces.

Like Go, Dots and Boxes offers a high average branching factor. A 19×19 Go game has an average branching factor of 250, while a 15×15 Dots and Boxes game has 240.5 as average branching factor.

On an empty board of $m \times n$ boxes there are $2 \cdot m \cdot n + m + n$ lines which can be drawn. [Beroo, p. 8] With each move, this number is reduced by one. It follows that the average branching factor $\mathcal{B}$ for a whole Dots and Boxes game with L lines is $(L+1)/2$. The following equations show how the branching factor for Dots and Boxes was derived. Beginning with L lines, there is one less line with each move. Thus the total number of actions that are available in a game of Dots and Boxes is $\sum_{i=1}^{L} i$. The average branching factor is calculated by dividing the total number of available actions by the number of moves L. By replacing the sum with its corresponding formula, it is easy to further reduce the formula of the average branching factor.

$$\mathcal{B} = \frac{1}{L} \sum_{i=1}^{L} i \tag{1}$$

$$= \frac{1}{L} \cdot \frac{L \cdot (L+1)}{2} \tag{2}$$

$$= \frac{L \cdot (L+1)}{2 \cdot L} \tag{3}$$

$$= \frac{L+1}{2} \tag{4}$$

[3]The term *hard hearted handout* is described in Winning Ways Volume 3 [BCG03] on page 547. It simply cuts a short chain in half so that Double Dealing is not possible. Conversely a *half hearted handout* is when a short chain is opened on one of its ends, so that the opponent can apply Double Dealing by drawing a line at the other end of the chain.

To provide an overview of the complexity of various board sizes in Dots and Boxes, table 1 presents the number of lines and the average branching factor for increasingly larger quadratic Dots and Boxes boards.

Board size	Lines	Average branching factor
3×3	24	12.5
4×4	40	20.5
5×5	60	30.5
6×6	84	42.5
7×7	112	56.5
8×8	144	72.5
9×9	180	90.5
10×10	220	110.5
11×11	264	132.5
12×12	312	156.5
13×13	364	182.5
14×14	420	210.5
15×15	480	240.5

Table 1: Complexity of various quadratic Dots and Boxes boards

4. Machine learning methods

This chapter will focus on some of the fundamental concepts that are used in AlphaGo and AlphaZero to learn to play the game. Since the "learning" part is facilitated with neural networks, this chapter is about the basic principles that were used to implement neural networks. The techniques that are described here were not implemented for this thesis because they are widely available in existing frameworks. [Cho+15; Mar+15; Olio6]

Neural networks are organized in layers. Layers contain weights that control the network's behaviour. Weights are also-called parameters and are adapted during training to make the network "learn" desired behaviour. In a so-called *forward pass* a network receives input data, which is then passed through and processed by the network's layers to produce an output. When training a neural network, there is example data that is comprised of input data and expected output data. The example input data is fed into the neural network in a forward pass to produce output data. A loss function evaluates the difference between the network's output and the expected output, which is then used to change the weights in the network's layer to minimize the difference (loss). Minimizing the loss is done by adapting the weights with a technique called *backpropagation* where the loss is propagated backwards through the network's layers. [LBH15, p. 438]

4.1. Stochastic gradient descent

Stochastic gradient descent (SGD) is a method to train neural networks. It works by repeatedly applying gradient descent to a small set of randomly selected examples. Gradient descent is an optimization algorithm that can be used to minimize the loss function of a neural network. Any optimization algorithm seeks to minimize a function that usually has multiple parameters. It works by partially differentiating the loss function with respect to each weight and then using the gradients to adapt the weights. A function's gradient at a certain point x represents the function's slope at point x. When the goal is to minimize a function, its gradient provides the direction of the nearest local minimum. As a consequence of using gradient descent, all parts of a neural network that contribute to its output have to be differentiable, because all parts of the neural network contribute to the result of the final loss function.

4.2. Types of neural network layers

Most neural network architectures work with layers that usually represent different levels of abstractions. [IWK17] AlphaGo's neural network architecture makes use of two types of layers. Mostly, it is built with convolutional layers – only the last few layers are fully connected. [Sil+17b] Since fully connected layers were the first type

of layer to be widely used, they will be described prior to convolutional layers. Fully connected neural networks date back to about 1958, when Frank Rosenblatt introduced the first "perceptron", which resembles the design of a neuron that is described in this section. [WR17, p. 3, 9]

4.2.1. Fully connected neural networks

Fully connected neural networks are built with neurons that are arranged in layers. Neurons are also-called perceptrons or units. One neuron is depicted[4] in figure 7. A neuron has a fixed number of n inputs $x_1 \dots x_n$. Each input is multiplied with its associated weight $w_1 \dots w_n$ and then summed up. The weights represent the trainable parameters of the neuron. Optionally, a neuron can have a bias b that acts as additional input that is fixed to the value one and also added to the sum of weighted inputs. Finally an activation function f is applied to the sum.

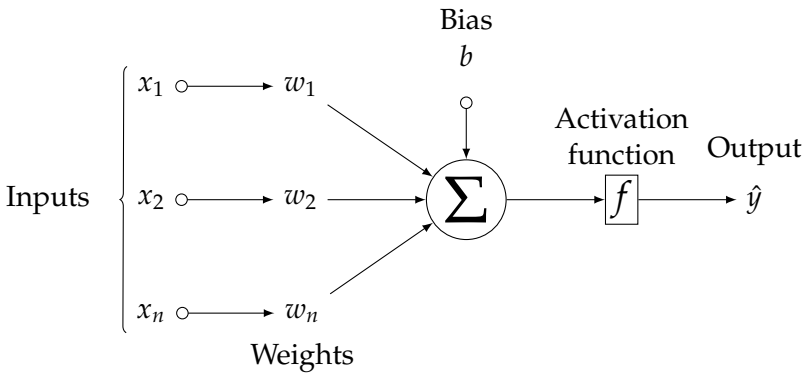


Figure 7: A single neuron in a fully connected layer of a neural network

The function for the output $\hat{y}$ of a single neuron can be written as follows:

$$\hat{y} = f\left(b + \sum_{i=1}^{n} x_i w_i\right)$$

In practical implementations, a neural network's forward pass is often realized with matrix operations where multiple examples are processed batchwise. This means that multiple examples are grouped together in a batch that is processed as a single chunk of data. Handling data in batches often increases efficiency and also has other advantages that are described in the sections about stochastic gradient descent and batch normalization.

A layer of neurons requires a fixed number of inputs, because for each input there needs to be a weight. As a consequence of its design, it is not possible to change

[4]Figure 7 is taken from `http://tex.stackexchange.com/questions/132444`

the shape of a fully connected layer's input after training, because that would add or remove weights which are subject to training. It is possible to combine a layer of fully connected neurons with other types of layers – for example with convolutional layers – as long as those layers produce an output of fixed size when prepended to a fully connected layer. The shape of the output of a fully connected layer is determined by the number of units in the layer. For each node in a fully connected layer there is one output. Nodes can be arranged in two or more dimensions to reflect the shape of the processed data.

The following example illustrates how a forward pass is realized for a fully connected layer with matrix multiplication. Assume that there are two neurons N_1 and N_2 in one layer and that the input to the layer has three values. As a result, each neuron has three weights – one for each value of the input. Then there are six weights: neuron N_1 has $w_{N_1.1}$, $w_{N_1.2}$ and $w_{N_1.3}$ while neuron N_2 has the weights $w_{N_2.1}$, $w_{N_2.2}$ and $w_{N_2.3}$. When we want to process a batch of two samples A and B, there are six values that go into the layer – three values for each sample: A_1, A_2, A_3, B_1, B_2 and B_3. Furthermore we assume that there is an activation function f and no bias. Calculating the outputs $\hat{y}_{A.N_1}$ and $\hat{y}_{A.N_2}$ for sample A and $\hat{y}_{B.N_1}$ and $\hat{y}_{B.N_2}$ for sample B can be done with matrix multiplication:

$$
\begin{aligned}
& f\left(\begin{bmatrix} A_1 & A_2 & A_3 \\ B_1 & B_2 & B_3 \end{bmatrix} \begin{bmatrix} w_{N_1.1} & w_{N_2.1} \\ w_{N_1.2} & w_{N_2.2} \\ w_{N_1.3} & w_{N_2.3} \end{bmatrix}\right) \\
=& f\left(\begin{bmatrix} A_1 \cdot w_{N_1.1} + A_2 \cdot w_{N_1.2} + A_3 \cdot w_{N_1.3} & A_1 \cdot w_{N_2.1} + A_2 \cdot w_{N_2.2} + A_3 \cdot w_{N_2.3} \\ B_1 \cdot w_{N_1.1} + B_2 \cdot w_{N_1.2} + B_3 \cdot w_{N_1.3} & B_1 \cdot w_{N_2.1} + B_2 \cdot w_{N_2.2} + B_3 \cdot w_{N_2.3} \end{bmatrix}\right) \\
=& \begin{bmatrix} f\left(\sum_{i=1}^{3} A_i \cdot w_{N_1.i}\right) & f\left(\sum_{i=1}^{3} A_i \cdot w_{N_2.i}\right) \\ f\left(\sum_{i=1}^{3} B_i \cdot w_{N_1.i}\right) & f\left(\sum_{i=1}^{3} B_i \cdot w_{N_2.i}\right) \end{bmatrix} \\
=& \begin{bmatrix} \hat{y}_{A.N_1} & \hat{y}_{A.N_2} \\ \hat{y}_{B.N_1} & \hat{y}_{B.N_2} \end{bmatrix}
\end{aligned}
$$

4.2.2. Convolutional neural networks

Convolutional neural networks are useful for classifying images and performing visual tasks in general. Their properties resemble some biological aspects of the visual cortex in animals and humans, including a hierarchical organization and the ability to preserve locality. In 1962 David Hubel and Torsten Wiesel published their research [HW62] about the receptive field in the visual cortex of cats where they described neural responses to different visual stimuli of movement.

Convolutional layers consist of *kernels* that are applied to the layer's n-dimensional

input. A kernel, which is also-called *filter*, is an n-dimensional matrix of real numbers. In a neural network, the filters of a convolutional layer represent a form of shared weights that are adapted during training to fit the expected output. The operation of applying a kernel to data is called convolution. A kernel is convolved with input data by moving the kernel across the data and applying it in every position. [LBH15]

Before the recent application in deep neural networks, the principle of convolving an image with a kernel has been used in machine vision tasks. For example there is the Sobel operator, which was first described in 1973 by Duda et. al. and is used for edge-detection. The Sobel operator works by convolving two 3×3 kernels with an image to extract vertical and horizontal edges. [DH73] Edge-detection with the Sobel operator works with the following two kernels A and B:

$$A = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} \qquad B = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

Figure 8 shows an example where a 7×5 image I is overlaid by a 3×3 kernel K which is used to calculate the value of one pixel of the output image. Each pixel I_j of the input image that is currently overlaid by the kernel is multiplied with the corresponding value K_i in the kernel. By summing up the results of all multiplications, the value of one output pixel is calculated.

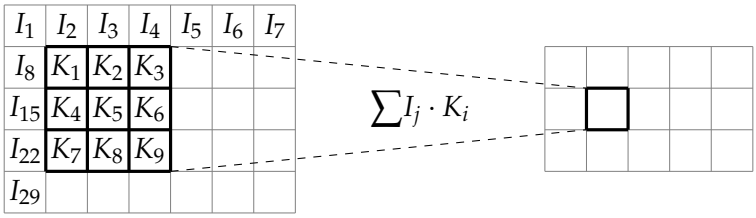


Figure 8: Illustration of the convolutional operation

The process in figure 8 is repeated for all possible overlay positions of the kernel, while keeping the kernel fully inside the image, resulting in a 5×3 output image. The resulting output of a convolution is also-called feature map, because convolving a kernel with an image can be interpreted as extracting a feature from the original image. In some cases it is useful to add a padding of zeroes around the original image so that the result has the same size as the input image. Furthermore the kernel may be moved with a higher stride so that it moves two or more pixels at each step. Using a higher stride creates smaller output images.

Usually, a convolutional layer in a neural network has more than one kernel and each kernel is applied to the input data. As a result of multiple filters, a layer might produce output data that has a larger shape than its input data. Considering for example input data in the shape of $N \times M \times 3$ that is processed by a convolutional

layer with ten filters: to match their input data, the ten kernels must have a size of $A \times B \times 3$. Since there are ten filters, the convolutional layer will produce an output shape of $N \times M \times 10$ if it uses padding. If the layer would not use padding, the first two dimensions of the output would be slightly smaller, depending on the size of A and B. When a convolutional layer processes a typical RGB image, the input data is three-dimensional. The three channels for red, green and blue form a third dimension, leading to three-dimensional kernels in the convolutional layer that receives the image as input.

A key aspect of convolutional neural networks is their ability to preserve local spatial relations. Since the kernels are moved across the input data, they are sensitive to local configurations of data. By stacking many convolutional layers on top of each other, they are able to gradually extract more complex features – the lowest layers detect various kinds edges, which are then used by later layers which detect shapes like for example human eyes. This constitutes a hierarchical approach to image processing, and has been shown to deliver good results in image classification. [KSH12]

4.3. Activation functions

Activation functions are commonly applied after each layer of a neural network. This section describes all activation functions that are used for this thesis. There are different types of activation functions that are useful for achieving different goals. Internal parts of the neural networks use the ReLU function after each layer, since they have achieved faster training time in large networks as shown in the paper about ImageNet, which is a successful image classification network. [KSH12] For the final output of neural networks used by AlphaDots, there are two activation functions – softmax and tanh. Softmax produces a vector that can easily be interpreted as a probability distribution while tanh yields a single value from -1 to 1 which is used to assess the chance to win a game.

4.3.1. Rectified linear activation

The rectified linear (ReLU) activation function is defined as follows:

$$f(x) = \max(0, x)$$

Figure 9 shows a plot of the ReLU activation function.

