# DeepZertz: Implementing AlphaZero to Play Zertz

**Kevin Culberg** [1] **Wei-ting Hsu** [1] **Alexia Xu** [1]

## Abstract

Inspired by the recent success in reinforcement learning for game playing that combine neural networks and Monte Carlo Tree Search, we adapted techniques from *AlphaGo Zero* and *AlphaZero* to the board game Zertz. Zertz provides some unique differences to Go, chess, and shogi in that it is played on a shrinking hexagonal board, both players use the same game pieces, and not all actions end the current player's turn. We managed to successfully train an agent to play a 19 ring variant of Zertz at the skill level of an average human opponent through self play generated games from a randomly initialized convolutional neural network with no dataset of recorded games zero domain knowledge except the simulation rules.

## 1. Introduction

Inspired by the recent breakthrough in reinforcement learning, AlphaGo Zero (Silver et al., 2017b) and AlphaZero (Silver et al., 2017a) by DeepMind, which was able to develop superhuman performance in Go, chess, and shogi without guidance from human experts, we are intrigued to re-implement AlphaZero on other strategic games with similar state and action space complexity. AlphaZero utilizes DNN to train value and policy functions, MCTS to evaluate future moves, and sophisticated procedures to perform self-plays and re-evaluations. We will investigate each component of AlphaZero and adapt them to train an agent to play Zertz.

Zertz (as shown in Figure 1) is a highly strategic full knowledge game where the two players take turns placing or capturing marbles on a hexagonal board of shrinking size with the goal of capturing certain combinations of marbles by jumping across them with other marbles. With a regular size board of 37 rings, the state space has an upper bound of $5^{37}$, and the game involves a lot of delayed consequences as players must sacrifice marbles with low importance to force poor moves by the opponent and enable the capture of high importance marbles. Different from most games where there is one particular goal for players to achieve, players in Zertz can win by capturing either a certain number of marbles of the same color (four white, five gray, or six black), or a combination of all marble types (three of each color), which further complicates the game as the importance of each type of marble to each player changes as the game progresses.

In this project, we are going to re-implement AlphaZero from scratch to play Zertz. We will evaluate each trained agent during simulated matches against each other, other available AI implementations, and human players. We also adapted a 19 ring variant of Zertz in order to evaluate our implementation on a smaller state and action space.

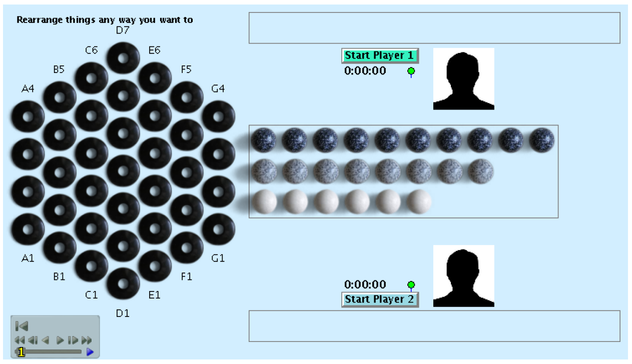

*Figure 1.* Set up of Zertz. At the beginning of the game, 37 rings assemble a hexagonal game board. In each turn, the player either places a marble and removes a ring from the board or captures a marble. The first player to capture either 3 marbles of each color, 4 white marbles, 5 gray marbles, or 6 black marbles wins. Screenshot obtained from Boardspace.net.

## 2. Related Work

AlphaGo (Silver et al., 2016) was the first program to achieve superhuman performance in Go. It is famous for defeating the European champion Fan Hui and Lee Sedol, the winner of 18 international titles. AlphaGo has two deep neural networks: the policy network predicts human expert moves and later is refined by policy gradient. The value network predicts the winner of the self-play games. Monte

[1]Computer Science Department, Stanford University. Correspondence to: <kculberg, hsuwt, alexiaxu@stanford.edu>.

Carlo Tree Search (MCTS) (Browne et al., 2012) was incorporated into the model to provide lookahead search and became a key to its success.

