AlphaZero: Mastering the game of Dots and Boxes without human knowledge

Johannes Scherer
johannes.scherer@uni-ulm.de
Project Deep Reinforcement Learning
Universität Ulm, WiSe 22/23

ABSTRACT

The introduction of AlphaZero in 2017 was a milestone in the field of game-playing artificial intelligence. Until then, development of the strongest programs was based on game-specific search techniques, adaptations, and handcrafted evaluations created by human experts. In contrast, AlphaZero learns and masters board games by reinforcement learning from self-play without human guidance beyond game rules, reaching superhuman performance for complex board games such as chess, shogi and Go. In this work, we apply the AlphaZero algorithm to the game of Dots and Boxes. In this context, we analyze the training process of AlphaZero and evaluate its performance against other artificial intelligence based game-playing programs for small board sizes. Further, we discuss the challenges and requirements involved in successfully applying AlphaZero to other board games. While showing its forward-looking capabilities, AlphaZero consistently beats its opponents in our experiments. The implementation code is publicly available at https://github.com/josch14/alpha-zero-dots-and-boxes.

1 INTRODUCTION

The idea of artificial intelligence (AI) playing games is as old as the electronic computer itself. Most early research on game AI was focused on classic board games such as checkers and chess due to the great complexity they offer despite their limited rules [14]. In 1949, Claude Shannon worked on the theoretical idea of constructing a computing routine or program in order to enable a modern computer to play chess [16]. Already in the 1960s, Arthur Samuel developed a checkers program that could improve its gameplay over time by only giving it the rules of the game, a sense of direction, and a redundant and incomplete list of parameters [15]. Another milestone in game-playing AI was the introduction of the Deep Blue chess computer by IBM [7]. In 1997, Deep Blue defeated the then-reigning world chess champion Garry Kasparov in a six-game match by using a sophisticated search algorithm, complex evaluation function, a database of pre-analyzed positions and specialized hardware.

Advances in Deep Learning in recent years have opened up new approaches to game-playing AI. With the introduction of AlphaGo Zero in 2017, Google DeepMind proposed a novel framework which is able to achieve superhuman performance in Go by representing Go knowledge using a deep convolutional neural network, trained solely by reinforcement learning from games of self-play without relying on human data or domain knowledge except the rules of the game [19]. This framework was generalized and applied to the games of chess, shogi and Go by the follow-up named AlphaZero, demonstrating that superhuman performance across many challenging domains can be achieved with a general-purpose reinforcement learning algorithm [18].

While AlphaZero has achieved impressive results, ongoing research is looking for ways to reproduce and improve the results for other games or tasks [23], while also addressing the challenges and limitations associated with the algorithm. For example, AlphaZero has high computational requirements. During training of AlphaZero only, 5000 tensor processing units (TPUs) were used to generate self-play games, and 16 TPUs were used to train its neural networks. Training lasted for approximately 9 hours in chess, 12 hours in shogi, and 13 days in Go [18]. Furthermore, during training AlphaZero may become trapped in local optima and fail to explore new strategies [8]. Finally, AlphaZero contains many hyperparameters, which makes hyperparameter optimization difficult in addition to the already high computational effort required for training.

With this work, we aim to reproduce AlphaZero's impressive results achieved for chess, shogi and Go for the game Dots and Boxes. We did this by implementing the AlphaZero algorithm and the game logic of Dots and Boxes from scratch, and training AlphaZero and evaluating its performance for Dots and Boxes games of different sizes. The game and its rules are introduced in the following section. Afterward, we explain the AlphaZero algorithm in detail and how we adapted it in order to learn and play Dots and Boxes (Sec. 3). After introducing the baseline game-playing AIs and selected hyperparameters (Sec. 4), we show the experimental results (Sec. 5) and discuss those with respect to the mentioned challenges when employing AlphaZero as game-playing AI (Sec. 6).

2 RELATED WORK

2.1 Dots and Boxes

Dots and Boxes is a simple pen-and-paper game for two players. The game starts with an empty $m \times n$ grid of dots. The two players alternately take turns by drawing (i.e., connecting) a single horizontal or vertical line between two unjoined adjacent dots. When a player completes the fourth side of a 1×1 box by drawing its remaining free line, the player claims the box, earns a point by doing so and has to take another turn. The game ends when no more lines can be drawn (or, when desired, if a player can no longer catch up with the opponent's points lead), with the winner being the player who claims the most boxes. Dots and Boxes instances with an even number of boxes may end in a draw. Figure 1 shows an example game of Dots and Boxes on a 2×2 square board (i.e., with a total of 4 boxes to be claimed). Here, the lines are colored only for the purpose of making clear which player drew the last line. Note that for a given Dots and Boxes position, it has no relevance which lines were drawn by which player, since for claiming a box a player only has to draw its remaining free line.

Dots and Boxes can be played on boards of arbitrary size. In research, Dots and Boxes has received a considerable amount of attention discussing its mathematical properties, complexity properties, and winning strategies in particular positions [5, 6]. For example, Barker and Korf [4] determined that Dots and Boxes

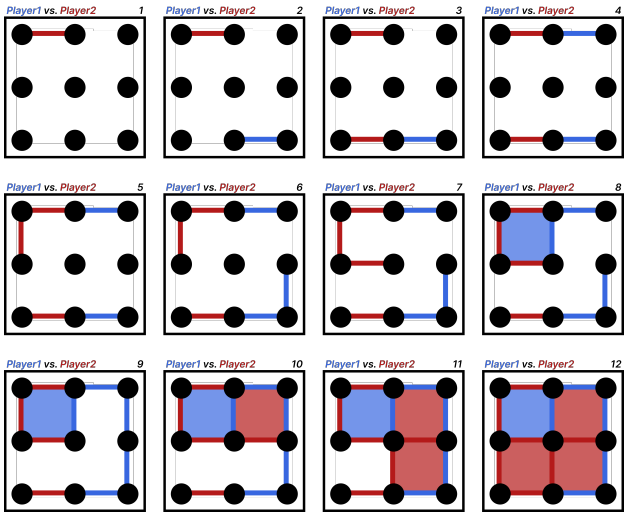

Figure 1: Example game of Dots and Boxes on a 2×2 square board, starting with the first move at the top left, and ending with the last move at the bottom right. Red wins the match with 3:1 points against Blue.

on a board of 4×5 boxes is a tie given optimal play. Their solver employs an alpha-Beta search, while applying both problem-specific and general techniques to reduce the search space to a manageable size.

Dots and Boxes is a simple game only on the surface. Despite its simple rules, its gameplay and properties offer high complexity especially for increasing board sizes [6]. In combinatorial game theory a game is seen as *impartial* if it meets two criteria: 1) the set of moves available to a player depend only on the game's position and not on which of the two players is currently moving, and 2) any particular move has equal value for both players. Furthermore, the *normal play convention* of an impartial game is that the player making the last move is the game winner. Dots and Boxes is unusual (as opposed to, e.g., tic-tac-toe) in the sense of being an impartial game lacking the normal play convention, resulting in more complicated analysis offered by combinatorial game theory.

A typical game usually results in a board state that opens the possibility for a player to claim a number of boxes with their next move(s). It is important to note that a player is not obliged to claim a box whenever they have the ability to do so. Winning strategies work with the concept of sacrifice, i.e., it can be beneficial for a player to sacrifice some boxes in order to claim a higher number of boxes in the long-term. For example, during the game shown in Figure 1, Red sacrifices one box to Blue. With Blue accepting the sacrifice and having to draw another line inside the chain, Red is able to claim the remaining boxes. Chains, i.e., line structures resulting in connected boxes which are close to claim, are formed during a typical game of Dots and Boxes. In the example, after accepting the sacrifice, Blue has to draw a line inside the chain, enabling Red to claim all boxes of the chain. As described by Buchin et al. [6], moves like the double-dealing (sacrificing few boxes but passing the turn onto the opponent) and double-cross moves (move that closes two boxes at once) are essential for players that consistently win games of Dots and Boxes, and therefore play an important role in research working on Dots and Boxes.