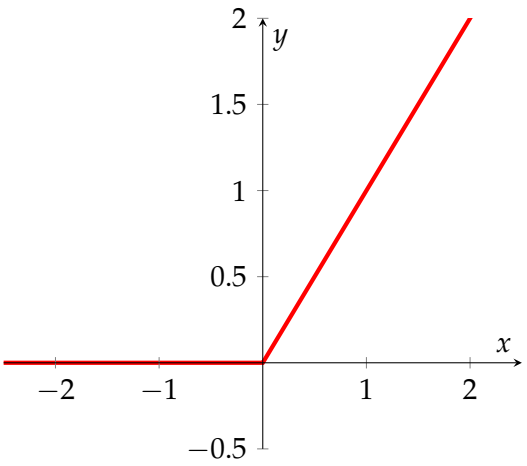


Figure 9: The rectified linear activation function

The ReLU activation function is widely used in deep neural networks. [LBH15] It was introduced for deep learning by Glorot et. al. in *Deep sparse rectifier neural networks*. [GBB11] In the paper about ImageNet classification with deep neural networks by Krizhevsky et. al., the ReLU activation function is credited with improving training times and enabling larger networks. [KSH12]

4.3.2. Softmax activation

The Softmax activation function provides a discrete probability distribution that is useful to assign probabilities to categories. Its output is a vector with k real-valued entries that add up to 1. As a result, each entry in the vector can be interpreted as the probability associated with the category that is represented by that entry. Given an input vector x of k arbitrary scores, the Softmax activation function $S(x)$ can be defined as follows:

$$S_j(x) = \frac{\exp(x_j)}{\sum_{i=1}^{k} \exp(x_i)}$$

The Softmax activation function is described by Sutton et. al. for action selection in the context of Reinforcement Learning in chapter 2.3 in [SB+98].

4.3.3. Tanh activation

The hyperbolic tangent function takes a real-valued input and produces a real-valued output in the range from -1 to 1. It is defined by using the exponential function as follows:

$$\tanh(x) = \frac{1 - \exp(-2x)}{1 + \exp(-2x)}$$

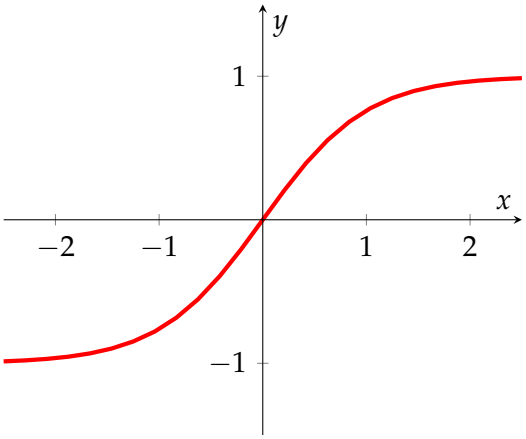


Figure 10: The hyperbolic tangent activation function

4.4. Regularization and normalization

When data is passed through a neural network it is subject to many transformations. Successful approaches usually employ some kind of normalization multiple times in a neural network – often the data is normalized after every layer in the network. Additionally there are cases where is has proven useful to keep the parameters or weights of a neural network close to zero if possible. This section describes one method for normalization and one method for regularization that both affect particular aspects of the neural network and the processed data in useful ways. One method is concerned with ensuring that the data adheres to a certain distribution while the other method tries to minimize the weights of the neural network. There are more notable methods of normalization, that are not mentioned in this thesis, because they were not used by AlphaGo or AlphaZero for Dots and Boxes.

4.4.1. Batch normalization

Actual implementations of neural networks usually process more than one sample at once. One reason is that processing data in bigger chunks is faster, because it reduces the overhead of reading and writing to slow storage for each sample. When a GPU is utilized to train neural networks, the data has to be transferred to the GPU's memory, which is a comparably slow process. Consequently, it is advisable to load many samples at once.

A batch is a small set of samples – typical batch sizes are 8, 16, 32 and even bigger multiples of two. Another reason to process data in batches is that Ioffe et. al. found that a neural network can be trained far more efficiently when the data is normalized batchwise. This means that after each layer, the data is normalized so that the batch's mean is close to 0 and the batch's standard deviation is close to 1. To better understand the benefit of batch normalization let us assume that a layer close to the network's input changes its weights. As a consequence the distribution of layer's output changes and forces subsequent layers to adapt to that change, thus forcing them to also update their weights. By using batch normalization, the effect of the layer's changed output data distribution does not affect subsequent layers, because the output is normalized before it is passed on to the next layer. [IS15]

4.4.2. L_2 regularization

When data is processed by a neural network, it has to be stored in suitable containers like arrays of integer or floating point values. Due to the inherent design properties of these data containers, it is not possible to store arbitrary values. There are constraints in terms of the minimum and maximum value and the precision a certain type can handle. To create a well working neural network, it is important to make sure that it is *numerically stable*, which means that the data processed by the network stays well within the constraints of the employed data types.

Training artificial neural networks tries to minimize the loss function by adapting the network's weights. Depending on the design of the loss function, it is possible that there are multiple optimal weight configurations from a purely mathematical viewpoint. Since the networks are run by a computer, it is important to select a solution that stays within the system's constraints by adding a penalty for using large weights. One way to define such a penalty is L_2 regularization. It is defined as follows for k weights w of a neural network:

$$L_2 = \lambda \sum_{i=1}^{k} w_i^2$$

The L_2 regularization is added to the loss function that is used by a neural network. L_2 regularization works by summing up the squared weights of the network and multiplying it with a constant factor λ, which is set to 10^{-4} for AlphaDots' neural network. According to Schmidhuber's overview of Deep Learning in Neural Networks [Sch15] the general principle of trying to keep the network's weights close to zero is also-called *weight decay* amongst other terms. The paper *A simple weight decay can improve generalization* [KH92] from 1992 describes regularization of the weights in a feed forward neural network.

5. Selected methods in AlphaGo, AlphaGo Zero and AlphaZero

AlphaGo has taken the state of the art for search algorithms in games to a higher level by beating the human champion in the game of Go, which is a major challenge for search algorithms due to its huge search space. Despite its simple rules, Go offers a high complexity than can not be searched exhaustively in an short amount of time with current computing hardware. As a result, it is necessary to restrict the search to moves that might lead to promising results. Now the challenge is to define what a promising move looks like in any given situation. Using classical rule based heuristics for the task is an option that is restricted to the game at hand and can obviously not generalize to different games or problems.

AlphaGo's solution is to use deep neural networks that iteratively learn to predict a move's merit and thus guide the search towards promising opportunities. AlphaGo's approach starts by training the neural network on data gathered from human games and subsequently improving the network with data generated from self-play. The improved version of AlphaGo, called AlphaGo Zero, purely relies on self-play reinforcement learning and thus does not need human knowledge to reach superior playing strength.

Besides deep neural networks, AlphaGo and AlphaGo Zero rely on Monte-Carlo Tree Search (MCTS), which is a search algorithm that works by repeatedly taking samples from the search space. In its original form, MCTS takes random samples. A sample is taken by playing a game to the end with random moves and then returning the score of the game as the sample's value. When MCTS employs a heuristic that guides the search, it can become far more powerful. The details of combining MCTS with a heuristic function that can be provided by a neural network are covered in [Ros11] and described in section 5.3.

5.1. The AlphaGo (Zero) algorithm

There are three tasks that are executed asynchronously in parallel:

- Training a neural network on recently generated self-play data.
- Evaluating the performance of recently trained neural networks in combination with Monte-Carlo Tree Search.
- Generating new self-play data with the network that currently has the best evaluation results.

Each of these three tasks will be described in more detail in the following three subsections, which provide a general overview of the concerned task and reference other

sections for further details. Overall, AlphaGo Zero has been trained on 4.9 million games of self-play data.

5.1.1. Training neural networks

A neural network is continually trained on data provided either by sourcing human games of Go or by its own self-play. AlphaGo [Sil+16] first uses human games as examples and then proceeds to improve itself with its own data that was generated from self-play. AlphaGo Zero [Sil+17b] abstains from using human data and solely relies on data generated with self-play. Section 5.2 provides an in depth description of the architecture of AlphaGo Zero's best neural network. Besides the network that is described in section 5.2, the authors also evaluated three different architectures that yielded comparably inferior performance.

Using the most recent 500,000 games of self-play, the authors of AlphaGo Zero trained the network in the Google Cloud with 64 GPU workers and 19 CPU parameter servers. Training happend on mini-batches of 2,048 samples that were selected randomly from the last 500,000 games. After 1,000 training steps they produced a checkpoint of the neural network to be evaluated as described in the following subsection. To optimize the weights of the neural network, the authors used stochastic gradient descend with momentum and learning rate annealing. They also used L_2-regularization to minimize the weights' values. [Sil+17b] Section 4.1 provides details on stochastic gradient descend and section 4.4.2 explains L_2-regularization.

5.1.2. Evaluating neural network performance

Each neural network that was saved during a checkpoint in training was evaluated against the currently best neural network in self-play using Monte-Carlo Tree Search. An evaluation consists of 400 games where the τ parameter in the move selection of Monte-Carlo Tree Search is set to zero, so that it selects the best move available. Section 5.3 describes AlphaGo Zero's variant of Monte-Carlo Tree Search and also details the effect of the τ parameter. A contending network needs to win against the best network by a margin of 55 % to become the new best network.

5.1.3. Generating self-play data

New data is generated by using Monte-Carlo Tree Search (MCTS) with the best currently available network as determined by the latest evaluation. Both players use the same neural network with MCTS to determine their moves. Each run of MCTS is executed for 1,600 iterations, which takes 0.4 seconds per run. According to the paper about AlphaGo Zero [Sil+17b] 4.9 million games of self-play were generated.

5.2. Neural network architecture

AlphaGo Zero uses a network that is made of a residual tower that is connected to two outputs: the policy head that suggests moves and the value head that assesses the chance to win the game. The concept of a *residual network* is introduced in *Deep residual learning for image recognition* by He et. al. in [He+16]. Its main feature is a "skip" or "shortcut" connection that bypasses a few layers and adds the original input to the output of the skipped layers. This principle is usually repeated multiple times and allows for deeper networks, because the skip connections allow for easier training of deep networks. [He+16]

The input of AlphaGo Zero's neural network is made of 17 planes (images) of 19×19 pixels. Each 19×19 plane shows one binary feature of the 19×19 Go board. One pixel in an image plane represents one intersection on the Go board. For each player there is a plane that indicates if a stone of the player's color is present at the pixel's location. In addition to the two planes that represent the current state of the board, there are 14 more planes that provide a history for the previous 7 time steps of the game, totaling 16 planes that represent the last 8 states of the Go board. The final plane represents the current player: it is one in all positions if black is to play and zero in all positions if it is white's turn.

Figure 11 shows a visualization of AlphaGo Zero's network. First, the input planes are fed into a convolutional layer with 256 kernels of size 3×3 with stride 1. Section 4.2.2 goes into detail about convolutional layers, their properties and parameters. After the data passed a rectified linear (ReLU) activation (see section 4.3.1 for an explanation) and batch normalization (see section 4.4.1), it enters the first *Res Block*. A Res Block is short for residual block, which is part of a residual network that was initially described. There are two versions of the AlphaGo Zero network: one uses 19 residual blocks, the other uses 39 blocks.

Inside a Res Block the data takes two routes. One route passes the data through a first convolutional layer with 256 3×3 feature maps, batch normalization, ReLU activation, then through a second convolutional layer with the same configuration and finally another batch normalization. The other route passes the data directly to the output of the first route where both routes are merged before entering a final ReLU activation that concludes the residual block and constitutes its output.

After the data was processed by all residual blocks, it is split up to be passed to two output "heads" that complete the neural network. One output of the network is called the policy head, because it assigns recommendations to all possible actions. The other output is the value head and provides an approximation of the chance to win or lose the game.

The policy output first passes the data through a convolutional layer with two 1×1 kernels, thus reducing the data to two "images". After passing a batch normalization and a ReLU activation, the two images reach a final fully connected layer with 362

units. Consequently the policy output produces a vector with 362 entries that represents all possible moves: $19 \cdot 19 = 361$ intersections on the Go board plus the option to make a pass move. See section 4.2.1 for an explanation of fully connected layers.

The value output reduces the incoming data to a single image with a convolutional layer that has one 1×1 kernel. After passing a batch normalization and a ReLU activation, the image enters a fully connected layer with 256 units and is then processed by a ReLU activation function. The resulting vector with 256 entries is handed over to another fully connected layer with just one unit, creating a single output value. Finally the output value is limited to the range from -1 to 1 by a tanh activation function, which concludes the value head. Section 4.3.3 provides details on tanh activation functions.

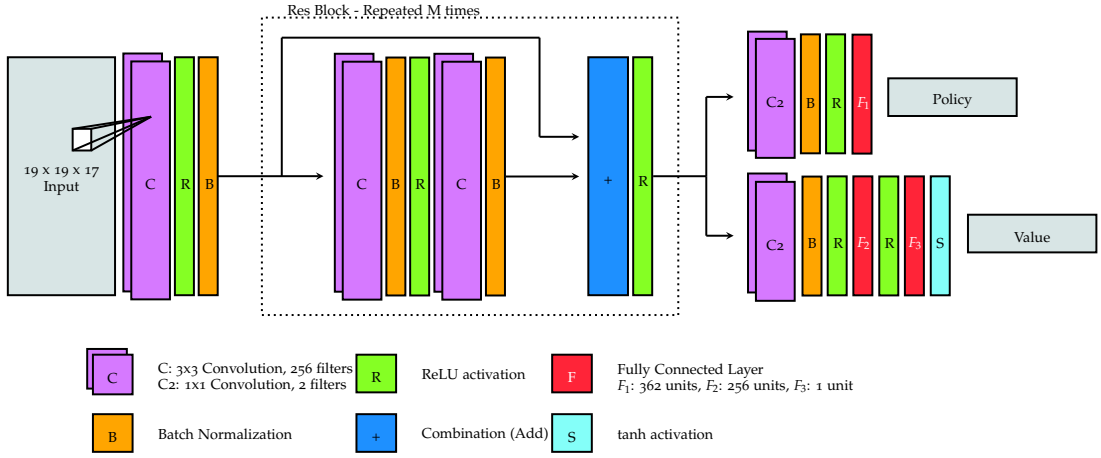


Figure 11: AlphaGo Zero's neural network architecture according to [Sil+17b]

Besides the described network, Silver et. al. evaluated three other network architectures that will only be described briefly here:

dual–res This is the network that was described in detail. Two outputs are connected to one "tower" of residual blocks.

sep–res Instead of using one tower of residual blocks, this variant has two towers – one for each output. As a result there are two independent networks instead of one with two outputs.

dual–conv This network does not use residual blocks. It is instead made of 12 convolutional layers that end with two outputs.

sep–conv This variant is made of two independent networks that are made of 12 convolutional layers each to provide the policy and value output.