In 2017, AlphaGo Zero (Silver et al., 2017b) became the strongest Go agent in the world. Comparing to AlphaGo, AlphaGo Zero is trained solely by self-play reinforcement learning, starting from random play, without any supervision or use of human data. It uses a single neural network with two heads to predict the policy and the value for a given game state. Based on AlphaGo Zero, AlphaZero (Silver et al., 2017a) generalizes the same approach and achieves super human performance on chess, shogi, and Go, without any additional domain knowledge except the rules of the game. It gives us the hope of conquering other challenging reinforcement learning domains than board games.

To decide which board game to explore, we investigated the complexity of various strategic full-knowledge board games. (Heule & Rothkrantz, 2007) showed that Zertz allows complicated strategy but constrains the length of episodes by shrinking the game board over time. Since Zertz balances game complexity and the size of state space, we choose it as the task for our AlphaZero implementation.

## 3. Approach

### 3.1. Zertz Implementation

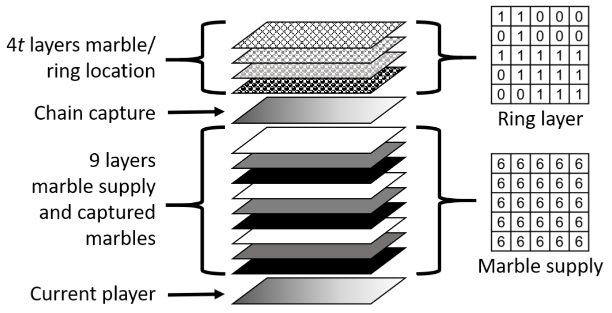

*Figure 2.* Game state representation for Zertz used as input to neural networks. The first $4t$ layers represent locations of marbles and rings on the game board. The following 11 layers record information about chain captures, counts of marbles in supply or captured, and the current player.

We represent the game of Zertz as a finite state MDP. The game state is represented by a matrix with three dimensions $x \times y \times l$, see Figure 2. The height, $y$, and width, $x$, of the state matrix matches that of the game board. The final dimension corresponds to the number of layers, $l$, with each layer representing information about the game. The first $4t$ layers, where $t$ is the number of remembered turns, represent the locations of rings and marbles in a binary matrix. The following layer is used to track chain capture actions and contains a single 1 at the board position of the marble being used in a chain capture with 0's at all other locations. The following 9 layers record the supply and amount of marbles captured for each player and for each type of marble (white, gray, and black). Each layer in this group is set to the number of marbles of that type. For example, if there are 4 white marbles in the supply, then that corresponding layer will be set to all 4. The final layer in the game state is set to the value of the current player, which is 0 for player 1 and 1 for player 2. The game state matrix size for a 37 ring game of Zertz where we remember the previous 5 turns $(t = 5)$ is $7 \times 7 \times 31$.

There are two types of actions that a player may take in Zertz. First is a capture action which consists of selecting a marble on the board, moving it to a location on the other side of a neighboring marble, removing the captured marble from the board, and incrementing the number of captured marbles of that type for the player. This method of capturing by 'jumping' over a neighboring marble is similar to the game checkers and can happen in any of the six directions as long as the landing ring is empty. Capture actions are compulsory, meaning the player must take a capture action if one is available, and can be chained together which means that the player's turn may not end after a capture action is taken. The second type of action is a placement action which consists of selecting a marble type to place on the board, a location of an empty ring to place that marble, and the location of an empty ring to remove from the board and shrink its size. To create the policy for any state both a capture action matrix and a placement action matrix are flattened and concatenated together into a single vector. Capture actions are represented in matrix form with dimensions $x \times y \times 6$ for the height and width of the game board and a layer for each direction. Each index into this matrix corresponds to the location of the marble doing the capturing and which direction it is capturing. Placement actions are also represented in matrix form with dimensions $xy \times (xy + 1) \times 3$. The first index corresponds to the $x, y$ location to place a marble. The second index corresponds to the $x, y$ location to remove a ring with an added option to not remove a ring because this part of the action is not possible in some board states. The final index corresponds to the type of marble that will be placed.