2.2 Deep Reinforcement Learning

In Reinforcement Learning (RL) an agent learns to make decisions by interacting with an environment over time [13]. Formally speaking, given a state s_t at time t, the agent selects an action a_t. The agent makes this decision using its *policy* $\pi(a_t, s_t)$, a decision-making function mapping states s_t to actions a_t. Performing an action a_t results in receiving a scalar reward r_t, and transition to the next state s_{t+1}. The agent's goal is to maximize the cumulative reward received over time from each state. In order to measure the expected cumulative reward that can be obtained by following a particular policy, the agent makes use of *value* functions. Here, it is differentiated between two types of value functions. The *state value* function $v_\pi(s)$ estimates the expected cumulative reward for following the policy π from state s. The *action value* function $q_\pi(s, a)$ estimates the expected cumulative reward for selecting action a in state s and then following policy π. In general, RL is most centrally concerned with different methods for computing and updating the value function(s) by trial-and-error [20]. By using these basic concepts, RL algorithms are able to solve a wide range of problems, from playing games to controlling robots.

Advances in Deep Learning have also accelerated progress in RL, with the use of deep learning algorithms within RL defining the field of Deep Reinforcement Learning (DRL). Here, neural networks are used to approximate the optimal policy or value function, enabling RL to scale to decision-making problems which were previously intractable due to high-dimensional state and action spaces. This is also the case for AlphaGo Zero and AlphaZero, with both being DRL systems. By enriching the RL approach with deep neural networks for policy and value estimation in combination with a heuristic search algorithm, AlphaGo Zero and AlphaZero are able to outperform game-playing AI approaches with handcrafted heuristics and to defeat the world's best players in chess and Go [3].

2.3 Alpha-beta Search

The minimax search is a traditional algorithm that has been used for many years in game-playing programs [1]. Given a game position, the minimax search recursively determines the optimal move (in the sense of minimizing the maximum possible loss) for a player by evaluating the game tree of all possible moves, limited to a given search depth. The evaluation of a board position requires the definition of a game-specific handcrafted evaluation function, which returns higher values for board positions that more likely result in a positive game outcome.

Alpha–beta pruning improves the efficiency of tree search algorithms by removing unnecessary branches. For this, a depth-first search is performed while maintaining two values, alpha and beta, that represent the best possible scores that the maximizing and minimizing players can achieve, respectively. By maintaining alpha and beta, subtrees may be pruned without having to be completely explored when nodes fall outside the range defined by alpha and beta. Barker and Korf [4], e.g., used the minimax algorithm with alpha-beta pruning for showing that Dots and Boxes on a board of 4×5 boxes is a tie given optimal play.

While alpha-beta search works with improved efficiency, tree search algorithms in general are still limited by the fact that they rely on human-designed and game-specific heuristics. On the other side, game-playing DRL approaches are able to search the game tree more efficiently and learn playing a game from scratch without human knowledge, and, by doing so, are capable of developing innovative and unconventional playing styles that

may not be discovered by human-designed heuristics. This is the one of the key reasons why DRL approaches such as AlphaZero are so popular.

3 MASTERING THE GAME OF DOTS AND BOXES WITH ALPHAZERO

3.1 AlphaZero

AlphaZero [18] is a more generic version of the AlphaGo Zero [19] algorithm. Traditional game-playing programs, such as the alpha-beta search, require human-designed and game-specific heuristics (see Sec. 2.3). In contrast, with AlphaZero being an DRL algorithm, it removes the dependence on any human knowledge by enriching the RL approach with deep neural networks for policy and value estimation in combination with a heuristic search algorithm. In the following, we provide a high-level overview of the AlphaZero algorithm.

Instead of a game-specific handcrafted evaluation function and move ordering heuristics (as required, e.g., for alpha-beta search), AlphaZero uses a **deep neural network** $f_\theta(s) = (p, v)$ with parameters θ. The neural network takes a board position s as input, and outputs both *move probabilities* and a *value*. The vector of move probabilities p consists of probabilities $p_a = Pr(a|s)$ of selecting a move a given the board position s. The scalar value v is the neural network's probability estimation of the current player winning from the given board position, i.e., $v \approx E[z|s]$. The use of a single neural network, rather than separate policy and value networks, was a crucial change leading to the success of AlphaZero. We introduce the deep neural network architecture that was used for AlphaZero in detail later (see Sec. 3.2).

Instead of an alpha-beta search with domain-specific enhancements, AlphaZero uses a general purpose **Monte-Carlo tree search** (MCTS) algorithm. Given a board position s, the MCTS outputs *search probabilities* π of playing each move. Each search consists of a series of simulated games of self-play that traverse the game tree from root to leaf (with the current board position s being the root). Each simulation proceeds by, starting from the root state, iteratively selecting moves with low visit count, high move probability and high action value, until a leaf node s' is encountered. The leaf position is then evaluated by the neural network to generate both move probabilities and value, $(p', v') = f_\theta(s')$, and added to the game tree. Afterward, the visit counts and action values of all nodes encountered on the path from s to s' are updated and the next simulation, again starting from root s, is initiated. When finally all simulations are performed, the search probabilities π of playing each move given the board position s are calculated proportional to the visit count for each possible move (i.e., higher probability is assigned to moves which were selected in more simulations). Please refer to [19] for a more detailed description of MCTS in AlphaZero, especially how the action value of a move is updated, how the search probabilities are calculated, and how moves are selected with respect to their visit count, probability and action value. In summary, the neural network guides the MCTS algorithm by providing move probabilities and values in specific positions. The MCTS algorithm uses this information within subsequent simulations, and finally calculates search probabilities π of playing each move given a board position s.

The introduced deep neural network and MCTS are the key features of AlphaZero. Starting from randomly initialized parameters θ, the neural network is trained by **reinforcement learning from self-play** (see Fig. 2). To generate training data

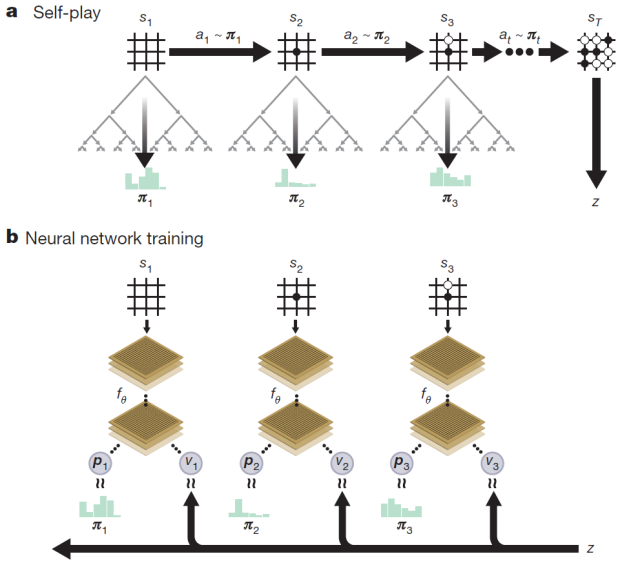

Figure 2: Self-play reinforcement learning in AlphaZero. (a) The program performs games of self-play. Given a board position s_t, a move a_t is selected according to the search probabilities π_t calculated by MCTS. (b) The neural network $f_\theta(s_t) = (p_t, v_t)$ is updated by minimizing the error between predicted outcome v_t and game outcome z_t, and maximizing the similarity of move probabilities p_t and search probabilities π_t. Image from [19].

for the neural network, games are played by selecting moves a_t for both players according to the search probabilities π_t that are calculated by MCTS given a board position s_t. The game outcome z_t is determined according to the game outcome z_T, which is computed by scoring the terminal position s_T with respect to the game rules: -1 for a loss, 0 for a draw, and +1 for a win. Using self-play, training examples of the form (s_t, π_t, z_t) are generated. The neural network $f_\theta(s_t) = (p_t, v_t)$ is now updated by minimizing the error between predicted outcome v_t and game outcome z_t (using the mean-squared error), and maximizing the similarity of move probabilities p_t and search probabilities π_t (using the cross-entropy loss). Therefore, with the parameter c controlling the level of L_2 weight regularization [21], the loss function is defined as:

$$l = (z - v)^2 - \pi^T log p + c||\theta||^2 \tag{1}$$

It is important to note that the search probabilities π, as calculated by the MCTS, usually select much stronger moves than the raw move probabilities p predicted by the neural network. This is why the neural network, when updated with training data (s, π, z) obtained from self-play using MCTS, is indeed an improved neural network. The updated neural network is used in subsequent iterations of self-play, resulting in training data of improved quality.