According to the evaluation in [Sil+17b] the dual–res variant performs best.

5.3. Monte-Carlo Tree Search

Monte-Carlo Tree Search (MCTS) works by repeatedly evaluating different outcomes of the game. Starting from a given position to analyze with MCTS, the algorithm proceeds through three stages where a tree structure – called Monte-Carlo tree in this thesis – is built and updated. Beginning with an empty Monte-Carlo tree, MCTS iteratively expands it and updates nodes. Each node represents one state s of the game's board. A node in AlphaGo Zero [Sil+17b] contains the following information, where $a \in A(s)$ is one action of all legal actions A in state s:

$N(s,a)$ stores the visit count. Every time a node is traversed, its visit count is incremented by one.

$W(s,a)$ represents the total action value that is the sum of all scores that were backed up through the node.

$Q(s,a)$ saves the mean action value which is the result of dividing the node's total action value by its visit count:

$$Q(s,a) = \frac{W(s,a)}{N(s,a)}$$

$P(s,a)$ is the prior probability of selecting the move a in state s assigned by the neural network.

Starting from an initial state s_0, AlphaGo Zero MCTS iterates the following three steps 1,600 times. Figure 12 shows a visualization of the three stages.

Selection The existing Monte-Carlo tree is traversed until a leaf node is reached, which is then returned as the result of the selection. The path through the tree is determined by a combination of an action value $Q(s,a)$ and the result of the PUCT algorithm $U(s,a)$ that takes the prior probability $P(s,a)$ into account. For each layer t in the Monte-Carlo tree, a move a_t is selected which facilitates the transition from the state s_t to the next state s_{t+1}. An action a_t is selected using the following method:

$$a_t = \underset{a}{\operatorname{argmax}}(Q(s_t,a) + U(s_t,a))$$

where $U(s_t,a)$ is defined as follows:

$$U(s,a) = C_{PUCT} \cdot P(s,a) \cdot \frac{\sqrt{\sum_b N(s,b)}}{1 + N(s,a)}$$

With $U(s,a)$ the search makes use of the PUCT algorithm [Ros11] that balances exploration of new moves and exploitation of strong moves. The balance is determined by the constant hyperparameter C_{PUCT}. Using the prior probability $P(s,a)$ assigned by the neural network, the search is guided towards promising new moves.

Expansion and Evaluation The state of the leaf node s of the Monte-Carlo tree is sent to the neural network or evaluated directly if it is a terminal state. After the network processed the given state, a child node is created for each valid move a. All child nodes are initialized as follows: the visit count $N(s,a)$ and mean action value $Q(s,a)$ are set to zero while the prior probability $P(s,a)$ receives the network's estimated probability of making move a. Finally the total action value is populated with the network's assessment of the chance to win the game.

Backup During the last phase, the total action value is backed up through the Monte-Carlo tree along the path that was taken in the selection step. Each node on the traversed path receives the following updates:

- $N(s,a)$ is incremented by one.
- $W(s,a)$ is incremented by the leaf node's action value.
- $Q(s,a)$ is updated according to the previously described formula.

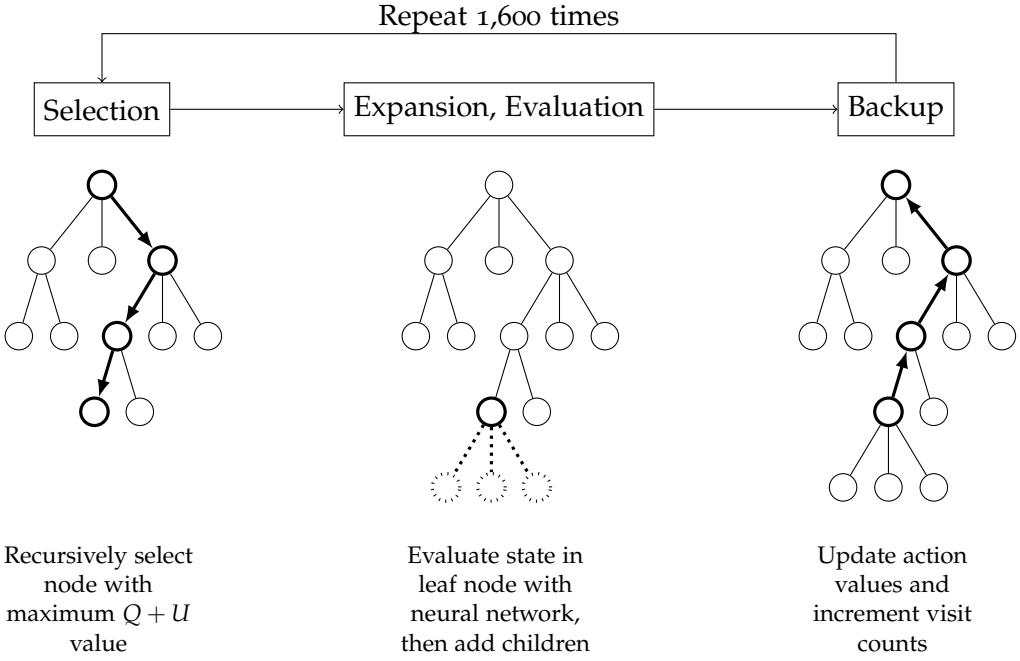


Figure 12: Simplified visualization of AlphaGo Zero Monte-Carlo Tree Search as described in [Sil+17b]

After the three steps were executed 1,600 times, a move to play is selected and returned as the result of the algorithm. The decision is based on the visit count of the root node's children. To select a move, AlphaGo Zero first defines the policy π for a given initial state s_0 as follows while using a temperature parameter τ that controls

exploration:

$$\pi(a|s_0) = \frac{N(s_0, a)^{\frac{1}{\tau}}}{\sum_{b \in A} N(s_0, b)^{\frac{1}{\tau}}}$$

Finally a move $a \in A$ is selected randomly from all valid moves A according to the distribution defined by π.

Due to the definition of π, the output of the policy head is normalized so that the sum of π for all actions in s_0 is one. Normalization is facilitated by dividing each action value $N(s_0, a)$ by the sum of all action values for s_0. Using τ, the level of exploration can be controlled by increasing or decreasing the difference between the values for all actions. Since the selection of a final move happens according to the probability distribution defined by π, the use of τ controls exploration versus exploitation by equalizing or amplifying the difference between probabilities. If τ is lower than one, the value of $N(s_0, a)$ is exponentiated with a value that is bigger than one and thus the difference between the values for different actions is increased, which results in less exploration: Moves with a high initial value will be raised far higher than moves with lower initial values. This effect is amplified with increasingly lower values for τ. Conversely the values for various moves are moved closer together when τ is bigger than one, because then $N(s_0, a)$ is exponentiated wit a value lower than one.

6. Applying AlphaGo Zero's methods to Dots and Boxes

This section will describe the methods employed to realize an opponent in Dots and Boxes that is based on Monte-Carlo Tree Search and deep neural networks. First there is a description of the neural networks and how they are integrated into KSquares, which is an application for playing Dots and Boxes. Afterwards section 6.5 will provide insight into the Monte-Carlo Tree Search algorithm for Dots and Boxes. Finally section 6.6 explains the Self-Play reinforcement learning methods that tie together neural networks and tree search to create a powerful opponent that is able to improve itself. In reference to AlphaGo, this incarnation of the AlphaZero algorithm for Dots and Boxes is called AlphaDots.

6.1. Using neural networks to play Dots and Boxes

Initially, KSquares was extended by two facilities to have a neural network play Dots and Boxes:

- Means to generate training data for a neural network.
- An interface to connect a network's input and output with KSquares so that the network can play Dots and Boxes against human players and other AIs.

For the means to generate training data, section 6.2 describes the various types of data generators that were implemented in KSquares for this thesis. Common to all data generators is the ability to create input and output data for neural networks in a suitable format. Since all networks for this thesis are built with Python in Tensorflow [Mar+15] and Keras [Cho+15] the data generators put data into *.npz* files, which are the native format of the underlying library NumPy [Oli06] that is used by Tensorflow and Keras.

To play Dots and Boxes against a neural network, KSquares provides *AI ConvNet*, which handles sending data to and receiving data from various types of neural networks. Depending on the type of neural network, the game's state is converted accordingly and sent to the network. The network's output assigns a value to each line and is further interpreted as follows:

- Only lines that are not yet drawn are considered. If the neural network assign the highest value to a line that has already been drawn, the interface in KSquares will ignore this and select the valid line with the highest assigned value.
- If multiple lines share the same highest value, the interface will randomly select one of them.
- If the network provides an invalid value like NaN, which means "'not a number"', KSquares will intentionally crash. Explicitly crashing KSquares on NaN is done to offensively indicate problems with the network.

The *AI ConvNet* in KSquares allows the user to dynamically select the model (neural network) to play against.

6.2. Training data

There are different types of training data that will be described in the following sections. Each of the following sections is named after the associated data generators' names. Different data generators often share the same basic format for input and output data. Therefore the sections are named after all data generators that provide the described format.

In some cases, the data was further transformed with Python's NumPy library before being fed into a neural network. Such transformations will be described in section 6.3 and are mentioned in section 6.4 which is about the neural networks' architectures, because the transformations are specific to the networks.

6.2.1. FirstTry, StageOne, StageTwo

In this representation both the input Dots and Boxes board and the expected output of the neural network are images. Both input and output image share the same size which depends on the number of boxes in the original board. The output image is always made of exactly one white pixel denoting the line that KSquares' Hard AI would choose in the given situation.

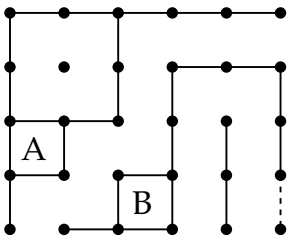


Figure 13: Original board

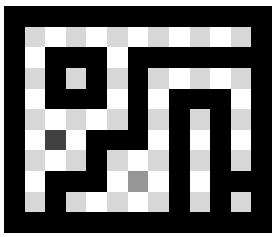


Figure 14: Input image

Figure 15: Output image

Figure 13 shows a Dots and Boxes board in the usual format that is used throughout this thesis, while figure 14 shows the same board as an image that is used as input for neural networks. The board shows a long chain, a cyclic chain and a short chain that was opened with a hard hearted handout. A dashed line in Figure 13 indicates the expected line from Figure 15. The images depicted in Figure 14 and Figure 15 are scaled up so that pixels are easily visible. In the input and output images, all elements like dots, lines and boxes are reduced to single pixels. Lines are white, dots and boxes are shades of gray and the background is drawn in black.

The board state that is used as the basis for the input image is generated as follows: A game is played by the fast, rule based Hard AI up to a random point. Then the board's state is converted to the input image. Afterwards the position of the next move of KSquares' Hard AI is marked on an otherwise empty output image.

6.2.2. BasicStrategy

During the first phase of Dots and Boxes, KSquares' Hard AI often selects a line randomly from a rather large pool of possible lines that do not give away boxes to the opponent. To provide the neural network with all possible options at the same time, the *BasicStrategy* data generator creates output images where all acceptable lines are shown.

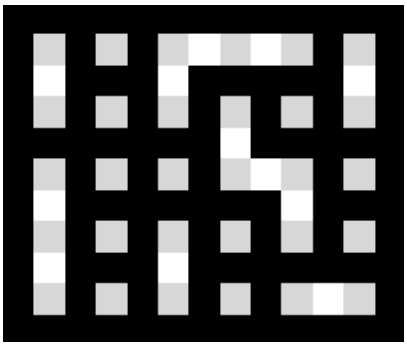


Figure 16: Input image

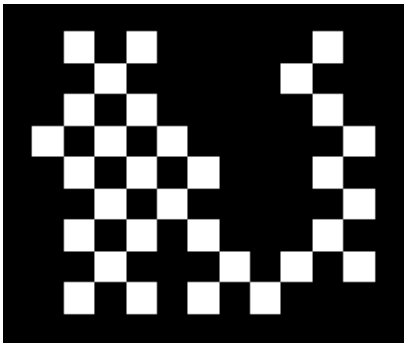


Figure 17: Output image

Figure 16 shows an input image generated by the *BasicStrategy* data generator. Its format does not differ from the format described in section 6.2.1. Figure 17 shows the corresponding output image where many valid lines are shown as white pixels.

6.2.3. Sequence, TrainingSequence

The *Sequence* data generator creates data that shows a full Dots and Boxes game as a sequence of images like those in Figure 14. Both input and output data are in the format of Figure 14, which includes all elements like boxes and dots in the output image. Consequently, neural networks are trained to also output irrelevant parts of the game. Conversely, the *TrainingSequence* generator produces output sequences like the *BasicStrategy* generator.

6.2.4. StageThree

Introducing the value output, the *StageThree* data generator provides three groups of data:

Input The input data is unchanged compared to previous data generators. Figure 14 shows an example image.

Policy The policy output data is a vector of all possible lines where all entries are 0, except for the line selected by KSquares' Hard Ai, which is set to 1. All lines are indexed left to right, top to bottom.

Value The value output is a single real value in the range from -1 to 1. It is meant to reflect the chance to win the game where -1 is a certain loss and 1 an assured win.

Like in previous data generators, a game of Dots and Boxes is played up to a random point by KSquares' Hard AI to generate an input image and the policy output. To calculate the value output, the game is played to its end by the rule based Hard AI. Then the number of captured boxes by player A is subtracted from the number of captured boxes by player B. Here player A is the player to make a move on the generated input image. The difference of captured boxes is then divided by four fifth of the overall number of boxes. Formally the value V is calculated by subtracting the boxes of player A from the boxes of player B and dividing the result by 0.8 of the number of all boxes G.

$$V = \frac{A - B}{0.8 \cdot G}$$

The motive behind using 80 % of all boxes as divisor instead of the full number is that, due to Double Dealing in Dots and Boxes, a weak opponent will still score a few boxes. Thus even optimal play probably will not capture all boxes. Since V should reflect the chance to win the game, the difference between captured boxes was amplified.