All state transitions in Zertz are guaranteed and there is no knowledge hidden from the players. Once a player takes an action available to them then the board and counts of marble types are updated accordingly and play transfers to the next player if there does not exist a chain capture. Rewards for all state transitions are 0 except for transitioning to an end state in which case the reward is 1 or -1 according to the player that caused the game to end. The game is in an end state if a player has met at least one of the win conditions by capturing enough marbles of certain types or there are no more valid actions. In the very rare event that the game progresses past a high number of turns we declare the game a draw and assign the reward as 0.

## 3.2. Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) (Chaslot, 2010) (as shown in Figure 3) is undoubtedly one of the most important algorithms that brings success to AlphaGo. It was the first search algorithm that enabled AI to achieve amateur level play in Go(Gelly & Silver, 2008). MCTS strengths come in when the state space is too large for other tree search algorithms like minimax to be feasible, or when the available actions at each state differ, which make algorithms like policy gradient difficult to train. Zertz, like Go, is a game that illustrates both of the properties mentioned above.

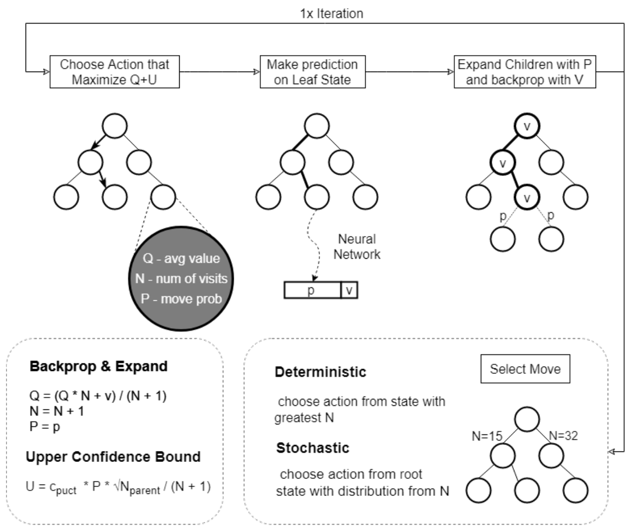
1x Iteration
Choose Action that Maximize Q+U
Make prediction on Leaf State
Expand Children with P and backprop with V
Q - avg value
N - num of visits
P - move prob
Neural Network
p
v
Backprop & Expand
Q = (Q * N + v) / (N + 1)
N = N + 1
P = p
Upper Confidence Bound
U = c_puct * P * √N_parent / (N + 1)
Deterministic
choose action from state with greatest N
Stochastic
choose action from root state with distribution from N
Select Move
N=15
N=32

*Figure 3.* A scheme of Monte Carlo Tree Search. The algorithm first recursively searches for the optimal child nodes until a leaf node is reached. Then it creates one or more child nodes and selects one to run a simulated playout until a result is achieved. Based on the simulated results, it updates the current move sequence.

In MCTS, when the agent has to select an action, it will construct a search tree where the root is the current state, and run thousands of simulations, where each tree node stores $N$, $P$, $Q$, and $U$. $N$ represents the number of times the node has been visited, $P$ represents the move probability, $Q$ is the mean $Q$-value, and $U$ is the upper confidence which is a function of $Q$, $P$, and $N$. In each simulation, the agent selects the action with the highest $U$ until it has reached a leaf node. At which point, the leaf state will be passed to a policy evaluation function to predict the probability of next valid moves and the value of this state. The algorithm then propagates the value recursively up to the ancestor of the tree and updates the $Q$-values. After the simulations, we then either deterministically select the move from the root state with the highest $N$, or stochastically chooses from a distribution based on $N$ if we want more exploration. The formula for a child node's $U$ is below, where $c_{puct}$ is a hyperparameter that controls the degree of exploration.

$$U = Q + c_{puct} \cdot P \cdot \frac{\sqrt{N_{\text{parent}}}}{1+N}$$

There are a few distinctions between MCTS we implemented and the version shown in Alpha Go papers. The first modification we made is to accommodate our game model. In Zertz, players are compelled to make a capture move when possible and capture moves can be chained. We introduces a node attribute $s \in \{1, -1\}$ to indicate the value of the player responsible for selecting the actions amongst children nodes at the state represented by that node. If player 1 is the current player at that node then $s = 1$ and if player 2 is the current player then $s = -1$. In the case that there is a chain capture, $s$ shall remain unchanged between subsequent nodes in the tree.