To summarize, the two main components of AlphaZero's training pipeline are the generation of new self-play data using MCTS guided by the neural network, and the optimization of the neural network parameters θ from recent self-play data. In the original work, both components are actually executed asynchronously in parallel, i.e., AlphaZero maintains a single neural network that is updated continually while also being used to generate new self-play data.

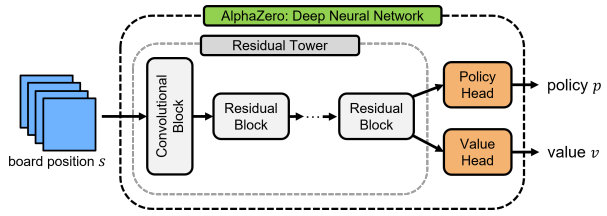

Figure 3: We employ the same neural network architecture that was used for AlphaGo Zero. The input is an image stack corresponding to a given board position, which is processed by a convolutional block and a series of residual blocks, and finally by the policy and value head to calculate policy and value.

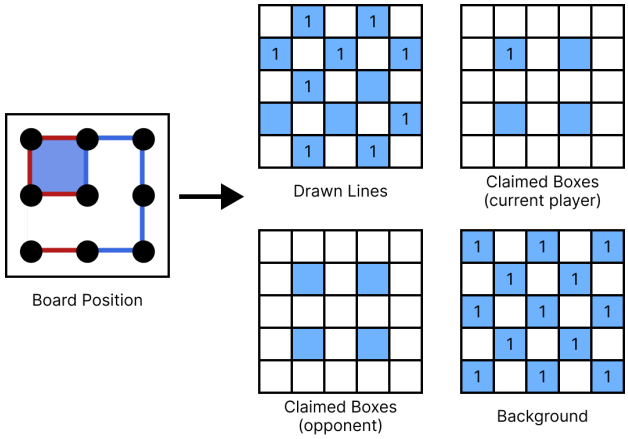

Figure 4: Before forwarded to the neural network, a given Dots and Boxes board position is encoded from the current player's point of view (here: Blue) into four binary feature planes.

3.2 Neural Network Architecture

AlphaZero uses the same convolutional neural network architecture as AlphaGo Zero for chess, shogi, and Go [17] (see Fig. 3). In AlphaGo Zero, the input to the neural network is a $19 \times 19 \times 17$ image stack, consisting of 17 binary feature planes corresponding to a given 19×19 board position [19]. The input features are then processed by a residual tower, consisting of a single convolutional block followed a series of residual blocks. Finally, the output of the residual tower is passed into two separate heads for policy and value calculation. In detail, for AlphaGo Zero the neural network components consist of the following modules.

The **convolutional block** applies (1) a convolution of ($F = 256$, $K = 3$, $S = 1$), i.e., a convolution [9] with F filters of kernel size $K \times K$ with stride S, followed by (2) batch normalization [11] and (3) ReLU activation [2].

Each **residual block** [10] sequentially applies (1) a convolution of ($F = 256$, $K = 3$, $S = 1$), (2) batch normalization, (3) ReLU activation, (4) a convolution of ($F = 256$, $K = 3$, $S = 1$), (5) batch normalization, (6) a skip connection that adds the input to the residual block, and (7) ReLU activation.

The **policy head** applies (1) a convolution of ($F = 2$, $K = 1$, $S = 1$), (2) batch normalization, (3) ReLU activation and (4) a fully connected linear layer that outputs a vector of size $19^2 + 1 = 362$, corresponding to the probabilities of all possible moves offered by the 19×19 Go board (including moves that were already performed and the pass move).

The **value head** applies (1) a convolution of ($F = 1$, $K = 1$, $S = 1$), (2) batch normalization, (3) ReLU activation, (4) a fully connected linear layer to a hidden layer of size 256, (5) ReLU activation, (6) a fully connected linear layer to a scalar and (7) Tanh activation outputting a scalar in the range [-1, 1].

3.3 Adapting AlphaZero to learn and play Dots and Boxes

For Dots and Boxes, we use the same neural network architecture as AlphaGo Zero, i.e., use the exact same sequence of modules for the convolutional block, the residual blocks, and the policy and value head. In order to normalize the output of the policy head to a probability distribution, we use softmax as the final activation function. Due to the lower complexity and dimensionality of Dots and Boxes as compared to chess, shogi, and Go (at least for the board sizes that we consider), what we change compared to AlphaZero's neural network architecture are the module sizes (see Sec. 4.3). That is, depending on the board size of the corresponding Dots and Boxes game, we use fewer residual blocks, use fewer filters for convolutions within the convolutional and residual blocks, and the fully connected layers within the policy and value head contain fewer nodes.

In AlphaGo Zero, the input to the neural network is a $19 \times 19 \times 17$ image stack, consisting of 17 binary feature planes corresponding to a given 19×19 board position. Each eight feature planes (corresponding to the last eight time-steps) consist of binary values indicating the presence of the current player's stones and opponents' stones, respectively. This is necessary for the reason that Go is not fully observable solely from the current stones as repetitions are forbidden. The final feature plane represents the color to play.

Same as for Go, in order to enable the neural network to make valuable predictions for the game of Dots and Boxes a specific set of feature planes has to be designed, containing all important information of a given board position. We encode a given $m \times n$ Dots and Boxes board position into four binary feature planes $s_t = [L_t, P_t, O_t, B_t]$ of size $(2m+1) \times (2n+1)$ (see example in Fig. 4). The first feature plane L_t consists of binary values indicating the presence of drawn lines ($L_t = 1$ if the corresponding line is drawn). The second feature plane P_t consists of binary values indicating the claimed boxes of the current player ($P_t = 1$ if the corresponding box was claimed by the current player). Similarly, O_t consists of binary values indicating the claimed boxes of the opponent. The fourth and last feature plane B_t indicates the pixels that never contain position-specific information (i.e., $B_t = 1$ if the element does not correspond to any line or box), and therefore is the same for any board position of a given size.

Note that for a given Dots and Boxes position, it is of no relevance which lines were drawn by which player, since for claiming a box a player only has to draw its remaining free line. This is why we do not differentiate between the two players for the first feature plane, and provide information about the current score with the second and third feature planes. Furthermore, note that the encoding of a board position into the first three feature planes is not efficient. For example, for a given 2×2 board position, the feature plane corresponding to the claimed boxes of the current player may contain ones for a maximum of only four pixels. This, however, is required in order to allow the use of convolutions within the neural network, allowing to

extract spatial information from board positions. Lastly, we only need four feature planes as opposed to the 17 feature planes that were used for AlphaGo Zero. This is because of Go's property of not being fully observable solely from the current stones. We could perhaps even have reduced the number of feature planes for Dots and Boxes to two by encoding the current score into a feature plane instead of encoding the claimed boxes of both players, and getting rid of the fourth feature map containing background information. This, however, may be the substance of Future Work (see Sec. 6.2).