6.2.5. StageFour

StageFour is an improved version of the *StageThree* data generator. Like its predecessor, it creates data for one input and two outputs. Calculating the number of moves before an input image is generated has been simplified: The *StageFour* uses a parameterized Gaussian distribution to determine the number of moves left to play in an input state. Figure 18 shows a histogram of 1,000,000 samples of *moves left* for a 3×3 board with default parameters.

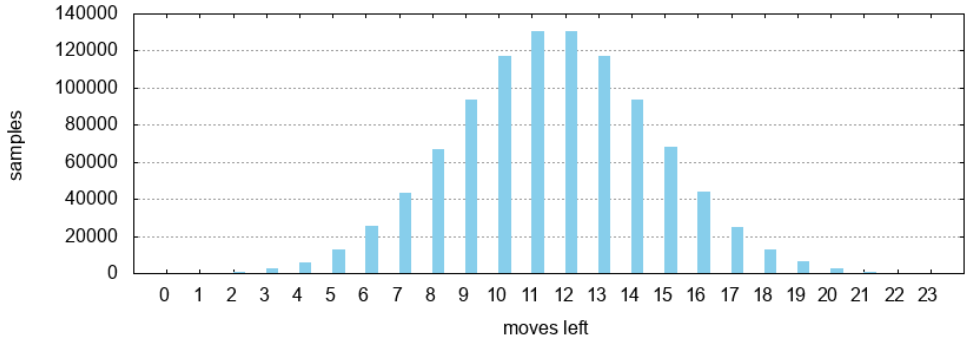


Figure 18: Histogram of *moves left* for 1,000,000 samples

There are two hyperparameters that control the distribution of *moves left* in the *StageFour* data generator. Both parameters act as scaling values for the number of lines. The parameters σ and μ influence the Gaussian distribution $G(s)$. Adapting σ will change the deviation from the mean of the distribution. The mean of the distribution can be influenced by μ. By default σ is set to 0.125 and μ is set to 0.5 so that most samples are in the middle of the game. The total number of lines is denoted as L and depends on the configured board size for the data generator.

$$\text{moves left} = G(\sigma L) + \mu L$$

Figure 19 shows a histogram for 1,000,000 samples where σ = 0.2 and μ = 0.6 to move the focus of the data generator towards an earlier phase of the game and distribute the generated samples more evenly over the whole game.

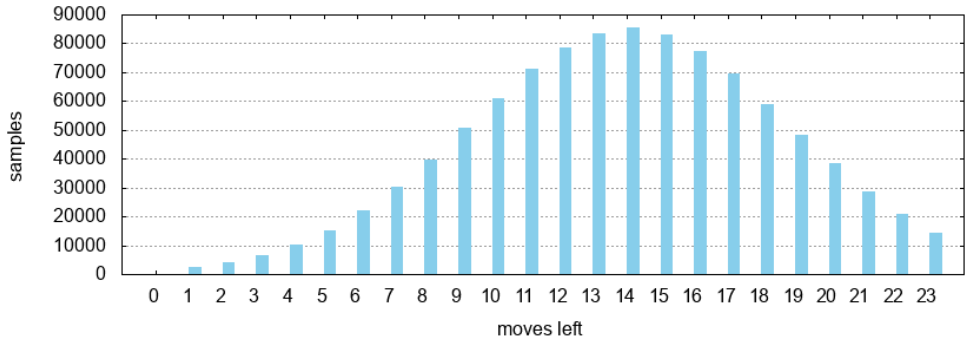


Figure 19: Histogram of *moves left* for 1,000,000 samples with σ set to 0.2 and μ set to 0.6

The *StageFour* data generator is used in a simplified self-play reinforcement learning loop. It generates a game with the Hard AI, until there is a given number of moves left. Then it can be configured to either use the Hard AI or the AlphaZero MCTS AI

to generate the policy output. If the policy output was generated with the Hard AI, the game is played to completion to calculate the value V as follows:

$$V = \frac{A - B}{G}$$

Otherwise the value V is taken from the selected child node's value in the Monte-Carlo tree.

When the *StageFour* data generator uses the AlphaZero MCTS AI, it is hypothesized to create improved training data that contains stronger moves than the data that AlphaZero's network was originally trained on. Using this improved data to train AlphaZero's network should make it possible to create a self-play reinforcement learning loop with a minimized amount of necessary computing power.

6.3. Data transformations

After the data is generated with KSquares' data generators, it is transformed in some cases. Usually the image data is normalized so that a black pixel corresponds to 0 and a white pixel corresponds to 1. Besides simple normalization, data was transformed in two ways:

- Input images were split up into multiple planes as described in the paper about AlphaZero. [Sil+17a]
- Some neural networks generate a vector of all possible lines instead of full images.

The following subsections will describe both transformations in detail.

6.3.1. Input transformation

According to the description of AlphaGo and AlphaZero, the input data for neural networks was provided in multiple planes where each plane contained only one feature of the game. For chess, Silver et. al. used six planes per player for the six types of pieces in chess: king, queen, rooks, bishops, knights and pawns as described in table S1 in [Sil+16]. In the previously described images of Dots and Boxes boards there are five different elements:

1. Background
2. Lines
3. Boxes captured by player A
4. Boxes captured by player B
5. Dots

For this reason, an input image is split up into five planes. An example is shown in figure 20 where the five planes are shown below the original input image. In each plane a pixel is either black or white. All pixels are black except those pixels at positions where the element is present in the original image.

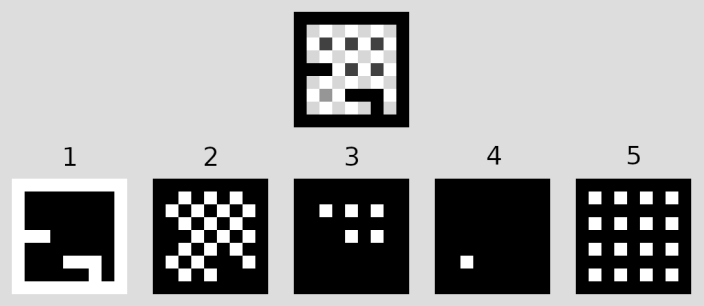


Figure 20: Dots and Boxes board image and the five planes generated from that image

6.3.2. Output transformation

Early attempts at designing neural networks required the network to generate output data as full sized images where the network had to provide predictions for pixels that were simply ignored by KSquares. Newer networks were designed to only predict values for the lines to reduce the overhead of predicting pixel values that are subsequently ignored. In the reduced version, the output is a vector where each element represents one line.

To facilitate the output as a vector of lines, a custom layer called `LineFilterLayer` was created that extracts all line pixels from a typical output image (like in figure 15) and puts them in a vector. The custom Keras layer uses Tensorflow's `BooleanMask` function.

6.3.3. Data augmentation

Since it takes a comparably long time to generate samples with Monte-Carlo Tree Search, an `AugmentationSequence` class was developed which augments given data in a suitable format for the `fit_generator` method in Keras. Augmenting data works by randomly applying one of the following mutations:

- Leave the data unchanged.
- Flip the data horizontally.
- Flip the data vertically.
- Flip the data horizontally, then vertically.

By using data augmentation, the amount of available training data is increased.

6.4. Network architectures

This section describes the various network architectures that were implemented to play Dots and Boxes. The first attempts are rather simple networks with poor performance and are not based on the architecture of AlphaGo or AlphaZero.

All models are fully convolutional – they are able to handle input images of arbitrary size. Consequently they are able to play Dots and Boxes on any given board size.

6.4.1. First Try, Stage One, Basic Strategy

This model is a 7 layer convolutional neural network. Each convolutional layer consists of 64 filters with 3 x 3 kernels. After the activation of each convolutional layer, batch normalization is applied to the data. The first six layers use the ReLU activation function while the last layer uses softmax. The softmax activation function works on each pixel individually, by treating each pixel as a boolean classification task. Figure 21 visualizes the model's previously described architecture.

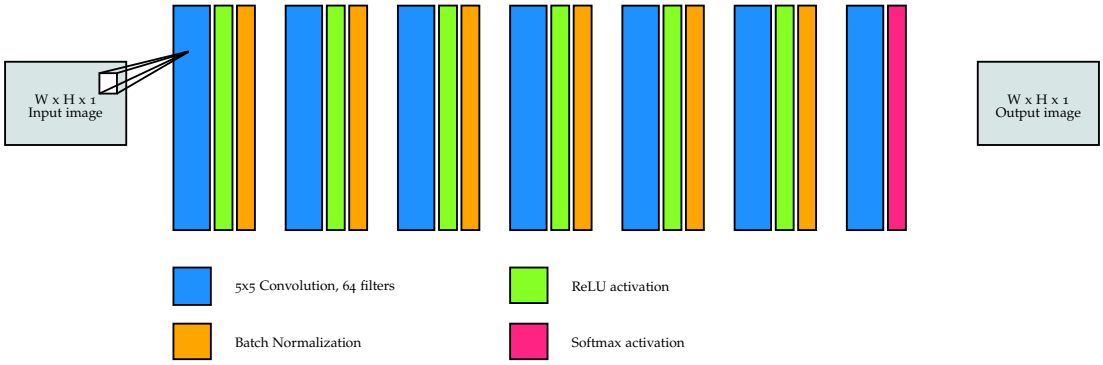


Figure 21: Network architecture of the First Try, Stage One and Basic Strategy models

The model was trained with stochastic gradient descend using a categorical crossentropy loss function. Training occurred for 50 epochs on a small dataset of 10,000 examples generated with the FirstTry data generator.

The input for the model is a grayscale image of the game's state. The image data is normalized so that black corresponds to 0 and white to 1.

There are two *Stage One* neural networks. The first one has the same architecture as the *First Try* model from the previous section, while the second version differs in one

aspect: it has 5×5 instead of 3×3 kernels. As the name suggests, both networks were trained on data generated by the *StageOne* data generator.

Besides the *First Try* and *Stage One* networks there is another network that shares the same architecture: *Basic Strategy*. Here the training was executed on data from the *BasicStrategy* data generator.

6.4.2. LSTM

The *LSTM* network architecture is built with Keras' `ConvLSTM2D` layer [Shi+15] which combines Long Short Term Memory (LSTM) units with convolutional input transformations and convolutional recurrent transformations. This architecture is based on an example provided by Keras for predicting the movement of boxes in a short image sequence.

Data is fed into the network as a sequence of length L that starts with an empty board and leads up to the state of the game where the network should make a move. The network's output is the predicted next image for the provided sequence.

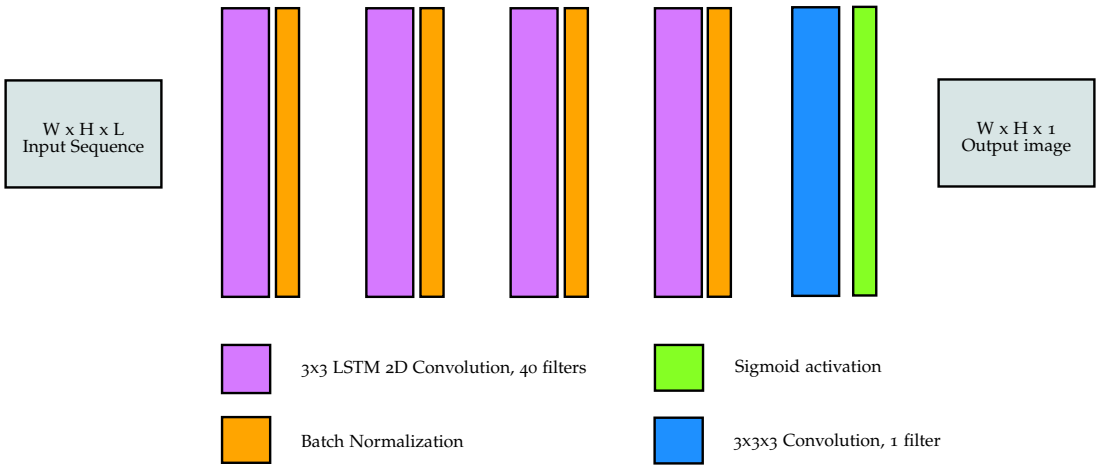


Figure 22: Network architecture of the LSTM model

Figure 22 shows a visualization of the LSTM model. The input sequence has a dimensionality of $W \times H \times L$ where W represents the board's image width, H the height and L is the number of images in the sequence. Each image in the input sequence shows one more line than the previous one. Processing of the sequence happens in four layers made of batch normalized `ConvLSTM2D` units with 40 filters and a final three-dimensional $3 \times 3 \times 3$ convolutional layer that produces an output image with a sigmoid activation function.

6.4.3. AlphaZero network architecture for Dots and Boxes

Heavily inspired by the AlphaGo Zero template described in the methods section of [Sil+17b], this model architecture is made of convolutional layers that are arranged in residual blocks which have an extra shortcut connection that skips layers. In contrast to the original architecture, this incarnation is made fully convolutional by replacing the final dense layers with custom ones that support arbitrary input sizes. Thus it is possible to play Dots and Boxes on different board sizes with the same network.

Data is provided in five input planes where each plane represents one element of the original input image. See figure 20 for an example of an input image that is split into five planes. After being passed through one convolutional layer with N filters, ReLU activation and batch normalization, the data enters the first "Res Block". Res Block is a short form of *residual block* and it is proposed in [He+16]. In a Res Block the data takes two ways: It is processed and a copy of it takes a shortcut.

Processing Data is passed through a convolutional layer with N filters, batch normalization, ReLU activation, a second convolutional layer with N filters and a second batch normalization before it enters a combination layer that reconciles both paths.

Shortcut Data is passed directly to the combination layer and thus takes a shortcut through the Res Block.

The combination layer adds up both input and processed data and puts the result through a ReLU activation. This process is repeated M times until finally the data reaches the end of the Res Blocks where it is again split up for the two outputs of the network: the value head and the policy head.

Value Incoming data is reduced to a single plane by a convolutional layer with one filter. Afterwards the custom value layer[5] reduces the image to a single value by behaving like a classical dense layer with one neuron without bias where all weights are set to $\frac{1}{\text{number of pixels}}$. The resulting value is passed to a tanh activation function so that the value output is between -1 and 1.

Policy Incoming data is reduced to a single plane by a convolutional layer with one filter. Then the resulting image is fed into the custom made line filter layer, which extracts the lines' pixel from the image and puts them into a vector. Finally the vector is passed to a softmax activation function. Due to the softmax function, the sum of all policy vector entries is 1 and the vector can be interpreted as a probability distribution.