As MCTS is the major bottle neck of our algorithm efficiency, we applied an optimization technique while generating self-play data. After the agent has selected a move after running number of MCTS simulations and the game has not ended, we move the root to the node corresponding to the selected move instead of reseting the tree. This way the new root would contain prior information from previous simulations. To make this optimization work, the tree uses the value of $s$ from the parent node to correctly calculate $U$ for its children by multiplying $s_{\text{parent}} \cdot Q$. This way if both the average $Q$-value and $s$ are negative the action selected to maximize $U$ will still be correct. Our modified calculation for $U$ that incorporates information about the current player is given below.

$$U = s_{\text{parent}} \cdot Q + c_{puct} \cdot P \cdot \frac{\sqrt{N_{\text{parent}}}}{1+N}$$

As AlphaZero completely relies on self-generated data to learn, well distributed data and avoiding overfitting is necessary to train a good policy. Measures that we have taken to achieve this comes in two part: making enough exploration in self-play and tune regularization in neural network. The later will be discussed in subsequent section. We adopted temperature and Dirichlet noise while selecting moves. The temperature is responsible for diminishing the bias towards large $N$ while selecting a move from the Monte-Carlo tree. It enables the agent to select a move stochastically with exponentiated $N$: $P(s,a) = N(s,a)^{1/\tau} / \sum_b (N(s,b)^{1/\tau})$, where $\tau$ is the temperature. We initialize temperature to a value proportional to the average game length and let it decay until 0 as the episode length progresses. Additionally, Dirichlet noise is applied independently from temperature and it encourages the agent to explore for a fraction of $\epsilon$ of the time according to the Dirichlet distribution: $P(s, a) = (1 - \epsilon)P_a + \epsilon\eta$, where $\eta \sim Dir(\alpha)$ and $\alpha$ is set inversely proportional to the number of available actions at state $s$.

Another way to avoid overfitting on self-generated data is to perform data augmentation. Since the Zertz game state is symmetrical, we generate 8 symmetries at every sate encountered during an episode and add them to the training examples: 4 symmetries from rotation and mirroring and times 2 for the opponent's state since both players play with the same pieces. While running MCTS simulations and invoking the neural network for policy prediction, we pass a randomly sampled symmetrical game state from the 8 symmetries to the neural network for prediction to limit the effect of bias.

### 3.3. Neural Networks

The input to the neural networks is the current game state $s_t$. As mentioned in Section 3.1, $s_t$ is an $x \times y \times 31$ stack image. The model we ultimately use is a 10-layer convolutional neural network, and each layer consists of the following modules: A rectifier nonlinearity; Batch normalization (Ioffe & Szegedy (2015)); A convolution of $\{16, 32, 64\}$ filters of kernel size $3 \times 3$ with stride 1. The output of the convolutional layers is flattened and passed to two dense layers, which consists of the following modules: A dropout layer with dropout rate 0.1; A rectifier nonlinearity; Batch normalization; A fully-connected layer with 1024 and 512 output units, respectively. The output of the dense layers is passed to two separate heads to compute value and policy. The policy head is a fully-connected layer with $S_c + S_p$ output units followed by a softmax activation, where $S_c$ is the size of the capture actions and $S_p$ is the size of the placement/removal actions. In our setting,

$$S_c = x \times y \times 6$$
$$S_p = xy \times (xy + 1) \times 3$$

$x$ and $y$ are the height and width of the board. The value head is a fully-connected layer with a scalar output followed by a tanh activation. We define the loss as follows

$$\mathcal{L}(\theta) = \frac{1}{n}\sum_{i=1}^{n}\left[(V_{s_i} - \hat{V}_{s_i})^2 + \sum_{a \in A} -\mathbf{1}(a_{true} = a) log P(a)\right] + \alpha\|\theta\|_2^2,$$

where $\alpha$ is the regularization coefficient. Figure 4 shows an overview of the structure of the neural networks we used.