Same as for Go (and opposed to, e.g., chess), the rules of Dots and Boxes are invariant to rotation and reflection. Same as in AlphaGo Zero, we exploit this by 1) augmenting the generated training examples by including rotations and reflections of each position (results in an eight times larger dataset) and by, 2) during MCTS, sampling a random rotation or reflection of a given board position before being evaluated by the neural network (averaging the Monte-Carlo evaluation over different biases).

Same as in AlphaZero, we maintain a single neural network that is updated continually, i.e., self-play games are generated by using the latest updated neural network parameters. This is in contrast to AlphaGo Zero, where each updated neural network is only used for self-play generation in subsequent training iterations when it wins a specific margin of games against the current best network. Still, our training pipeline differs from AlphaZero in the way that the generation of self-play data using MCTS and the neural network update are not executed asynchronously in parallel, but separately one after the other (see Sec. 4.2).

4 EXPERIMENTAL APPARATUS

4.1 Baselines

To analyze the capabilities of our trained instances of AlphaZero to play and win games of Dots and Boxes, we implemented two baseline game-playing AIs. The *RandomPlayer* randomly chooses a valid move (i.e., a random line which is not drawn yet) at each step in the game. Further, the *AlphaBetaPlayer* uses the minimax algorithm with alpha-beta pruning to determine its next move (see Sec. 2.3). Based on our formulated evaluation function, the *AlphaBetaPlayer* aims to maximize its own claimed boxes and minimize the boxes claimed by AlphaZero within the next d moves. We limit the search depth of the *AlphaBetaPlayer* to $d = 1, 2, 3$ moves. The move selection of our *AlphaBetaPlayer* could be improved by a more sophisticated evaluation function, which returns higher values for board positions that more likely result in a positive game outcome [4].

4.2 Implementation

We implemented the Dots and Boxes game logic, the AlphaZero algorithm including MCTS, the deep neural network and the training pipeline, and the baseline methods from scratch in Python, using PyTorch 1.12.1 for the implementation of the neural network. Since during a training iteration we perform the generation of self-play data using MCTS and the neural network update separately one after the other (and not asynchronously in parallel), our AlphaZero training pipeline differs slightly from the one used in the original work. Due to the high computational effort required to train AlphaZero (see Sec. 1) and our limitations in computational resources, we were only able to train and evaluate AlphaZero for the 2×2, 3×3, and 4×4 Dots and Boxes games. For example, a single training iteration of AlphaZero using the hyperparameters presented in the following section lasted approximately 2.5 hours for the 4×4 board, including generation of self-play games, neural network update and evaluation against the baseline methods. Our implementation code, including the hyperparameter configurations and trained neural networks for the 2×2, 3×3, and 4×4 boards, is publicly available at https://github.com/josch14/alpha-zero-dots-and-boxes.

4.3 Hyperparameters

Some research argues that there is no sufficient discussion about how to tune the hyperparameters for AlphaZero [22], and other sources discuss how proper hyperparameter selection accelerates training progress and improves the quality of AlphaZero move selections[1]. Due to the high computational effort required for AlphaZero training and the many hyperparameters for its different components, hyperparameter optimization for AlphaZero is difficult. In AlphaGo Zero, the MCTS hyperparameters were tuned by Bayesian optimization. In the follow-up AlphaZero, the same hyperparameters were re-used for chess, shogi and Go without game-specific tuning. Since we employed AlphaZero for a fundamentally different game than chess, shogi and Go, we could only guess a sensible set of hyperparameters. In the following, we list and explain the hyperparameters of AlphaZero and present the values that we used for training and playing AlphaZero for Dots and Boxes on different board sizes.

Self-Play.
During each training iteration, self-play data is generated by playing 500 games of Dots and Boxes (AlphaGo Zero: 25,000), using 100 / 150 / 150 simulations of MCTS to select each move on the 2×2 / 3×3 / 4×4 board (AlphaGo Zero: 1,600).

The temperature move threshold m is a parameter regarding the search space diversity. For the first $n \leq m$ moves of each self-play game, moves are selected proportionally to their visit count in MCTS, ensuring a diverse set of encountered positions. For the remainder of the game, the best moves are chosen according to the search probabilities calculated by MCTS. We set the temperature move threshold to 5 / 7 / 10 (AlphaGo Zero: 30). Note that during evaluation, i.e., playing games against the baselines, the threshold is set to 0 in order to always select the strongest possible play determined by MCTS.

Another parameter balancing the exploration and exploitation in MCTS is c_{puct}, which determines how much weight to give to the prior probability of a move versus its action value (please refer to [19] for details). We set $c_{puct} = 1$ for all board sizes.

Finally, additional exploration is achieved by adding Dirichlet noise to the move probabilities in root s. This encourages to explore a wider variety of moves during MCTS, rather than always selecting the most promising moves. The noise η is sampled from a Dirichlet distribution, with $\eta \sim \text{Dir}(\alpha)$, and added to the move probabilities with a weight of ϵ. Usually, α is scaled in inverse proportion to the approximate number of legal moves in a typical position. Therefore, we set α to 1.0 / 1.0 / 0.5 (AlphaGo Zero: 0.03), and the weight ϵ to 0.25 (AlphaGo Zero: 0.25).

Neural Network Architecture.
We use the same neural network architecture as AlphaGo Zero (see Sec. 3.2). Due to the lower complexity and dimensionality of Dots and Boxes (for the board sizes that we consider) as compared to chess, shogi, and Go, we reduced the neural network

[1] Lessons from AlphaZero: Hyperparameter Tuning. By Anthony Young, available at https://medium.com (visited at April 07, 2023).

module sizes. As a consequence of our restriction in computational resources not allowing us to optimize the neural network size in reasonable time, we use the same neural network module sizes for the 2×2, 3×3 and 4×4 boards.

Our deep neural network employs 4 residual blocks (AlphaGo Zero: 20). Each convolution within the convolutional block and each residual block applies 64 filters (256) of kernel size 3×3 with stride 1. Same as in AlphaGo Zero, the convolutions within the policy and value heads apply 2 and 1 filter of size 1×1 and stride 1, respectively. The number of neurons of a fully connected layer within the policy or value head is reduced proportional to the board size.

Neural Network Optimization.
In AlphaGo Zero, neural network parameters were optimized by stochastic gradient descent, using the loss in equation (1), with momentum and learning rate annealing following a specific schedule. Furthermore, L2 weight regularization [21] was used to reduce overfitting. In order to be less sensitive to a learning rate schedule, instead of stochastic gradient descent we use the Adam optimizer [12] with a learning rate of 0.001, and set the L2 regularization parameter to 10^{-4} (AlphaGo Zero: 10^{-4}).

During a single training iteration, the neural network is updated from 1500 / 3000 / 5600 mini-batches of 512 positions (AlphaGo Zero: 2,048 positions). Each mini-batch is sampled uniformly at random from all positions of the most recent 5,000 games (AlphaGo Zero: 500,000), i.e., from all positions from generated self-play games of the 10 most recent training iterations.