Figure 23 provides a visualization of the preceding description of the network's architecture.

[5] A simpler alternative to the custom layer would have been Keras' `GlobalAveragePooling2D` layer. Unfortunately the layer was overlooked until the thesis was almost done.

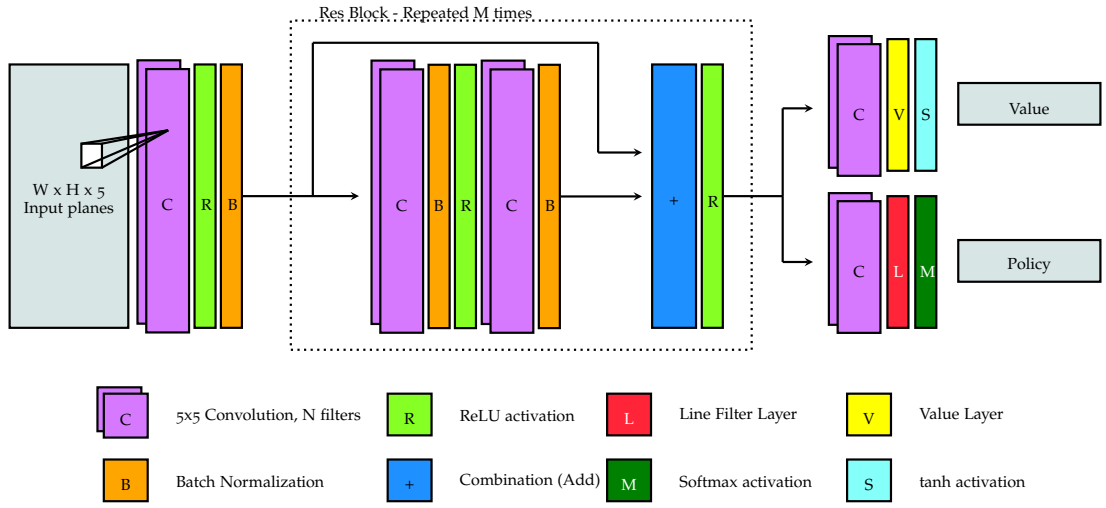


Figure 23: Network architecture of the AlphaZero model

Depending on the instance, the values for N are 64, 128 or 256 while M is either 4, 8 or 16. In the chapter about evaluation the values for N and M are explicitly stated for each model.

6.5. Monte-Carlo Tree Search

In AlphaDots, the Monte-Carlo Tree Search algorithm is based on the description in the paper about AlphaGo Zero [Sil+17b], which is also described in section 5.3. Similar aspects include the information stored for the nodes in the Monte-Carlo tree:

$N(s,a)$ is the visit count for move a in state s.

$W(s,a)$ is the total action value for move a in state s.

$Q(s,a)$ is the mean action value, calculated by $\frac{W(s,a)}{N(s,a)}$.

$P(s,a)$ is the prior probability to make move a in state s as assigned by a neural network.

Further similarities include the three steps Selection, Evaluation & Expansion and Backup which are described in section 5.3.

There are a few points where the implementation of AlphaDots diverges from its paragon:

- AlphaDots has two methods for the final move selection.

- AlphaDots has the option to aggregate moves so that a node in the Monte-Carlo tree may represent multiple moves that belong together.

6.5.1. Final move selection

AlphaDots supports two methods to select the final move that is returned by the algorithm. The first method is based on AlphaGo's method of defining a policy π with a temperature parameter τ and then using π as a probability distribution to randomly select a move. A detailed description of this approach can be found in section 5.3 about AlphaGo's final move selection method. Additionally AlphaDots has a method that simply maximizes the expected output:

$$\underset{a \in A}{\operatorname{argmax}} \frac{N(s_0, a)}{\sum_{b \in A} N(s_0, b)}$$

where a is an element of all valid actions A in the state s_0 that is the input to the MCTS algorithm.

Due to the softmax activation function in the AlphaZero network architecture for Dots and Boxes (see section 6.4.3) the sum of the network's policy output is already normalized to 1. This is in contrast to the original AlphaGo Zero network architecture, that does not have a softmax activation function and solely uses the definition of π for normalization of the output. Despite the softmax feature of providing a vector that sums up to one, it is necessary to normalize in the move selection function, because the interface in KSquares discards all invalid lines. As a consequence, only a subset of all lines is considered when selecting a move, which can lead to $\sum_{b \in A} N(s_0, b)$ being smaller than one, if the network assigned an invalid move with a value bigger than zero.

6.5.2. Move aggregation

AlphaDots supports two modes for searching moves with Monte-Carlo Tree Search. The first mode does not use any knowledge of the game and thus treats each undrawn line as a possible option to be considered by the search algorithm.

Building on previous work on Dots and Boxes, a second mode uses move sequences as they were generated for the Alpha-Beta search algorithm for the related bachelor's thesis. [Pet15] These move sequences reduce the search space by leaving out some options that are guaranteed to not contribute to any optimal strategy and those that are equivalent to other moves. Excluding certain moves is based on the work by Barker et. al. who solved 5×4 Dots and Boxes. [BK12] Move sequences for example only include one line for each corner of the field since drawing either of both sides of a corner would result in equivalent board states.Besides providing one representative for equivalent corner moves, the sequences only include moves that fully capture a chain

or end with Double Dealing, leaving out all move sequences where capturing a chain is interrupted half way through. [Pet15] Using move sequences provides the algorithm with rather powerful knowledge of the game. As a consequence it is possible to easily disable the usage of move sequences with a command line and GUI option so that experiments with the raw abilities of the search algorithm can be conducted.

6.5.3. Performance

Executing Monte-Carlo Tree Search with an Nvidia GTX 970M GPU for 1,600 iterations on a 4×4 board takes about 40 seconds. In contrast the execution of AlphaGo Zero's Monte-Carlo Tree Search takes 0.4 seconds. [Sil+17b] Besides more powerful hardware, AlphaGo Zero most probably benefits from a more efficient implementation than AlphaDots.

6.6. Self-Play reinforcement learning

AlphaDots is completed by the ability to use self-play to surpass its previous level of skill. Self-play ties together the two central aspects of the AlphaDots implementation:

- Training a neural network that is then able to approximately reach the playing strength presented in the training data.
- A MCTS algorithm that is able to leverage the knowledge of a neural network to find even better moves by executing a well informed, directed sampling of possible moves.

The self-play functionality of AlphaDots supports two modes of operation. First, it generates training data with the currently best neural network. Afterwards it trains the best neural network on the newly generated data. Depending on the mode of operation, AlphaDots does one of two things:

1. Assume that the newly trained network is better than the previous one and simply use it as the new best network.
2. Evaluate the newly trained network and only use it if it won more than 50 % of all games.

All three parts of AlphaDots self-play functionality support a wide range of options that govern different aspects of their operation.

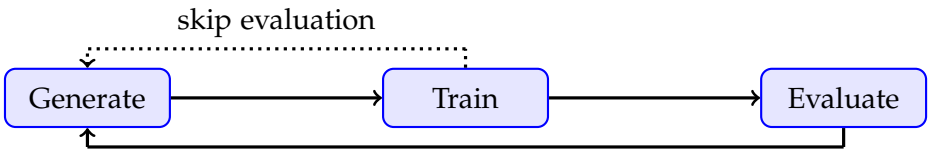


Figure 24: Flow of self-play reinforcement learning in AlphaDots

6.6.1. Generating data

In an attempt to require less computational resources, the self-play reinforcement learning method of AlphaDots deviates from the methods used by AlphaGo Zero. Instead of playing full games with Monte-Carlo Tree Search and sampling from them, AlphaDots generates a board state with the Hard AI and then uses a single pass of Monte-Carlo Tree Search with 1,500 iterations to produce the expected output. AlphaDots also provides an option to set a custom number of MCTS iterations to allow for fewer iterations in experiments, thus further reducing computational costs.

Section 6.2.5 provides a detailed description on how the data is generated. Besides the default behavior of generating data with MCTS, AlphaDots also supports all other integrated AIs when it generates data with a dedicated command line option. As a result, the self-play mode can be used to train a neural network as described in section 6.4.3 on various types of data.

6.6.2. Training new networks

To improve the currently "best" neural network, it is trained on the data that was generated in the first phase of self-play. AlphaDots supports two modes of training a neural network. First, there is the classical method of training directly on the provided data. Since it takes a long time to generate new data with MCTS, AlphaDots also has the ability to increase the number of samples by augmenting the given data. Data augmentation in AlphaDots is described in detail in section 6.3.3.

Besides either training with or without data augmentation, AlphaDots offers a cumulative training mode if the evaluation phase is enabled. By default AlphaDots discards a network that was not able to beat the best network in at least 50 % of the games. When it uses cumulative training, the failed network is not discarded but instead used as the starting point for training in the next iteration. As a result the knowledge that is contained in the generated training data iteratively accumulates in the contending network, even if it does not win against the best network immediately.

6.6.3. Evaluating new networks

Further diverging from the original methods in AlphaGo Zero, this implementation does not enforce a model evaluation but also supports the option to repeatedly generate data and train the network on it, without evaluating the playing strength of the new network after training.

During evaluation, instead of playing full games to evaluate the playing strength, AlphaDots initializes the boards with a few random moves to reduce the number of moves that have to be computed. The number of initial random moves is set so low that the game is in the middle of the forming phase, which is explained in section 3.4.1. It is also possible to disable the random board initialization. Nevertheless in some cases it is not advisable to disable this functionality, because empirical results have shown that AlphaDots makes mostly deterministic moves, which results in an evaluation where it almost always plays the same game against itself.

7. Evaluation

This section describes the efforts to evaluate the playing strength of various AIs for Dots and Boxes. The first part covers notable AIs for Dots and Boxes as it details the general methods of those AIs. Two ensuing sections present methods and results of evaluations that were conducted for this thesis.

7.1. Existing AIs for Dots and Boxes

This section describes some notable AIs for Dots and Boxes. Many of the AIs that are listed here are integrated into KSquares to allow for easy evaluation of their playing strength in comparison to other AIs.

7.1.1. KSquares – Easy, Medium, Hard

There are three rule based AIs in KSquares that mimic typical novice human strategies. All three AIs are very fast and rely on hand crafted rules to play the game. The AIs have been available in KSquares before the beginning of the related bachelor's thesis. [Pet15]

Easy During the forming phase, this AI randomly draws lines that do not give away boxes. In the endgame phase it opens chains randomly.

Medium The Medium AI behaves like the Easy AI but during the endgame it opens chains ordered by length to give away fewer boxes.

Hard Besides the skills of the Medium AI, it is able to do Double Dealing. Furthermore it exclusively opens short chains with a hard hearted handout to avoid that the opponent can apply Double Dealing in those chains.

Table 2 shows the evaluation result of the Easy, Medium and Hard AIs playing against the Easy, Medium and Hard AIs. Each combination played 10,000 matches where 5,000 games started with one AI and 5,000 games started with the other one.

	Wins vs. Easy	Wins vs. Medium	Wins vs. Hard
Easy	50.36 %	9.43 %	2.64 %
Medium	90.82 %	49.72 %	8.93 %
Hard	97.71 %	91.11 %	49.50 %

Table 2: Results of Easy, Medium and Hard AIs playing against each other

The results show that when one of the AIs plays against itself, it wins about half of the time. Furthermore it becomes clear that each AI deserves its name in terms of

playing strength – Medium beats the Easy in 90 % of the games while it loses against the Hard AI which beats its subordinate counterparts in most cases. The results can be reproduced in mere minutes by running the Docker Image mentioned in section 7.3 with the parameter `EasyMediumHard`.

7.1.2. KSquares – Alpha Beta

The Alpha Beta AI ($\alpha\beta$)was the main research topic for the related bachelor's thesis. It is based on the minimax search algorithm and was enhanced with alpha-beta pruning, move ordering, transposition tables and move sequences that reduce the search space. One central feature of the search algorithm was the ability to order moves in a way that supports the alpha-beta pruning mechanism, which relies on a move ordering that places potentially strong moves first. The code to generate all possible move sequences has been reused in the implementation of the Monte-Carlo Tree Search algorithm for this thesis. Besides move ordering, the Alpha Beta AI makes use of transposition tables which enable the algorithm to efficiently reuse earlier search results. If the algorithm does not explore at least 10 % of the available lines in the root board configuration, it uses a flat heuristic search to select a move to prevent early preemptive sacrifices during the forming phase. [Pet15]

7.1.3. KSquares – ConvNet

The ConvNet AI in KSquares was written for this thesis. It allows the user to directly play against a configurable neural network. To avoid many invalid moves, this AI provides an slightly fault tolerant interface for the neural networks: it converts the given board to an image like the one shown in figure 14 and sends it to the network. After receiving the network's answer it handles the data as follows:

- Only valid lines are considered. If the network assigns values greater than zero to invalid lines, they are ignored.
- The line with the highest assigned value is selected. If two or more lines share the maximum value, a line is selected at random.
- If the network provides an invalid value like NaN, it will throw an exception.

7.1.4. KSquares – AlphaZero

The AlphaZero AI in KSquares was written for this thesis. Section 6.5 provides an in depth description of the methods used in this AI. KSquares' user interface provides means to adapt all hyperparameters of this AI and to select the neural network that is used to support the search.

7.1.5. Dabble

Dabble was written by J. P. Grossman in 2000. It works with an highly optimized implementation of minimax search that is enhanced with alpha-beta pruning and many other features. Internally it relies on the Strings and Coins board representation. [Grooo] Dabble was integrated into KSquares for the related Bachelor's thesis.

7.1.6. QDab

QDab was created by Yimeng Zhuang in 2014 and is built with Monte-Carlo Tree Search and a feed forward neural network. It exclusively plays on boards with 5×5 boxes and is the strongest AI for Dots and Boxes according to [Pet15]. QDab uses a feed forward artificial neural network with one input layer, one hidden layer and a one unit output layer. The neural network provides a real-valued assessment of the provided board from the perspective of the current player where -1 represents a certain loss of the game, while an output value of 1 predicts a sure win. Unlike AlphaZero, QDab uses the neural network exclusively for the simulation step of the Monte-Carlo Tree Search – the neural network is used to predict the winner of the game instead of playing it to its end with random moves. The network's input layer has 25 units which receive certain features that are extracted from the actual board configuration. Those features are the number of chains on the board, the number of loop chains and the number of nodes with certain valences when viewed in the Strings and Coins representation. A central part of QDab is a new board representation which groups the chains on a Dots and Boxes board in 12 different categories, which are used as the input features of its neural network. Using minimax search, training data for the neural network was generated. [Zhu14]

7.2. Evaluation framework

Although there already was a framework for evaluating the playing strength of AIs within KSquares, a new improved version was developed for this thesis. In contrast to the first evaluation framework, the new one provides a user interface that presents the current progress of evaluation and offers functionality to create human readable evaluation reports in the markdown format. Additionally it provides an interface to analyze games and provides the option to count the occurrences of Double Dealing.