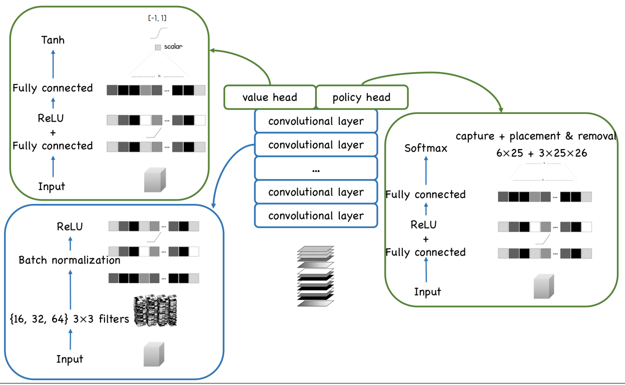

*Figure 4.* The structure of the CNN

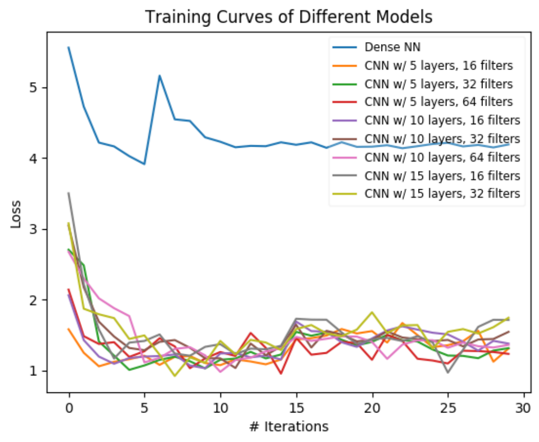

*Figure 5.* The training curve of the dense neural network and 8 CNNs with 5, 10, 15 convolutional layers and $\{16, 32, 64\}$ filters. CNNs perform similarly, while strictly better than the dense neural network

## 4. Experiment Results

### 4.1. Experimental Setup

We train our model described in Section 3.3 for 30 iterations. Each iteration on the 19 ring game board starts with the generation of 750 episodes of self-play data which results in approximately 60,000 example turn tuples. These examples are then randomly shuffled and added to a queue of size 150,000 which is used as a replay buffer. Each action is generated after 40 simulations of MCTS. The model is then trained on these examples for 50 epochs with a batch size of 1000 from the examples contained within the replay buffer. When training on the full-sized 37 ring game board, the number of episodes generated is 500 (approximately 120,000 example turns), the replay buffer is set to 250,000,