5 RESULTS

5.1 Training Evaluation

We trained AlphaZero for 20, 25, 25 iterations for the 2×2, 3×3, and 4×4 Dots and Boxes game, respectively. Figure 5 shows the policy loss, value loss, and total loss evolutions during AlphaZero training, with each left plot displaying the calculated loss of each batch update during model training, and each right plot displaying the evaluation loss which is calculated by again calculating the loss for all batches after model update. For all board sizes we observe that especially during the first few iterations and after the 10th iteration the total loss decreases drastically. The loss decrease after the 10th iteration is due to the fact that each mini-batch is sampled from all positions from generated self-play games of the 10 most recent training iterations, meaning that, e.g., with the 11th iteration onwards the self-play games of the first iteration (i.e., the training data of lowest quality) are not used anymore. Further, we observe that in general the policy loss is higher for larger board sizes. This may be due to the fact that the prediction of the policy head, i.e., a vector containing move probabilities, is of higher dimension for larger board sizes. Another reason could be that the deep neural network, as specified in Section 4.3, may not be sufficiently large anymore for larger board sizes. For example, using more residual blocks or using convolutions with a higher number of filters may help the neural network to further minimize loss. Interestingly, as opposed to the training for the 2×2 and 3×3 board, for the 4×4 board the value loss remains nearly unchanged and does not drop below 0.5 within the first 10 training iterations. Although we terminated AlphaZero training after 20, 25, and 25 iterations respectively, the loss evolution towards the final iteration suggests that, at least

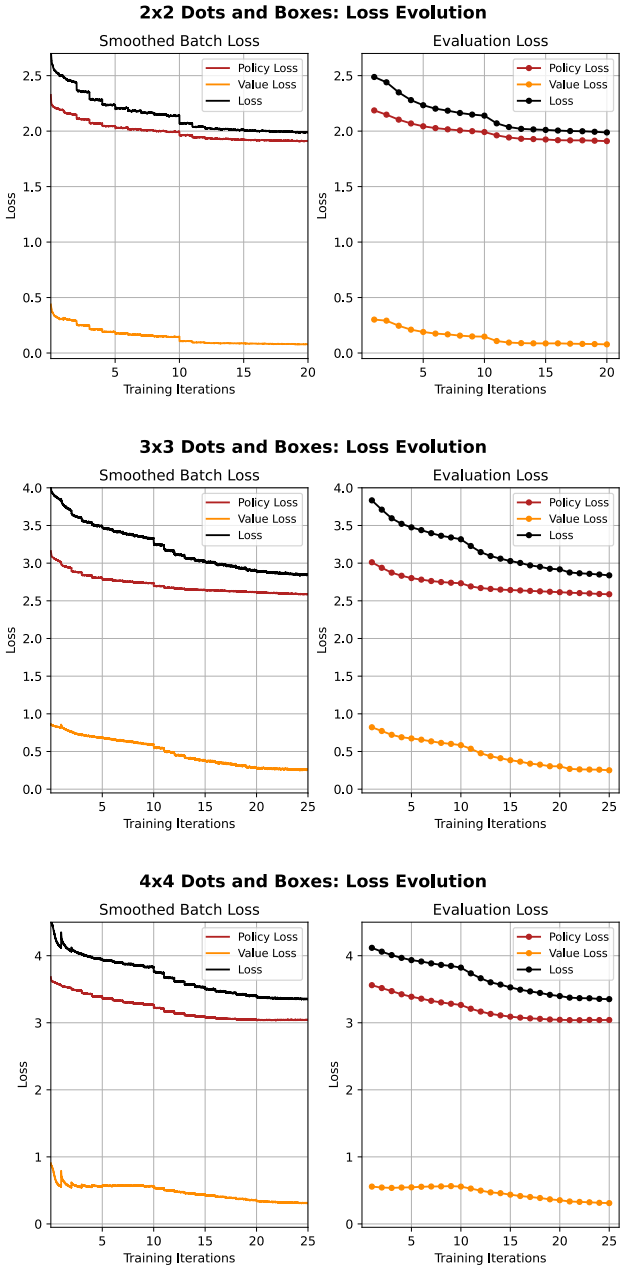

Figure 5: Loss evolution during AlphaZero training for the 2×2, 3×3, and 4×4 Dots and Boxes game. The plots show smoothed loss of each batch update during model training (left) and evaluation loss, i.e., the average loss calculated again for all batches after model update (right).

for the 3×3 and 4×4 board, further decrease in loss is achievable. Therefore, by training AlphaZero in additional iterations, an improved performance against opponents can be expected.

After each neural network update, we evaluated AlphaZero's capability to win games by playing a total of 100 games (each 50 games as player with first and second move) against our baseline game-playing AI methods (see Sec. 4.1). The results evolution is shown in Figure 6. Note that, since the total number of claimable boxes is even for the 2×2 and 4×4 boards, a Dots and Boxes game may result in a draw for these board sizes. For the 2×2 board, we observe that already after 2 iterations AlphaZero is

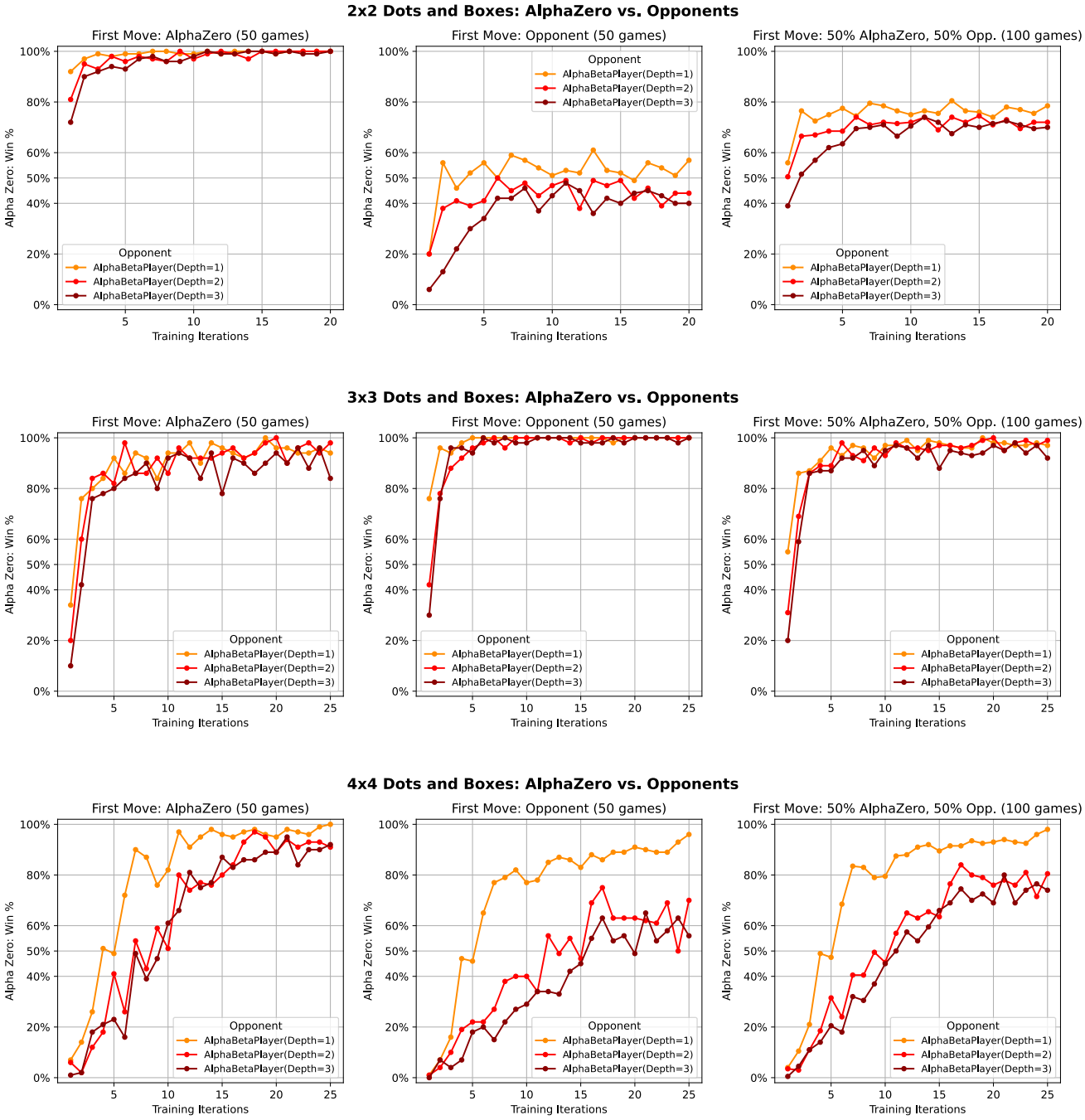

Figure 6: Results evolution during AlphaZero training for the 2×2, 3×3, and 4×4 Dots and Boxes game. After each training iteration, AlphaZero played 100 games (each 50 games as player with first and second move) against the baseline methods. For the 2×2 and 4×4 game, a draw is counted as half win.