In the new framework there are two modes of operation that use different Dots and Boxes engines. Firstly there is the slow, visual mode that utilizes the same engine as the normal KSquares interface for playing Dots and Boxes. Secondly there is the fast, multi-threaded mode that does not display games but instead focuses on fast execution of many games in parallel. Both modes feed the same result storage, which

facilitates a thorough record of played games and associated events like any errors that occurred.

7.3. Replicating the presented results

The following sections present the approach and results of experiments that evaluate the playing strength and other properties of various neural networks and of their combination with Monte-Carlo Tree Search. The experiments are designed to provide answers to the posed research questions. Before presenting the experiments, this chapter explains how the neural networks can be created, trained and evaluated by providing instructions on how to easily replicate the experiments. To ensure that the experiments are easily reproducible, they are made available in a Docker Image. The Docker Image and instructions on how to run it are available at the following address:

`https://tom.vincent-peters.de/master/`

The integrity of the downloaded Docker Image can be verified by computing the SHA 256 hash of the downloaded image as shown in the first line and comparing it with the value presented in the second line.

```
1 shasum -a 256 ksquares.tar.gz
2 5d301d5b5ec33ae082ae2d10987a57b60565adb87fc0a905de184802519c3180
```

Besides the Docker Image, the website also provides a detailed technical documentation of KSquares and AlphaDots, including command line arguments and in-depth descriptions of the implemented neural networks.

7.4. Preliminary experiments

During the development of the replicated AlphaZero algorithm, a wide range of experiments was conducted to check the viability of the implemented methods. First efforts were directed towards designing a working neural network and providing the necessary training data. Afterwards the focus moved to implementing, testing and verifying the Monte-Carlo Tree Search algorithm.

In the beginning, data generators and artificial neural networks were designed in parallel. While the data generators are integrated in KSquares to utilize available rule based AIs, the neural network architectures were explored using Jupyter Notebooks that are able to store notes about the design besides the actual code that instantiates and trains the networks. During development the design of the data remained relatively stable, while the neural network architecture underwent diverse changes until

it settled on a design that closely matches AlphaGo Zero's architecture but lacks the final dense layer. Early attempts at designing a neural network used a low number of layers, no skip-connections and relied on a loss function that operated on each pixel individually. Yielding poor results, the early network designs were deemed insufficient. After adapting a network architecture based on Res Blocks and a specialized output layer, the evaluation results show that the networks are able to approximately capture the playing strength inherent to their training data. A far more detailed description of the network architectures is available in section 6.4 where First Try, Stage One, Basic Strategy and LSTM are the working titles of the early, unsuccessful architectures. Network architectures AlphaZero version 1 to 6 only have a policy output that is supplemented by a value output in version 7 and higher. There are more AlphaZero networks that received various amounts of training. Table 3 shows the performance of selected incarnations of the examined network architectures. Each network played 1,000 games against Easy, Medium and Hard each on a 5×5 boxes board to assess their playing strength.

Model	Wins vs. Easy	Wins vs. Medium	Wins vs. Hard
FirstTry	0.2 %	0.2 %	0.1 %
BasicStrategy	57.9 %	24.3 %	5.0 %
StageOne 5x5	0.9 %	0.4 %	0.3 %
AlphaZeroV1	80.0 %	72.4 %	37.9 %
AlphaZeroV5	89.6 %	84.0 %	48.7 %
AlphaZeroV7	94.6 %	85.9 %	46.3 %

Table 3: Preliminary model evaluation results on a 5×5 boxes board

Reproducing the results in table 3 can be accomplished with the mentioned Docker Image by starting it with the parameter `PreliminaryModelEvaluation`. The win rates for the FirstTry and StageOne model are very poor, while BasicStrategy plays slightly better than Easy. Nevertheless BasicStrategy's results are inferior since it was trained on data generated with the Hard AI and should thus be able to beat Easy and Medium with a comfortable margin. By increasing the amount of training, the AlphaZero networks can reach the approximate playing strength of their paragon. As the results indicate, AlphaZero version 1 received the least amount of training, while version 5 and 7 saw increasingly more examples.

After finding a viable network architecture, the Monte-Carlo Tree Search algorithm was implemented. To verify that the algorithm works as expected, it was repeatedly run on board configurations that were devised by Elwin Berlekamp and described in his book [Ber00] about Dots and Boxes. A central test case to debug the MCTS algorithm was the board shown in figure 6 because the only winning move is to make a preemptive sacrifice. Making preemptive sacrifices is central to an expert strategy in Dots and Boxes. Furthermore it is not taught to the neural network, because the Hard

AI, which is used to generate training data, does not know how to make preemptive sacrifices. Thus the referenced board served as a representative example for the ability of the MCTS algorithm to improve the policy provided by the neural network. Due to the example's small board size, it was possible to manually check the values in the generated Monte-Carlo tree for errors.

In a first attempt to assess the network's performance on different board sizes, the neural network AlphaZero version 7, which was trained on boards with a maximum size of 5 × 4 boxes, is evaluated on increasingly larger board sizes. Beginning with 5 × 5 boxes boards, it plays 1,000 games each against the Easy, Medium and Hard AIs on ever larger quadratic boards up to 15 × 15 boxes.

Model	Board size	Wins vs. Easy	Wins vs. Medium	Wins vs. Hard
AlphaZeroV7	5x5	95.2 %	86.2 %	46.7 %
AlphaZeroV7	6x6	94.3 %	68.1 %	32.9 %
AlphaZeroV7	7x7	92.3 %	51.2 %	19.4 %
AlphaZeroV7	8x8	88.1 %	33.9 %	9.6 %
AlphaZeroV7	9x9	84.8 %	24.5 %	5.2 %
AlphaZeroV7	10x10	80.8 %	13.4 %	1.5 %
AlphaZeroV7	11x11	73.9 %	8.3 %	0.5 %
AlphaZeroV7	12x12	67.1 %	3.8 %	0.2 %
AlphaZeroV7	13x13	62.3 %	1.8 %	0.9 %
AlphaZeroV7	14x14	49.2 %	0.9 %	0 %
AlphaZeroV7	15x15	43.0 %	0.3 %	0 %

Table 4: Results of AlphaZero version 7 playing on increasingly larger quadratic Dots and Boxes boards

Launching the Docker Image with the parameter `BoardPerformance` will reproduce the listed results.

7.5. Berlekamp's tests

In his book about Dots and Boxes [Beroo], Elwyn Berlekamp has provided a wide range of solved and unsolved boards. To develop and debug the Monte-Carlo Tree Search algorithm, 13 boards from chapter 3 have been used. Table 5 shows the selected tests, including the correct solutions as dashed lines. Some tests have more than one correct solution – in these cases each dashed line is a valid solution to the posed problem.

Berlekamp's tests require advanced strategies to be solved correctly. For example already the third test needs a Preemptive Sacrifice to be solved. Consequently they

provide a useful basis to verify the correct operation of Monte-Carlo Tree Search and an option to assess the abilities of the raw neural network output.

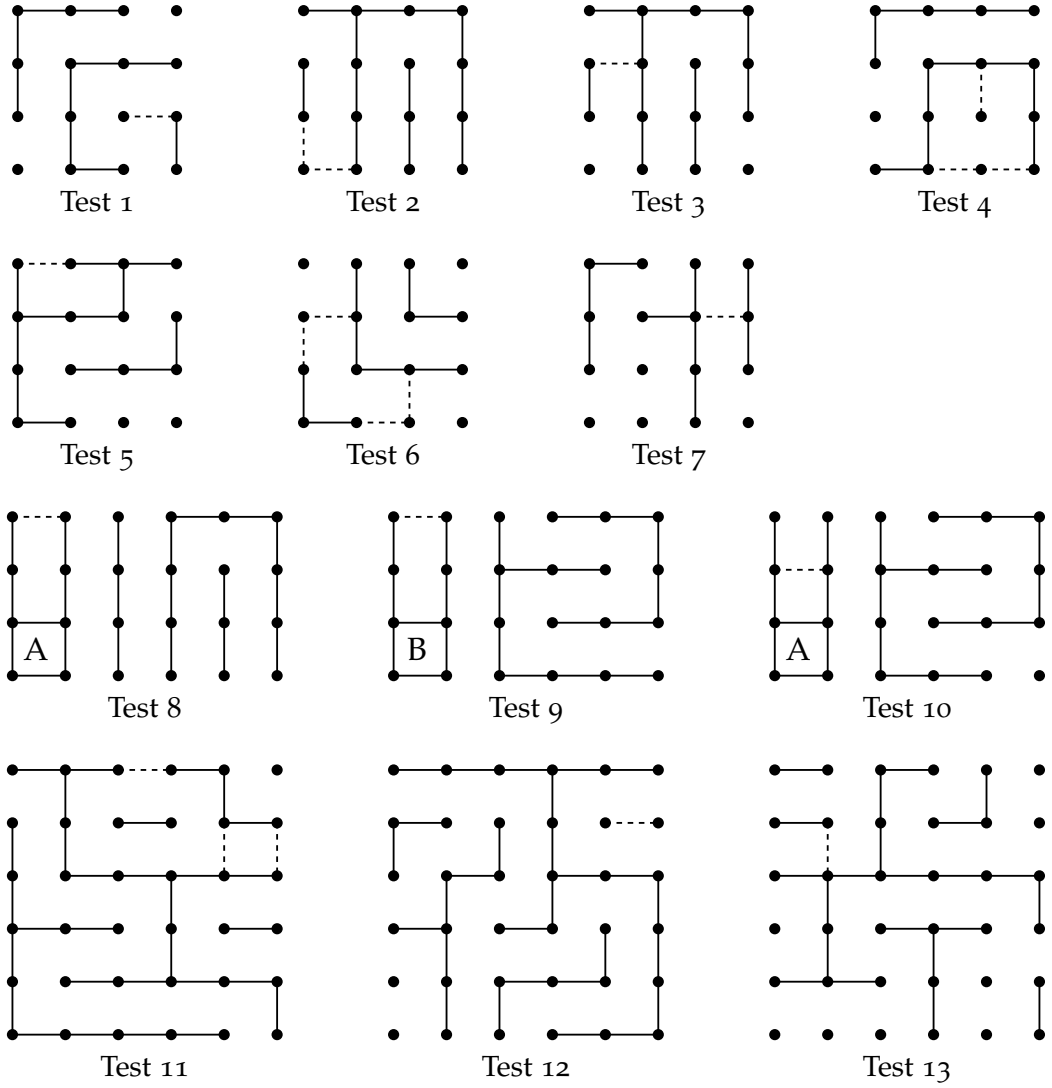


Table 5: Selected test boards from Elwyn Berlekamp's book [Beroo] with solutions marked as dashed lines

The following table shows the ability to solve Berlekamp's tests of the *AlphaZeroV7* network and of one network that was created in the *Competition* experiment described in section 7.8. Both networks are tested for their raw performance and for their abilities when combined with Monte-Carlo Tree Search. In the table a passed test is marked with an S for successful, while a failed test is marked with F. The results show that Monte-Carlo Tree search solves more tests correctly than the neural networks in direct play. Furthermore they show that MCTS is sensitive to the abilities of the underlying

neural network.

AI	Test 1	2	3	4	5	6	7	8	9	10	11	12	13
AlphaZeroV7	F	S	F	F	F	S	F	S	S	S	S	F	F
AlphaZeroV7-MCTS	S	S	S	S	S	S	S	S	S	S	S	S	F
AlphaZero-Comp-SP	F	S	F	F	S	S	F	S	S	F	S	F	F
AlphaZero-Comp-SP-MCTS	S	S	S	S	S	S	F	F	S	S	S	S	F

Table 6: Test results on Berlekamp's tests for neural networks in direct play and in combination with MCTS

Running the Docker Image with the argument `Berlekamp` will reproduce the results.

7.6. Performance on Large Boards experiment

Leaning on the preliminary experiment that evaluated AlphaZero version 7 on increasingly larger boards, this experiment first trains two networks and then evaluates them on various board sizes. There are two networks in this experiment that share the same AlphaZero architecture – both are made of 8 res-blocks and have convolutional layers with 256 filters and 3×3 kernels. Network A is trained on 2,000,000 samples of 10×10 boxes boards, while network B is trained on 2,000,000 samples of 5×4 boxes boards.All boards were generated with the Stage Four generator using the Hard AI. A major difference between A and B is the time it takes to generate the training data. For quadratic boards with a side length of n boxes, the number of lines L can be calculated as follows:

$$L = 2n^2 + 2n$$

The number of lines increases exponentially relative to n and as a result, it takes far longer on average to generate a sample of a 10×10 boxes board than it takes for a 5×4 boxes board. Generating a single 5×4 boxes sample took 0.6 milliseconds on average, while it took 28 milliseconds to generate a 10×10 boxes sample. Consequently it takes 20 minutes to generate 2,000,000 million 5×4 samples and more than 15 hours to generate the same number of 10×10 samples. After training, both networks are evaluated in direct play on quadratic boards ranging from 3×3 boxes to 15×15 boxes. Table 7 shows the results.