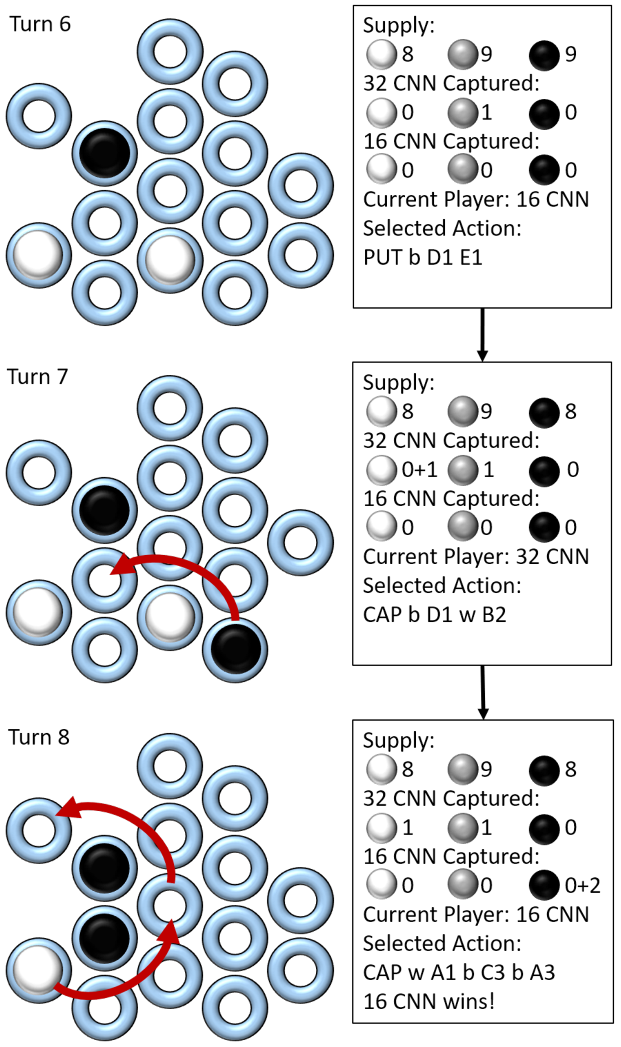

*Figure 6.* The final three turns of game five between the 32 filter CNN (player 1) and the 16 filter CNN (player 2). On turn 6, the 16 filter CNN sacrifices a white marble to force a capture for the 32 filter CNN (turn 7) that puts the board in a state where a double capture is possible (turn 8) that will win the game.

and number of simulations per action is 25. We set the regularization coefficient $\alpha$ to 0.0002. We employ an Adam Optimizer to optimize the objective function with a learning rate of 0.01. The learning rate decays by a factor of 10 after half of the iterations have completed. Dirichlet noise was added to the prior probabilities in the root node with the $\alpha$ parameter set proportional to the inverse of the sum of the number of legal moves.

In fixing the neural network structure, we compared dense neural networks and convolutional neural networks (CNNs) with different number of layers and filters per layer, as shown in Figure 5. CNNs consistently outperformed dense

*Table 1.* Games between AIs and Human

| GAME | PLAYER 1 WINS |
|---|---|
| $10 \times 16$ CNN VS. $10 \times 32$ CNN | 4/8 |
| HUMAN VS. $10 \times 16$ CNN | 5/8 |
| HUMAN VS. $10 \times 32$ CNN | 4/8 |

neural networks at training loss and game play. We used CNNs with 10 layers and $\{16, 32\}$ filters per layer to train on the small game board. For the full-sized game board, we trained only one model with 10 hidden layers and 64 filters per layer to adjust to larger hypothesis space.

## 4.2. Playing with the Models

Since there is no existing AI playing Zertz on the 19 ring variation of the game board, we evaluated each model by making them play against each other and a human opponent. The number of simulations of MCTS for both agents was increased to 1000 and all actions were selected greedily for the purposes of evaluation. Even with these modified parameters our models selected an action within a few seconds. We played eight games between each combination of the 10 layers $\times$ 16 filters CNN, the 10 layers $\times$ 32 filters CNN, and Kevin (who plays the best among us). Table 1 shows the results of each match. The two CNNs achieved a similar level of skill to each other and were both capable of winning against a human opponent. The 10 layer $\times$ 16 filters CNN performed slightly worse against the human opponent because an exploitable weakness was identified during the course of the games. This weakness was caused by the 16 filters CNN favoring a specific board state that an observant human player could take advantage of to force a game win if the human player goes first. To further compare the performance of each AI we reviewed the game history for the AI vs AI games. Even though we selected the actions for both AIs greedily during evaluation the games were all unique. We attribute this randomness to the MCTS predicting the policy and and state value using a randomly sampled symmetrical state to the current game state. Although the first player is heavily favored in the 19 ring adaption of Zertz, both AIs managed to win one of the games where they went second. We believe that our model will continue to improve with more training examples, simulations, and longer training.