able to win nearly all games when having the first move. However, when having the second move, AlphaZero's performance stagnates after only 6 training iterations. For example, against the *AlphaBetaPlayer* with a search depth of 3, AlphaZero never wins more than 50 % of the games when being having the second moves. This may be due to a great advantage for the starting player on the 2×2 board which AlphaZero can not compensate for with the limited number of moves in a Dots and Boxes game of this size. For the 3×3 board, we observe that AlphaZero is able to win nearly all games against all opponents after few iterations, regardless of being the starting or second player. In contrast to 2×2 board, AlphaZero performs slightly worse when having the first move on the 3×3 board, which may be an indication that having the first move in a 3×3 Dots and Boxes game comes with a small disadvantage. Finally, for the 4×4 board we observe that AlphaZero's performance against the opponents improves with nearly each training iteration, including the final iteration. This implies that, in contrast to the training for the other board sizes, the 25 iterations were not sufficient to show the full capabilities of AlphaZero playing 4×4 Dots and Boxes.

AlphaZero vs. Opponents: 2×2 Dots and Boxes (after 20 iterations)						
	wins / losses			win % of AlphaZero		
Opponent	starting	second	total	starting	second	total
RandomPlayer	250 / 0 / 0	184 / 65 / 2	434 / 64 / 2	100.00	89.40	93.20
AlphaBetaPlayer(depth=1)	250 / 0 / 0	34 / 209 / 7	284 / 209 / 7	100.00	55.40	77.70
AlphaBetaPlayer(depth=2)	250 / 0 / 0	3 / 204 / 43	253 / 204 / 43	100.00	42.00	71.00
AlphaBetaPlayer(depth=3)	248 / 2 / 0	6 / 187 / 57	254 / 189 / 57	99.60	39.80	69.70

AlphaZero vs. Opponents: 3×3 Dots and Boxes (after 25 iterations)						
	wins / losses			win % of AlphaZero		
Opponent	starting	second	total	starting	second	total
RandomPlayer	250 / 0 / 0	250 / 0 / 0	500 / 0 / 0	100.00	100.00	100.00
AlphaBetaPlayer(depth=1)	240 / 0 / 10	250 / 0 / 0	490 / 0 / 10	96.00	100.00	98.00
AlphaBetaPlayer(depth=2)	235 / 0 / 15	250 / 0 / 0	485 / 0 / 15	94.00	100.00	97.00
AlphaBetaPlayer(depth=3)	227 / 0 / 23	247 / 0 / 1	476 / 0 / 24	90.80	99.60	95.20

AlphaZero vs. Opponents: 4×4 Dots and Boxes (after 25 iterations)						
	wins / losses			win % of AlphaZero		
Opponent	starting	second	total	starting	second	total
RandomPlayer	250 / 0 / 0	250 / 0 / 0	500 / 0 / 0	100.00	100.00	100.00
AlphaBetaPlayer(depth=1)	242 / 7 / 1	221 / 21 / 8	463 / 28 / 9	98.20	92.60	95.40
AlphaBetaPlayer(depth=2)	231 / 15 / 4	149 / 34 / 67	380 / 49 / 71	95.40	66.40	80.90
AlphaBetaPlayer(depth=3)	225 / 14 / 11	142 / 32 / 76	367 / 46 / 87	92.80	63.20	78.00

Table 1: Results of AlphaZero playing Dots and Boxes against our baseline opponents on different board sizes. For the 2×2 and 4×4 boards, a draw is counted as half win.

The detailed results of AlphaZero playing Dots and Boxes against our baseline game-playing AIs after the final training iteration are shown in Table 1. Here, we let AlphaZero play 500 games against each opponent for each board size. Largely the same observations can be made as for Figure 6. For the 2×2 Dots and Boxes game we now observe that, when having the second move of a game (and therefore the suggested disadvantage), AlphaZero is actually able to reach a draw in most games, and is defeated less frequently. In contrast to that, when the *AlphaBetaPlayer* has the second move, it was able to reach a draw in only 2 games when using a search depth of 3. Since even the *RandomPlayer* was able to reach a draw in 65 games (and even win 2 games) when having the first move, we conclude that for the 2×2 Dots and Boxes game the player with the first move in fact has an unfair advantage over the second player.

5.2 Gameplay Analysis

We now want to analyze the move selections and the overall capability of AlphaZero for Dots and Boxes in a more qualitative way. For this, we played games of Dots and Boxes against the trained AlphaZero instances.

AlphaZero (having the first move) won the 2×2 game shown in Figure 7 by claiming a box with the 7th overall move of the game, and more importantly, by sacrificing the bottom right box with the following move by drawing a line to this box. We accepted this sacrifice, and by having to draw an additional line, AlphaZero is able to claim the remaining boxes. Note that the *AlphaBetaPlayer*s (which use a maximum search depth of 3) would have made the same move selection as we did, as it aims to maximize its own claimed boxes and minimize the claimed boxes of its opponent within the next few moves.

AlphaZero (having the second move) lost the 2×2 game shown in Figure 8 while not claiming a single box. Even when we assume a disadvantage for the player having the second move, AlphaZero would have been able to claim at least one box. With the 6th move, AlphaZero decides to draw the line between the center left and center Dots, allowing us to claim the top left box. Afterward, we are able to draw another line without enabling AlphaZero to claim a box with the following move. Instead, with the 9th move AlphaZero is now forced to draw a line within the chain, allowing us to claim all remaining boxes.

Finally, AlphaZero (having the first move) won the 3×3 game shown in Figure 9. AlphaZero, while finally showcasing its forward-looking move selection, allowed us to claim 4 boxes, before claiming all remaining boxes. Various moves were decisive here. With the 13th overall move, AlphaZero sacrifices the bottom right box. We accept the sacrifice, and aim to sacrifice the top center box. Instead, in addition to the top center box, with the 16th overall move, AlphaZero also sacrifices the top left and center left box. This move finally made us losing the game since we, regardless of the move we select, are being forced to draw a line inside the long chain. We decide to finally claim all three sacrifices, before having to draw a line within the chain with the 19th move. This enables AlphaZero to finally claim all remaining boxes and win the game. Therefore, we observe that AlphaZero is in fact able to value the importance of long chains in Dots and Boxes games, and selects moves aimed at forcing the opponent to draw a line within such a chain.

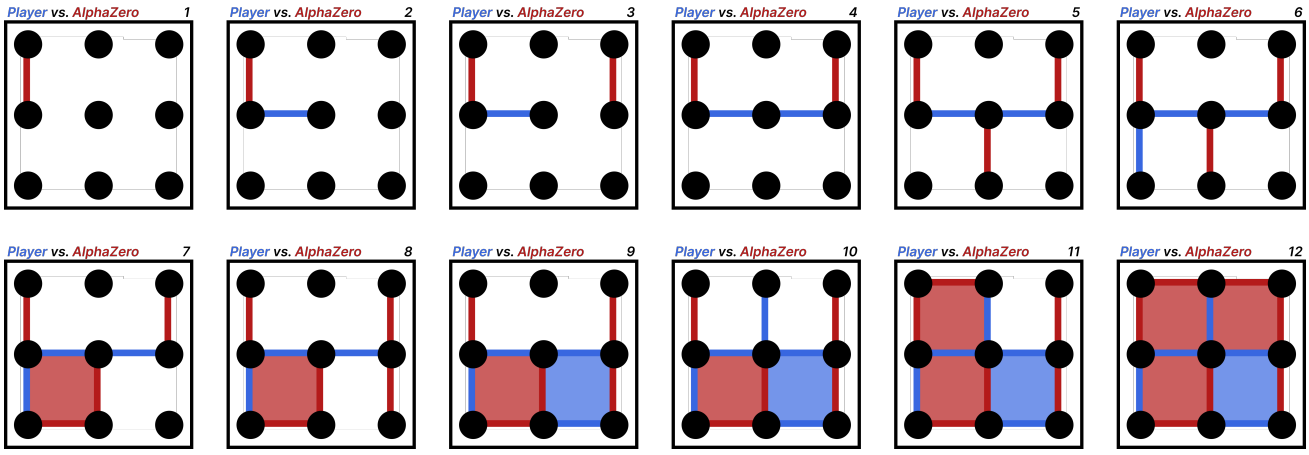

Figure 7: AlphaZero (having the first move) winning a game of Dots and Boxes on the 2×2 board against a human player. AlphaZero had the first move.