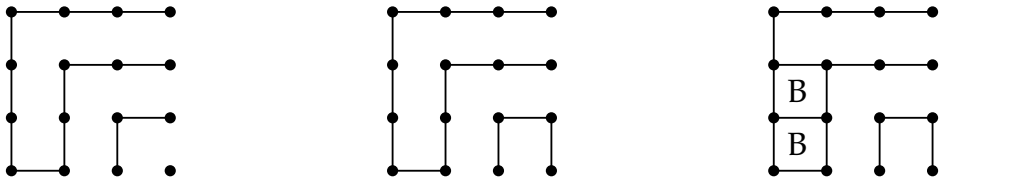
Model	Board	Wins vs. Easy	Wins vs. Medium	Wins vs. Hard
AlphaZero-A-10x10	3×3	36.3 %	28.3 %	27.7 %
AlphaZero-A-10x10	4×4	34.3 %	32.2 %	27.5 %
AlphaZero-A-10x10	5×5	40.5 %	44.0 %	32.1 %

Model	Board	Wins vs. Easy	Wins vs. Medium	Wins vs. Hard
AlphaZero-A-10x10	6×6	31.4 %	46.0 %	26.3 %
AlphaZero-A-10x10	7×7	32.3 %	44.6 %	27.4 %
AlphaZero-A-10x10	8×8	23.8 %	46.6 %	18.9 %
AlphaZero-A-10x10	9×9	23.0 %	47.6 %	19.5 %
AlphaZero-A-10x10	10×10	17.8 %	45.7 %	15.2 %
AlphaZero-A-10x10	11×11	17.7 %	48.3 %	15.0 %
AlphaZero-A-10x10	12×12	12.6 %	48.6 %	9.3 %
AlphaZero-A-10x10	13×13	11.0 %	45.5 %	8.2 %
AlphaZero-A-10x10	14×14	8.7 %	45.2 %	4.8 %
AlphaZero-A-10x10	15×15	7.6 %	46.4 %	4.4 %
AlphaZero-B-5x4	3×3	37.0 %	29.1 %	28.3 %
AlphaZero-B-5x4	4×4	52.0 %	49.1 %	31.9 %
AlphaZero-B-5x4	5×5	61.5 %	63.7 %	38.9 %
AlphaZero-B-5x4	6×6	57.8 %	63.0 %	30.5 %
AlphaZero-B-5x4	7×7	57.9 %	63.0 %	27.6 %
AlphaZero-B-5x4	8×8	59.9 %	59.2 %	22.8 %
AlphaZero-B-5x4	9×9	56.8 %	60.6 %	16.8 %
AlphaZero-B-5x4	10×10	61.1 %	60.5 %	12.1 %
AlphaZero-B-5x4	11×11	55.7 %	57.3 %	9.5 %
AlphaZero-B-5x4	12×12	51.5 %	57.3 %	6.6 %
AlphaZero-B-5x4	13×13	59.8 %	55.4 %	4.7 %
AlphaZero-B-5x4	14×14	55.9 %	57.1 %	3.7 %
AlphaZero-B-5x4	15×15	55.7 %	52.6 %	2.9 %

Table 7: Large Board experiment (version 2018-10-18) direct play results for 1,000 games each against the Easy, Medium and Hard AI on various board sizes

The results in table 7 show that version *A*, which was trained on 10×10 boards, performs significantly worse than version *B*, which was trained on 5×4 boxes boards. As a matter of fact, version *A* shows a curious inconsistency in playing strength – it loses against the Easy AI more often than against the Medium AI. This is unexpected since the Easy AI offers more opportunities to win the game than the Medium AI, hence it should be easier to defeat than the Medium AI. A possible reason for the weakness against the Easy AI is that games with it produce situations that stray too far from optimal play. The Easy AI opens random chains and thus can create board configurations where a long chain can be captured while there still are short chains available. Since a neural network does not understand the game but instead applies rules it gathered from large amounts of training data, it is confused by such situations and might not fully capture a long chain and open a short chain instead. To verify the "confusion theory", a simple test case was designed that tries to reproduce the described erroneous behaviour. Figure 25 shows the test board and the reaction of the *AlphaZe-*

roV14 network and the *AlphaZero-Competition-2018-10-28-23-44-19-SP* network. Both networks fail to make the right call and fully capture the long chain. *AlphaZeroV14* directly opens the short chain, while *AlphaZero-Competition-2018-10-28-23-44-19-SP* first captures two boxes of the long chain before it opens the short chain. The behavior of both networks confirms the confusion theory and consequently provides a reason for the deficiencies against the Easy AI.



Input board | AlphaZeroV14 | AlphaZero-Competition-2018-10-28-23-44-19-SP

Figure 25: Board for the "confusion theory" test case and the moves of two neural networks

Considering the results of network *B* in table 7, it is clear that *B* has captured key strategies of the game as it is able to consistently beat the Easy and Medium AI even on very large boards in more than half of the games. An apparent reason for the success of network *B* is that it was trained on small boards and was thus able to see many essential configurations and consequently derive general strategies, while network *A* was not able to focus on the important part of the game. Besides the powerful playing strength, network *B* also beats the training time by an order of magnitude.

To verify the results, the experiment was executed a second time with training on slightly diverging board sizes. Network *C* was trained on 2,000,000 samples of 8 × 8 boxes boards while network *D* was trained on 2,000,000 samples of 5 × 5 boxes boards. Table 8 shows the results of network *C* and *D*. Depending on the board's size it is possible that a game ends in a draw, which is not counted towards the win rate.

Model	Board	Wins vs. Easy	Wins vs. Medium	Wins vs. Hard
AlphaZero-C-8x8	3 × 3	36.6 %	25.9 %	32.2 %
AlphaZero-C-8x8	4 × 4	34.0 %	33.8 %	26.3 %
AlphaZero-C-8x8	5 × 5	47.1 %	43.2 %	30.3 %
AlphaZero-C-8x8	6 × 6	42.8 %	42.2 %	25.5 %
AlphaZero-C-8x8	7 × 7	48.1 %	47.5 %	24.0 %
AlphaZero-C-8x8	8 × 8	42.4 %	45.5 %	19.1 %
AlphaZero-C-8x8	9 × 9	42.7 %	50.6 %	16.5 %
AlphaZero-C-8x8	10 × 10	39.7 %	46.2 %	13.9 %
AlphaZero-D-5x5	3 × 3	41.6 %	29.4 %	31.8 %

Model	Board	Wins vs. Easy	Wins vs. Medium	Wins vs. Hard
AlphaZero-D-5x5	4×4	42.0 %	36.9 %	24.0 %
AlphaZero-D-5x5	5×5	56.3 %	55.7 %	34.7 %
AlphaZero-D-5x5	6×6	54.7 %	54.6 %	27.9 %
AlphaZero-D-5x5	7×7	49.8 %	56.2 %	24.3 %
AlphaZero-D-5x5	8×8	50.1 %	51.0 %	20.6 %
AlphaZero-D-5x5	9×9	52.3 %	56.2 %	18.3 %
AlphaZero-D-5x5	10×10	49.3 %	53.9 %	14.0 %

Table 8: Large Board experiment (version 2018-10-20) direct play results for 1,000 games each against the Easy, Medium and Hard AI on various quadratic board sizes

As network *C* and *D* were trained on boards that differ less in size than those of network *A* and *B*, the results of *C* and *D* are closer together. Nevertheless the trends that were observed in table 7 are also visible in table 8 but to a lesser extent. Network *C*, which was trained on training data made of large boards, still wins less often against the Easy AI than against the Medium AI on certain board sizes. Furthermore network *D* has a consistently higher rate of success.

This experiment can be reproduced by running the mentioned Docker Image with the command line argument `LargeBoards`. The results will be available in the associated Docker Volumes.

7.7. New Strategies experiment

This experiment creates two fully convolutional AlphaZero networks as described in section 6.4.3, which are each made of 8 res-blocks with convolutional layers that have 256 filters and 3×3 kernels. In the beginning the first network is trained on data that was generated by the Medium AI. This means that the network does not know about Double Dealing as it is only trained on data that fully captures chains. Afterwards the network's weights are copied to the second network so that the second network can be further trained in self-play, while the first network retains its original state that was only trained on rule-based data. Finally an evaluation checks if the network learned Double Dealing in self-play reinforcement learning.

For the first part, a newly created network is trained on data that was generated with the StageFour dataset generator using the Medium AI. The network is trained on 4,000,000 samples of 3×3 boxes boards. To make sure that the network does not know about Double Dealing, the experiment activates a specially crafted analysis module that detects instances of Double Dealing. After the training is done, the experiment executes a short evaluation to verify that the network does not apply Double Dealing.

Table 9 shows that the network plays on the level of the Medium AI and that not a single instance of Double Dealing was detected.

Model	Wins vs. Easy	Wins vs. Medium	Wins vs. Hard	Double Dealing
AlphaZero-NewStrat	75.3 %	50.5 %	20.4 %	0

Table 9: *New Strategies* experiment (version 2018-10-29) direct play results for 1,000 games each against the Easy, Medium and Hard AI on a 3×3 boxes board

For the second part of this experiment, the initialized network is trained in self-play on 3×3 boxes boards. Coming from a state where it has not seen Double Dealing, it is used to support Monte-Carlo Tree Search in finding strong moves. In this experiment, Monte-Carlo Tree Search is configured to operate on a per-line basis – using move sequences is disabled so MCTS does not have knowledge of the strategies that are encoded in the move sequences. Disabling move sequences is important, because the move sequence generator knows about Double Dealing and one of its prominent features is to generate chain capturing sequences that end in Double Dealing. Monte-Carlo Tree Search is run for 1,000 iterations using the currently best network to generate a single sample. For one iteration of self-play, the experiment generates 512 samples with MCTS, which are then used to train the network in 5 epochs using data augmentation as described in section 6.3.3. After training, the self-play iteration is concluded with an evaluation where the best network is evaluated against the newly trained one in direct play (i.e. without MCTS) in 1,000 games. If the contending network won at least 500 games, it becomes the new best network. If the contending network did not win, it will be used as the starting point for training in the next iteration but not to generate data. This self-play process is repeated for 20 iterations.

After self-play, the resulting network is evaluated against the Easy, Medium and Hard AI on 3×3, 5×5 and 15×15 boxes boards. Table 10 shows the results of both networks in direct play for 10,000 games each against the Easy, Medium and Hard AI on 3×3, 5×5 and 15×15 boxes boards. Especially the result of direct play against the Hard AI on the 3×3 boxes board shows a significant improvement with the win rate increasing from 21.17 % for the initial network to 46.21 % for the network that was trained in self-play. Analysis of Double Dealing shows that the network has learned to apply the strategy by using self-play reinforcement learning. Furthermore it is evident that the network heavily forfeits playing strength on a larger board. The most likely explanation for this is that larger boards offer exponentially more chances to make a wrong move that can be exploited by the rule based AIs. This might also explain the 76 instances of Double Dealing on the 5×5 board and the 779 instances of Double Dealing on 15×15 board – the network simply applied Double Dealing in error because larger boards offer more chances to make mistakes.

Model	Board	Wins vs. Easy	Wins vs. Medium	Wins vs. Hard	Double Dealing
AlphaZero-NewStrat	3×3	75.89 %	50.59 %	21.17 %	0
AlphaZero-NewStrat-SP	3×3	76.54 %	72.62 %	47.21 %	7,853
AlphaZero-NewStrat	5×5	82.12 %	27.71 %	5.17 %	76
AlphaZero-NewStrat-SP	5×5	71.33 %	37.52 %	8.45 %	6,908
AlphaZero-NewStrat	15×15	9.76 %	0 %	0 %	779
AlphaZero-NewStrat-SP	15×15	5.96 %	0.02 %	0 %	4,669

Table 10: *New Strategies* experiment (version 2018-10-29) direct play results for 10,000 games each against the Easy, Medium and Hard AI on 3×3, 5×5 and 15×15 boxes board

This experiment can be reproduced by running the mentioned Docker Image with the command line argument `NewStrategies`. The results will be available in the associated Docker Volumes.

7.8. Competition experiment

This experiment tries to train a neural network that plays Dots and Boxes on a board with 5×5 boxes as well as possible. First, it creates a fully convolutional *AlphaZero* network from scratch.Section 6.4.3 describes the design details of the network architecture. Specifically the network has 8 res-blocks featuring 256 filter maps in each convolutional layer with a kernel size of 3×3.

During the first part of the experiment, the initial network is trained for one epoch on 4,000,000 examples of 5×4 boards that were generated with the StageFour dataset generator using the Hard AI. Afterwards, the resulting network is copied to create a second network which acts as the starting point for self-play training.

For one iteration, the self-play mode starts to generate 512 samples using Monte-Carlo Tree Search with the best neural network. Using the newly generated samples, the newest network is trained cumulatively on the augmented data for five epochs, thus creating a contending network. The best network plays 1,000 games against the contending network in direct play. If the contending network wins at least half of the games, it becomes the new best network. This process is repeated for 32 iterations to create *AlphaZero-Comp-SP*.

For a second run of self-play training with different parameters, the initial network is copied again to create another starting point for self-play training. In this round, the training does not use move sequences and only runs Monte-Carlo Tree Search with 1,000 iterations instead of the default number of 1,500 iterations. Self-play is run for 32 iterations, creating *AlphaZero-Comp-SPv2*.

Finally, both networks trained in self-play and the initial network, that was only trained on Hard AI data are evaluated against the Hard AI, the Alpha Beta AI ($\alpha\beta$) from the Bachelor's thesis, Dabble and QDab, which is the best AI for Dots and Boxes according to [Pet15]. All AIs were allowed a maximum time of 10 seconds per move. Table 11 shows the results of the evaluation. They show that the AlphaZero-MCTS AIs are inferior to the other search based algorithms. With the setup of this experiment, AlphaDots does not reach state of the art performance.

Model	Wins vs. Easy	Medium	Hard	$\alpha\beta$	Dabble	QDab
AlphaZero-Comp	69 %	68 %	39 %	18 %	0 %	0 %
AlphaZero-Comp-MCTS	95 %	83 %	66 %	10 %	0 %	0 %
AlphaZero-Comp-SP	50 %	60 %	31 %	12 %	0 %	0 %
AlphaZero-Comp-SP-MCTS	94 %	89 %	58 %	9 %	0 %	0 %
AlphaZero-Comp-SPv2	36 %	58 %	33 %	12 %	0 %	0 %
AlphaZero-Comp-SPv2-MCTS	95 %	85 %	52 %	16 %	0 %	0 %

Table 11: Competition experiment (version 2018-10-28) direct play results for 100 games each against the Easy, Medium, Hard, Dabble and QDab AI on a 5 $\times$ 5 boxes board

8. Results

This section provides answers to the initially stated research questions on the basis of the evaluation results and other findings of this thesis.