Figure 6 depicts a series of turns during one of the $10 \times 16$ CNN vs $10 \times 32$ CNN games. This shows how the 16 filter CNN was still capable of beating the 32 filter CNN even with the disadvantage of playing second and having less marbles captured. This demonstrates the potential of the AIs to come up with remarkable strategies not obvious to a human player.

Due to training time constraints, we were limited to training only one implementation of our model on the full-sized board using a 10 layer $\times$ 64 filter CNN. Despite good performance on the small board, our AI lost all the games against available online AI opponents and an average human player. We found MCTS to be a significant bottle neck in the training of a successful AI on the full game. Training on the full game board took approximately three days with the majority of the time spent generating self-play example games. Given the computational resource and the time we have, it's difficult to perform enough simulations in MCTS and we had to restrict the number of simulations per turn to 25 and the number of episodes generated to 500. More details are discussed in Section 5.

## 5. Conclusion and Discussion

In this project, we re-implemented AlphaZero from scratch to play the board game Zertz. We managed to successfully train our agent to beat us at a 19 ring variant of Zertz in half of the games solely on self-play data without human knowledge.

We encountered numerous challenges to adapting ZlphaZero to Zertz. One characteristic of Zertz that posed a challenge initially is the difference between placement and capture actions. Placement actions have a much larger action space, but are only possible if there is no valid capture. Capture actions, while having a smaller action space, do not always end the turn and can feature branches of different lengths. Our first implementation of the neural networks and loss function featured two policy heads, one for placement and one for capture, in order to closely model the policy. We also passed as input to our model a mask to specify if the loss function should use the cross entropy loss of the placement policy head or the capture policy head for each game state input. This architecture did not perform well during training and made it harder to predict both capture and placement actions due to the difference in action space sizes and the weights of the model being shared. In our final implementation we flatten and concatenate the capture and placement policies and use that in the loss calculation for the policy head. Although the output action space is large, the model was able to successfully learn.

Another challenge we encountered was the proper tuning of hyper-parameters to avoid overfitting and allow MCTS to explore different game states in the early game. It was difficult to initially set our model's hyperparameters due to the difference between Zertz and games previously tested with AlphaZero such as Chess and Shogi. Additionally, some of the methods employed by AlphaZero were not as well documented in DeepMind's published work. We found that early problems with overfitting in the model had the potential to compound in future iterations of training because the model is only trained on self-play generated examples. The examples generated with a model that overfits the training examples is likely to generate very similar training examples and continue to overfit the data. During testing we discovered the importance of Dirichlet noise in encouraging exploration during self-play and increasing the diversity of examples created. The adjustment of the regularization term and implementation of board state symmetries in both training and test time also further reduced our earlier problems with overfitting.

While the agent learns very well on the reduced board size, it encounters more difficulty generalizing to the regular size board of 37 rings. Digging into the training process, we discover high loss of $v$ even after dozens of iterations. This is an indication of erroneously learned network weights. We believe it is mainly due to the inefficiency of MCTS on our computer resources. With our MCTS parameters, the algorithm is rarely going to reach an end game state and instead relies heavily on the predictions from the neural network. This causes training examples to not be good representations of the true state value which continues to perpetuate the problem in further iterations of training. The network is essentially using its initially generated random weights to feed itself. To solve this, we will need to increase MCTS simulations to very high number, which renders it extremely slow on our computers with our current implementation.

The results of our agent playing against a human opponent are promising and we believe that with enough computational power and time, our model can surpass expert human play. An area of future work would be to improve the efficiency with which example episodes are created to make our model more efficient to train. The biggest challenge remains the runtime of MCTS. By rewriting the implementation of MCTS and the game simulation in a more computationally efficient language it would be possible to run many more simulations in the same amount of time. Additionally, the task of generating self-play episodes could be completely parallelized across multiple processing units which would scale the most computationally costly part of training by the number of machines available. One final improvement to the efficiency of game generation could be made to the action space representation for placement actions. The current action space scales poorly by the height and width of the board because it attempts to represent all possible combinations of marble placement and ring removal. By separating the placement action into two actions, a marble placement and a ring removal, it is possible to shrink the action space by a factor of $xy$ where $x$ is width and $y$ is height. This change would come at the cost of increasing episode length, but would simplify the policy head of the neural network and improve the simulations of MCTS.