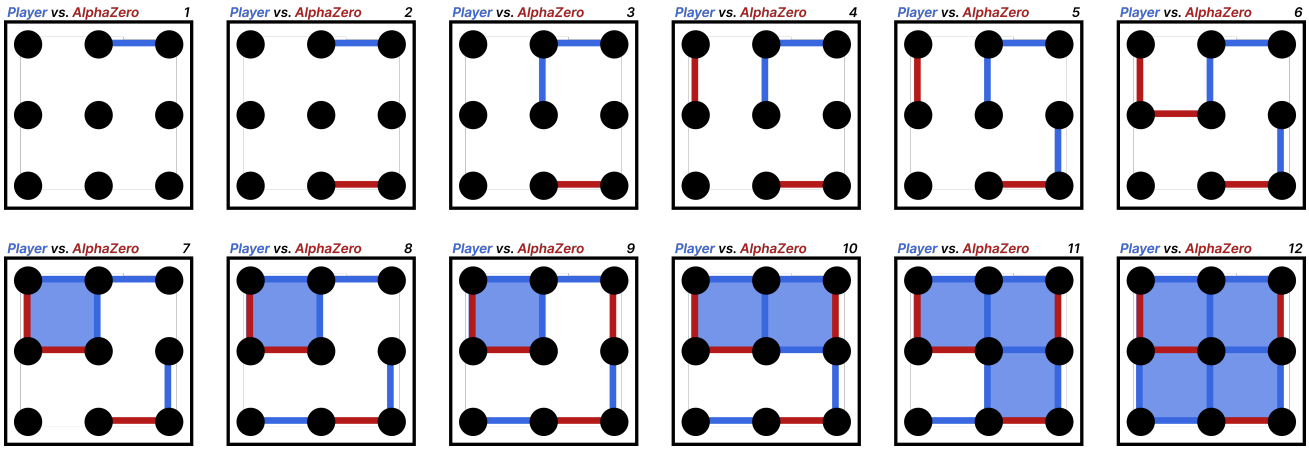

Figure 8: AlphaZero (having the second move) losing a game of Dots and Boxes on the 2×2 board against a human player.

6 DISCUSSION

6.1 Key Results

We observed that AlphaZero is able to win a high fraction of games against our baseline game-playing AIs. For 2×2 Dots and Boxes, AlphaZero is capable of winning almost all games when having the first move (which corresponds with an advantage on the 2×2 board), and is able to reach a draw in most games when having the second move. In general, AlphaZero is more computationally efficient than the *AlphaBetaPlayer*s, i.e., game-playing AIs which use minimax algorithm with alpha-beta pruning. Alpha-beta search evaluates every possible move in the game tree up to a certain depth, while AlphaZero uses MCTS guided by the deep neural network to efficiently search through the game tree and focus on the most promising moves. Consequently, when giving both methods limited amount of time for move selection, AlphaZero will consistently choose better moves. We observed that for 3×3 and 4×4 Dots and Boxes, AlphaZero is able to win most games, regardless of being the first or second player. Therefore, we conclude that AlphaZero can really showcase its forward-looking move selection only for sufficiently large boards sizes.

Unsurprisingly, we observed that AlphaZero requires more training iterations for increasing board sizes. Specifically for the 4×4 board, although after 25 training iterations not yet achieved, there is no reason why AlphaZero should not be capable of winning most games when being the player with the second move. The large computational effort required to train and fine-tune AlphaZero is one of the key limitations of AlphaZero. Most likely, e.g., for the 4×4 Dots and Boxes game, there is a set of hyperparameters different from ours that would allow faster improvement of AlphaZero during training, and thus fewer training iterations would be required until satisfactory performance.

6.2 Future Work

Even though we successfully employed AlphaZero for playing the game of Dots and Boxes, there are several open questions that require a more detailed analysis.

AlphaZero includes many hyperparameters. The large computational effort required for AlphaZero training makes hyperparameter optimization difficult. This forced us to only guess a sensible set of hyperparameters, resulting in the AlphaZero performance that we presented. Furthermore, even for the selected hyperparameter sets for the different board sizes, our computational resources limited us in the hyperparameter selection itself.

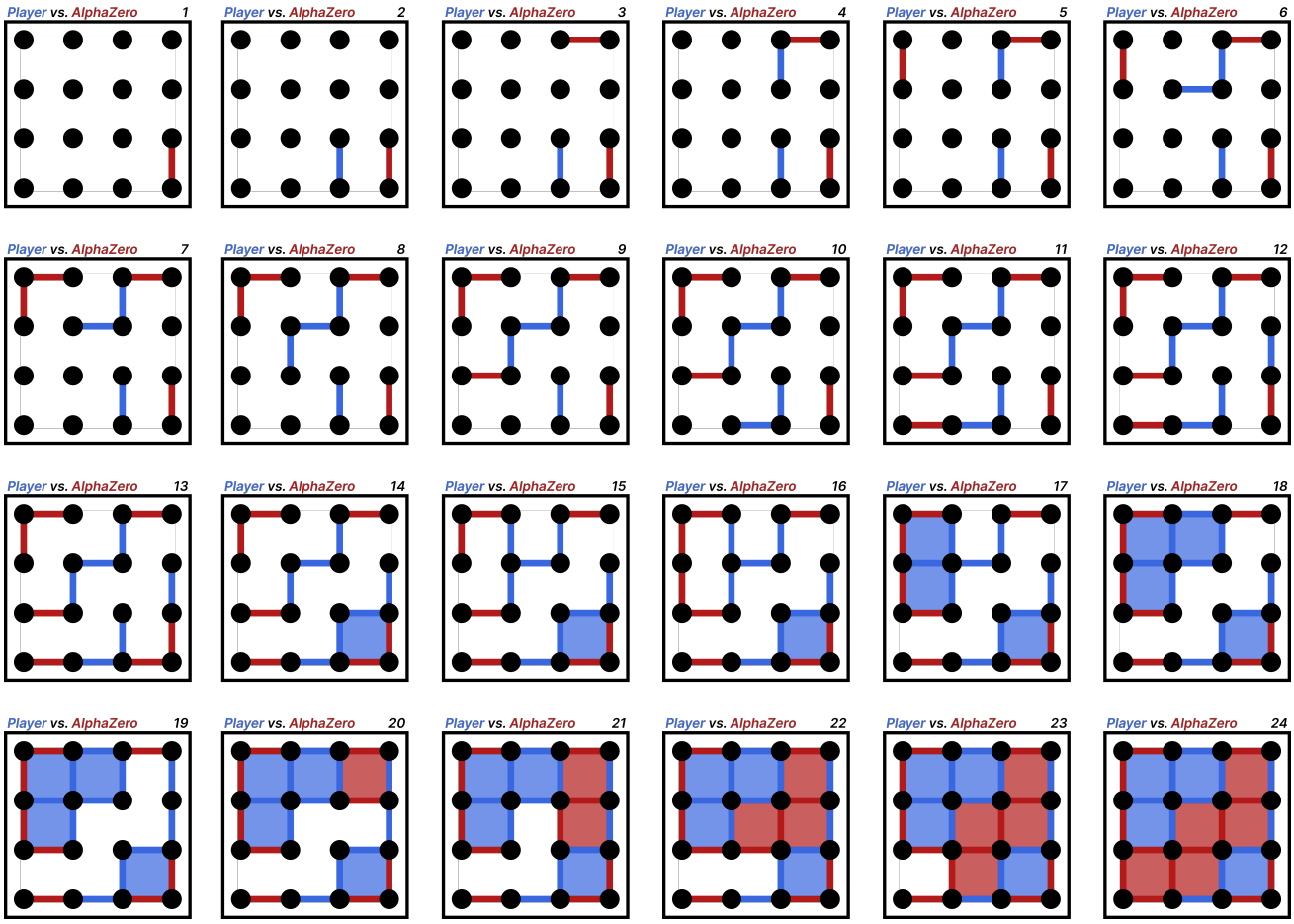

Figure 9: AlphaZero (having the first move) winning a game of Dots and Boxes on the 3×3 board against a human player.