8.1. Using less resources

The core of this thesis was to devise and evaluate methods to replicate the AlphaZero search and reinforcement learning algorithm while using far less resources. In an attempt to find out how to get tractable results with less computational power, the first step was to conceive a neural network architecture that could transfer strategies it learned on small boards to big boards. Further effort went into modifying the self-play reinforcement learning methodology to quickly yield improved playing strength. Instead of playing full games with MCTS and uniformly sampling from the generated data as is done in AlphaGo Zero, AlphaDots generates board configurations with a focus on the forming phase of Dots and Boxes and then executes MCTS on those boards. As described in section 6.2.5 the training data still covers all parts of the game, but provides more samples for the important phase of the game.

Evaluation has shown that it indeed is possible to train a neural network on small boards and have it play well on much bigger boards. Based on the results of the *Large Boards* experiment in section 7.6 it can be said that the neural network can be trained far faster on small boards while reaching superior results when compared to networks trained on large boards. It is clear that training on small boards conveys the key strategies to the neural network without distractions and thus enables it to transfer the strategies to situations with far higher complexity. A probable explanation for this success is that small boards are less complex so that the generated data covers all important situations many times, consequently providing ideal conditions for a fully convolutional neural network to grasp the crucial principles that underlie Dots and Boxes.

Reducing the required resources in self-play reinforcement learning has proven less successful. Results show that with the prescribed methods the AlphaDots algorithm does not reach state of the art performance. Although experiments show the general viability of the implemented approach, it has become clear that it needs at least an order of magnitude more computational resources to reach state of the art performance in acceptable time.

8.2. Neural network playing Dots and Boxes on varying board sizes

This research question was about how to design a neural network that is able to play well on various Dots and Boxes board sizes. In the first place it was about what network architecture enables a network to play on different boards. After successfully

implementing a working neural network, the question extends to the network's abilities to transfer strategies learned on one type of board to other types.

For the first part, a fully convolutional neural network was designed that mostly follows its paragon AlphaZero but uses custom final layers that dislodge a fixed board size. Section 6.4.3 describes the details of those final layers and Alpha Dot's network architecture in general. By modifying AlphaZero's architecture to be fully convolutional, the first part of this research question was solved successfully.

To answer the second part of how well the network is able to transfer learned strategies to different board sizes, the results of a range of experiments can be consulted. One preliminary experiment evaluated an early version of Alpha Dot's neural network on various quadratic boards, with results shown in table 4. Although the network was only trained on small boards, it is able to win more than half of the games against Easy on large boards up to 13×13 boxes. Since the network is fully convolutional, it manages to capture and apply general strategies in Dots and Boxes.

The *Large Boards* experiment in section 7.6 compares the performance of networks on various boards where the networks were trained exclusively on small or large boards. Networks that were trained on small boards capture general strategies for Dots and Boxes very well. They are able to learn strategies and apply them in new situations that vastly differ from their training data in terms of complexity. Conversely the networks that were trained on large boards struggle to grasp the required strategies.

According to the experiment's results the networks loose against the Easy AI more often than against the Medium AI. The results expose that neural networks learn by example and are not able to actually understand the logic of the strategies they apply. A simple test in figure 25 shows that the networks fail to act correctly in situations that do not occur in optimal play – since they have been trained on data generated with the Hard AI that did not show such situations, they did not know how to react appropriately. This underlines the importance of providing the neural network with a diverse range of training data that also strays from the path of optimal play to prepare the network for unexpected situations.

8.3. Finding new strategies in self-play

For this research question, the goal was to find out if new strategies emerge from self-play. Given a network that is able to play Dots and Boxes at a basic level, the task is to train the network in self-play and afterwards verify that it learned to apply a new strategy. AlphaDots' self-play reinforcement learning uses Monte-Carlo Tree Search in combination with a neural network to produce improved training data that is fed to the neural network.

A first step was to verify that Monte-Carlo Tree Search is able to find better moves than those directly provided by the supporting neural network. To perform the verification,

13 test boards from Berlekamp's book [Ber00] have been used. The test results in table 6 show that the neural networks fail to find the right move on 7 boards, especially when they have to make a preemptive sacrifice. Conversely Monte-Carlo Tree Search is able to pass almost all tests, thus demonstrating its ability to find stronger moves than the supporting neural network.

A more substantial positive answer to the posed research question is provided by the results of the *New Strategies* experiment described in section 7.7 where a network learned a new strategy in self-play that was not taught in training. In the experiment, the network was trained on data that did not contain Double Dealing moves. Using the Double Dealing analysis module in the evaluation framework, it was verified that the network does not apply Double Dealing. After it was trained in self-play on 10,240 examples for 5 epochs, the network is able to apply Double Dealing without MCTS and also exhibits significantly improved playing strength.

The results show that the AlphaDots self-play reinforcement learning algorithm is able to improve the playing strength. Nevertheless the *Competition* experiment shows that the amount of applied self-play reinforcement learning is nowhere near enough to reach state of the art performance.

8.4. Playing strength of AlphaDots

As the *Competition* experiment has shown, AlphaDots does not reach state of the art performance with the given amount of training. AlphaDots does not stand a chance against Dabble or QDab, which are two of the best AIs for Dots and Boxes.

Previous experiments and tests have shown that the algorithm works in principle. To create a version of AlphaDots that reaches a higher playing strength, it should be trained on more data that covers a diverse set of situations. During the *Competition* experiment, the neural network was trained on 16,384 samples generated by Monte-Carlo Tree Search, which is a very small amount of training data. In the *New Strategies* experiment it was shown that about 10,000 samples are enough for the network to learn Double Dealing. Practically every game of Dots and Boxes played by experts features Double Dealing and as a consequence many of the 10,000 samples train the network to apply Double Dealing. In contrast to learning Double Dealing, the moves required to learn more advanced strategies like preemptive Sacrifices can not be generated as easily. A manual inspection of 512 generated samples has uncovered that there are almost no samples that teach the network to make Preemptive Sacrifices.

With more computational resources available, it would be interesting to follow the path of AlphaZero where the neural network is trained from scratch in self-play. As a result of providing no rule based training data, the data generated by self-play should cover a very broad range of possible situations and therefore prepare the network well for every situation. Although this procedure requires a large amount of computational resources, it will most probably lead to state of the art performance.

9. Conclusion

To conclude this thesis, a short summary presents the major results of this work. Ultimately a section about possible future work presents aspects that have not been covered yet.

9.1. Summary

This thesis set out to find ways to replicate the success of AlphaGo Zero to the game of Dots and Boxes while using far less resources. A central part that permeates all design decisions that were made during realization of the replication process was to require as few resources as possible by various means. First there is the change of the network architecture to be fully convolutional by adding custom layers that enable operation on any board size. Moreover there is a novel way to generate training data – using a non-uniform distribution to direct the focus on the important part of the game, the generator produces a random board state on which Monte-Carlo Tree Search is executed. Utilizing the fully convolutional architecture of the neural network enables a training paradigm that operates on small boards and efficiently teaches the network important strategical aspects of Dots and Boxes, thus saving resources when generating data and training the network.

By using self-play reinforcement learning, AlphaDots can improve itself. It was shown that it is able to find new strategies that it has not seen before and consequently reach a higher level of expertise in Dots and Boxes. A final evaluation against QDab – the best AI for Dots and Boxes – has shown that AlphaDots does not reach state of the art performance in Dots and Boxes with the given amount of training in self-play. With more resources for self-play training it should be possible to reach state of the art performance.

9.2. Future work

The method of learning to play a game on small boards and transferring the learned strategies to bigger boards could be applied to other games where it is possible to play on various board sizes. Most prominently, the presented method can be applied to Go, because it is possible to play the game on boards of arbitrary size. Training a fully convolutional neural network on small Go instances will probably yield shortened training time due to the heavily reduced complexity of the data.

It would be interesting to find out if the weights that were learned by AlphaDots match the hand crafted features used by QDab. In image processing applications, deep neural networks often learn filters that match manually defined kernels of convolutional image operations like the Sobel Operator. [DH73] A viable approach could be to try to translate the learned kernels of AlphaDots into operations on chains as they were

presented in the paper about QDab. [Zhu14] If successful, it would be possible to compare the learned kernels with the manually crafted features in QDab.

As of writing this thesis, there is no proof for the complexity of Dots and Boxes. For Chess and Go is was proven in the early 1980s that it is Exponential Time Complete to find an optimal strategy for an $n \times n$ board. [FL81; Rob83] Since there is no such proof for Dots and Boxes it would be interesting to formally assess the complexity of finding an optimal strategy for an $n \times n$ board.

9.3. Acknowledgments

Thank you to those who have supported me in creating this thesis – Hans Meine, Jannis Stoppe and Rolf Drechsler for interesting and helpful conversations. Further thanks to Hans Meine for swiftly executing experiments on better hardware. Concluding this thesis, I want to express gratitude to my love, Manuela Illig, for providing steady support and insight.

Appendix

A. References

[BC14] K. Buzzard and M. Ciere. "Playing simple loony dots and boxes endgames optimally". In: INTEGERS 14 (2014).

[BCG03] Elwyn R. Berlekamp, John H. Conway, and Richard K. Guy. Winning Ways for Your Mathematical Plays. 2nd ed. Vol. 3. Natick, Massachusetts: A. K. Peters, Ltd., 2003. ISBN: 1568811438.

[Ber00] Elwyn R. Berlekamp. The Dots and Boxes Game. Natick, Massachusetts: A. K. Peters, Ltd., 2000. ISBN: 1568811292.

[BK12] Joseph K. Barker and Richard E. Korf. "Solving Dots-And-Boxes." In: Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence. 2012, pp. 414–419.

[Cha10] Guillaume Chaslot. "Monte-carlo tree search". PhD thesis. Maastricht University, 2010.

[Cho+15] François Chollet et al. Keras. `https://keras.io`. 2015.

[DH73] Richard O. Duda and Peter E. Hart. "Pattern classification and scene analysis". In: A Wiley-Interscience Publication (1973).

[FL81] Aviezri S. Fraenkel and David Lichtenstein. "Computing a perfect strategy for n × n chess requires time exponential in n". In: International Colloquium on Automata, Languages, and Programming. Springer. 1981, pp. 278–293.

[GBB11] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. "Deep sparse rectifier neural networks". In: Proceedings of the fourteenth international conference on artificial intelligence and statistics. 2011, pp. 315–323.

[Gel+12] Sylvain Gelly et al. "The grand challenge of computer Go: Monte Carlo tree search and extensions". In: Communications of the ACM 55.3 (2012), pp. 106–113.

[Gro00] J. P. Grossman. Dabble. `http://www.mathstat.dal.ca/~jpg/dabble/`. [AI for Dots and Boxes, presented on the second Combinatorial Game Theory Research Workshop, at the Mathematical Sciences Research Institute, 24-28 July 2000.] 2000.

[He+16] Kaiming He et al. "Deep residual learning for image recognition". In: Proceedings of the IEEE conference on computer vision and pattern recognition. 2016, pp. 770–778.

[HW62] David H. Hubel and Torsten N. Wiesel. "Receptive fields, binocular interaction and functional architecture in the cat's visual cortex". In: The Journal of physiology 160.1 (1962), pp. 106–154.

[IS15] Sergey Ioffe and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift". In: arXiv preprint arXiv:1502.03167 (2015).

[IWK17] Roman Ilin, Thomas Watson, and Robert Kozma. "Abstraction hierarchy in deep learning neural networks". In: International Joint Conference on Neural Networks Neural Networks (IJCNN). IEEE. 2017, pp. 768–774.

[Jou+17] Norman P. Jouppi et al. "In-datacenter performance analysis of a tensor processing unit". In: ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA). IEEE. 2017, pp. 1–12.

[KH92] Anders Krogh and John A. Hertz. "A simple weight decay can improve generalization". In: Advances in neural information processing systems. 1992, pp. 950–957.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks". In: Advances in neural information processing systems. 2012, pp. 1097–1105.

[LBH15] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning". In: nature 521.7553 (2015), p. 436.

[Mar+15] Martín Abadi et al. "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems". Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[Oli06] Travis E. Oliphant. A guide to NumPy. USA: Trelgol Publishing. 2006.

[Pet15] Tom V. Peters. "Implementierung und Vergleich von Computergegnern für Käsekästchen". B. Sc. Thesis. University Bremen, 2015.

[Rob83] John M. Robson. "The complexity of Go". In: Proceedings of the 9th World Computer Congress on Information Processing. 1983, pp. 413–417.

[Ros11] Christopher D. Rosin. "Multi-armed bandits with episode context". In: Annals of Mathematics and Artificial Intelligence 61.3 (2011), pp. 203–230.

[SB+98] Richard S. Sutton, Andrew G. Barto, et al. Reinforcement learning: An introduction. MIT press, 1998.

[Sch15] Jürgen Schmidhuber. "Deep Learning in Neural Networks: An Overview". In: Neural Networks 61 (2015). Published online 2014; based on TR arXiv:1404.7828 [cs.NE], pp. 85–117. DOI: 10.1016/j.neunet.2014.09.003.

[Sch89] Jonathan Schaeffer. "The history heuristic and alpha-beta search enhancements in practice". In: IEEE transactions on pattern analysis and machine intelligence 11.11 (1989), pp. 1203–1212.

[Shi+15] Xingjian Shi et al. "Convolutional LSTM Network: A Machine Learning Approach for Precipitation Nowcasting". In: ArXiv e-prints (June 2015). arXiv: 1506.04214 [cs.CV].

[Sil+16] David Silver et al. “Mastering the game of Go with deep neural networks and tree search”. In: Nature 529.7587 (2016), pp. 484–489.

[Sil+17a] David Silver et al. “Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm”. In: arXiv preprint arXiv:1712.01815 (2017).

[Sil+17b] David Silver et al. “Mastering the game of Go without human knowledge”. In: Nature 550.7676 (2017), p. 354.

[WB98] Lex Weaver and Terry Bossomaier. “Evolution of neural networks to play the game of dots-and-boxes”. In: arXiv preprint cs/9809111 (1998).

[WR17] Haohan Wang and Bhiksha Raj. “On the origin of deep learning”. In: arXiv preprint arXiv:1702.07800 (2017).

[Zhu14] Yimeng Zhuang. QDab. `http://dotsandboxes.tar.xyz/`. Opponent for Dots and Boxes based on Monte-Carlo Tree Search and a feed forward neural network. 2014.

B. List of Figures

C. List of Tables