## Contributions

K.Culberg, W.Hsu and A.Xu conceived the presented idea, investigated relevant papers together. K.Culberg implemented Zertz game interface and simulation. W.Hsu implemented MCTS and self-play game generation. A.Xu implemented all neural network models and main training loop. All group members contributed to high-level code development/debugging, gathered and discussed experiment results, and contributed to the write up of the final paper.

## References

Browne, Cameron B, Powley, Edward, Whitehouse, Daniel, Lucas, Simon M, Cowling, Peter I, Rohlfshagen, Philipp, Tavener, Stephen, Perez, Diego, Samothrakis, Spyridon, and Colton, Simon. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

Chaslot, Guillaume Maurice Jean-Bernard Chaslot. *Monte-carlo tree search*. PhD thesis, Maastricht University, 2010.

Gelly, Sylvain and Silver, David. Achieving master level play in 9 x 9 computer go. In *AAAI*, volume 8, pp. 1537–1540, 2008.

Heule, Marijn JH and Rothkrantz, Leon JM. Solving games: Dependence of applicable solving procedures. *Science of computer Programming*, 67(1):105–124, 2007.

Ioffe, Sergey and Szegedy, Christian. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pp. 448–456, 2015.

Silver, David, Huang, Aja, Maddison, Chris J, Guez, Arthur, Sifre, Laurent, Van Den Driessche, George, Schrittwieser, Julian, Antonoglou, Ioannis, Panneershelvam, Veda, Lanctot, Marc, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587): 484–489, 2016.

Silver, David, Hubert, Thomas, Schrittwieser, Julian, Antonoglou, Ioannis, Lai, Matthew, Guez, Arthur, Lanctot, Marc, Sifre, Laurent, Kumaran, Dharshan, Graepel, Thore, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017a.

Silver, David, Schrittwieser, Julian, Simonyan, Karen, Antonoglou, Ioannis, Huang, Aja, Guez, Arthur, Hubert, Thomas, Baker, Lucas, Lai, Matthew, Bolton, Adrian, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017b.

## Appendix

We present the logs of some representative games human played with the AI using the 16 filter CNN. Each line specifies an action in the game and preceded by 'PUT' for a placement action or 'CAP' for a capture action. The lowercase letter denotes the marble used in that action. The uppercase letter and number codes correspond to board positions, see Figure 1.

```
Game 2: AI first
AI:      PUT g A1 B4
Human:   PUT g D3 E3
AI:      PUT b B2 E1
Human:   CAP g A1 b C3
AI:      CAP g D3 g B2
Human:   PUT b C1 A1
AI:      PUT b A2 A3
(AI would have won the game this turn if
it put a gray marble instead of black)
Human:   CAP b A2 g C2
AI:      CAP b C1 b C3
Human:   PUT w A2 C5
AI:      PUT w D4 B3
Human:   PUT g D2 C1
AI:      PUT b B1 E2
Human:   PUT b D1 D3
AI:      PUT g C2 C4
AI wins
```

```
Game 4: AI first
AI:      PUT g E3 D1
Human:   PUT b A3 E1
AI:      PUT b B1 E2
Human:   PUT g A1 C1
AI:      PUT b C4 D4
Human:   PUT w A2 C5
AI:      PUT w C2 D2
Human:   PUT g B4 D3
AI:      PUT w C3
Human:   PUT b B3
AI:      CAP g B4 b B2
Human:   CAP b B1 g B3
Human wins
```

```
Game 5: Human first
Human:   PUT b B1 B4
AI:      PUT g C4 A3
Human:   PUT b D1 A1
AI:      PUT w E3 C5
Human:   PUT g A2 E1
AI:      PUT g C3 B3
Human:   CAP g C4 g C2
AI:      CAP b D1 g B2
Human:   CAP g A2 b C2
AI:      CAP b B1 g D2
AI wins
```