Most importantly, we expect the number of simulations during MCTS and the number of residual blocks within the deep neural network to be the key hyperparameters. However, e.g., for the 4×4 Dots and Boxes game, we were forced to limit the number of simulations to 150 and use a maximum of 4 residual blocks. Therefore, we are confident that with larger computational resources, we could improve AlphaZero's move selection on the one side, and accelerate the training progress of AlphaZero through proper hyperparameter tuning.

In addition, sufficiently large computing capacities are the pre-condition to train and use AlphaZero for larger board sizes. It would be interesting to analyze in which behavior the computational effort increases in order to successfully train AlphaZero for Dots and Boxes on larger boards.

We showed AlphaZero's capabilities to consistently beat our *AlphaBetaPlayer* baselines, i.e., a minimax algorithm with alpha-beta pruning. However, the evaluation function that we designed for the *AlphaBetaPlayer*s is not really sophisticated, as it only maximizes its own claimed boxes and minimizes the claimed boxes of the opponent within the next few moves. Consequently, our results offer only limited conclusions in the sense that we didn't evaluate AlphaZero's performance against better Dots and Boxes AIs. Therefore, this remains an open task. For example, a more sophisticated evaluation function giving higher values for board positions that more likely result in a positive game outcome could be defined for the *AlphaBetaPlayer*. This could enable more efficient alpha-beta pruning, potentially allowing to increase the search depth of the minimax-search.

Finally, in Section 3.3 we argued that perhaps we could have reduced the number of feature planes when encoding a Dots and Boxes position. In general, it would be interesting to analyze if and how different feature plane encodings affect the training and overall performance of AlphaZero.

7 CONCLUSION

We successfully trained AlphaZero to play and win games of Dots and Boxes through self-play learning, without any human knowledge except the game rules. Our available computational resources limited the hyperparameter selection and training of AlphaZero. Still, we were able to consistently beat our random player and alpha-beta player baselines for all board sizes, showing that it is possible to employ AlphaZero for games of reduced complexity, even when computational resources are limited.

REFERENCES

[1] Bruce Abramson. 1989. Control strategies for two-player games. *ACM Computing Surveys (CSUR)* 21, 2 (1989), 137–161.

[2] Abien Fred Agarap. 2018. Deep Learning using Rectified Linear Units (ReLU). *CoRR* abs/1803.08375 (2018). arXiv:1803.08375 http://arxiv.org/abs/1803.08375

[3] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. 2017. Deep Reinforcement Learning: A Brief Survey. *IEEE Signal Processing Magazine* 34, 6 (2017), 26–38. https://doi.org/10.1109/MSP.2017.2743240

[4] Joseph Kelly Barker and Richard E. Korf. 2012. Solving Dots-And-Boxes. *Proceedings of the AAAI Conference on Artificial Intelligence* (2012).

[5] E.R. Berlekamp. 2000. *The Dots and Boxes Game: Sophisticated Child's Play.* Taylor & Francis. https://books.google.de/books?id=TRk4ZLqZwCEC

[6] Kevin Buchin, Mart Hagedoorn, Irina Kostitsyna, and Max van Mulken. 2021. Dots & boxes is pspace-complete. *arXiv preprint arXiv:2105.02837* (2021).

[7] Murray Campbell, A.Joseph Hoane, and Feng hsiung Hsu. 2002. Deep Blue. *Artificial Intelligence* 134, 1 (2002), 57–83. https://doi.org/10.1016/S0004-3702(01)00129-1

[8] Johannes Czech, Patrick Korus, and Kristian Kersting. 2020. Monte-Carlo Graph Search for AlphaZero. *CoRR* abs/2012.11045 (2020). arXiv:2012.11045 https://arxiv.org/abs/2012.11045

[9] Vincent Dumoulin and Francesco Visin. 2018. A guide to convolution arithmetic for deep learning. arXiv:1603.07285 [stat.ML]

[10] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. *CoRR* abs/1512.03385 (2015). arXiv:1512.03385 http://arxiv.org/abs/1512.03385

[11] Sergey Ioffe and Christian Szegedy. 2015. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *CoRR* abs/1502.03167 (2015). arXiv:1502.03167 http://arxiv.org/abs/1502.03167

[12] Diederik P. Kingma and Jimmy Ba. 2017. Adam: A Method for Stochastic Optimization. arXiv:1412.6980 [cs.LG]

[13] Yuxi Li. 2017. Deep Reinforcement Learning: An Overview. *CoRR* abs/1701.07274 (2017). arXiv:1701.07274 http://arxiv.org/abs/1701.07274

[14] Yunlong Lu and Wenxin Li. 2022. Techniques and Paradigms in Modern Game AI Systems. *Algorithms* 15, 8 (2022). https://doi.org/10.3390/a15080282

[15] A. L. Samuel. 1959. Some Studies in Machine Learning Using the Game of Checkers. *IBM Journal of Research and Development* 3, 3 (1959), 210–229. https://doi.org/10.1147/rd.33.0210

[16] Claude E Shannon. 1950. XXII. Programming a computer for playing chess. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 41, 314 (1950), 256–275.

[17] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. 2018. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science* 362, 6419 (2018), 1140–1144. https://doi.org/10.1126/science.aar6404 arXiv:https://www.science.org/doi/pdf/10.1126/science.aar6404

[18] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy P. Lillicrap, Karen Simonyan, and Demis Hassabis. 2017. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. *CoRR* abs/1712.01815 (2017). arXiv:1712.01815 http://arxiv.org/abs/1712.01815

[19] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George Driessche, Thore Graepel, and Demis Hassabis. 2017. Mastering the game of Go without human knowledge. *Nature* 550 (10 2017), 354–359. https://doi.org/10.1038/nature24270

[20] Richard S Sutton, Andrew G Barto, et al. 1999. Reinforcement learning. *Journal of Cognitive Neuroscience* 11, 1 (1999), 126–134.

[21] Twan van Laarhoven. 2017. L2 Regularization versus Batch and Weight Normalization. *CoRR* abs/1706.05350 (2017). arXiv:1706.05350 http://arxiv.org/abs/1706.05350

[22] Hui Wang, Michael Emmerich, Mike Preuss, and Aske Plaat. 2019. Hyper-Parameter Sweep on AlphaZero General. *CoRR* abs/1903.08129 (2019). arXiv:1903.08129 http://arxiv.org/abs/1903.08129

[23] Hui Wang, Mike Preuss, and Aske Plaat. 2021. Adaptive warm-start MCTS in AlphaZero-like deep reinforcement learning. In *PRICAI 2021: Trends in Artificial Intelligence: 18th Pacific Rim International Conference on Artificial Intelligence, PRICAI 2021, Hanoi, Vietnam, November 8–12, 2021, Proceedings, Part III.* Springer, 60–71.