Training DeepMind's OpenSpiel AlphaZero Algorithm to Play Clobber

Christian Jans
University of Alberta
Edmonton, Alberta T6G 2R3
cjjans@ualberta.ca

Abstract: Google DeepMind's AlphaZero has revolutionized the world of game AI. Through pure self-play, the algorithm can achieve superhuman performance in games such as chess, shogi, and Go. However, accomplishing this requires an enormous amount of computing resources, the likes of which are not available to most researchers. Additionally, the multitude of AlphaZero hyperparameters and their effect on performance can be difficult to predict without undergoing numerous time-consuming training sessions. This paper recounts the process of training the AlphaZero algorithm in Google DeepMind's OpenSpiel framework on various board sizes of the game Clobber during Summer 2020. The aim is to provide a starting point for those looking to train the AlphaZero algorithm by addressing elements such as the encoding of the board, neural network architecture, and hyperparameter values. Inferences about these components are based on data available in a Google Doc located here: http://bit.ly/3qLwsHk.

1 Introduction

At a high level, AlphaZero consists of two main components: a Monte-Carlo Tree Search (MCTS) and a dual-headed neural network. The neural network takes as input an encoded version of the game state, and outputs a *value* scalar and *policy* vector. The value output of the neural network estimates the advantage of a given game state from the current player's perspective. It is a continuous value in the range -1 (loss) to 1 (win). The policy output consists of a vector of probabilities for each available action from the given game state. From a given state, the algorithm undergoes a neural-network-backed MCTS to obtain experience in the game through multiple simulations. During training, these experiences are then compiled into a replay buffer to be used later to learn from. Hence, an iteration of the AlphaZero training loop essentially consists of self-play and continuous evaluation, followed by learning, in which the neural network learns from the experiences gathered during self-play. For more detail, a working implementation can be found in Google DeepMind's OpenSpiel framework [6] (the framework we used to train our AlphaZero agents). Finally, as a note, the `Courier New` font will be used to refer to the hyperparameters of the framework's AlphaZero implementation.

Clobber [1] was invented by Michael Albert, J.P. Grossman, and Richard Nowakowski in 2001. It is classified as a two-player, combinatorial board game. That is, it has no randomization mechanism during play, and all players know everything about the current state of the game. The game is played on an $m \times n$ checkerboard starting with white pieces on white squares, and black pieces on black squares. The game can also be played with the symbols "x" and "o" in place of the colours. Players are then each assigned a colour or symbol and alternate turns moving one of their pieces on top of an adjacent opponent piece. This opponent piece is then removed from the board. The winner is the last person to play. Little is known about what constitutes "good" play at the beginning of the game, especially on larger board sizes [13]. As a result, hyperparameters that

have the ability to scale and also impact performance were of great interest in developing AlphaZero players for larger and larger board sizes.

Our training of the AlphaZero algorithm started off small with 3×3 Clobber boards, but gradually continued on to 4×4, 5×5, and 5×6. At each of these board sizes, hyperparameter values were experimented with to determine which hyperparameters affected the evaluation and learning of the AlphaZero player the most. We define a "small board" to have an area of 16 cells or less, such as sizes 3×3 or 4×4; and a "large board" as any board with an area larger than 16 cells such as 5×5, 5×6, or larger. The ultimate goal of this project was to scale up training to a board size of 10×10. Unfortunately, due to a lack of experimentation and testing, the 10×10 results are not discussed in this paper. Additionally, a few 8×8 board experiments were conducted, but are not discussed for the same reason.

The Python AlphaZero implementation in OpenSpiel had the ability to utilize GPUs, however, testing indicated that the use of GPUs hindered the performance of the algorithm. Reasons for this are still unknown. As a result, our hardware requirements focussed on the number of CPU cores and total amount of memory. For the small boards and the 5×5 boards, we used a machine with 8 CPU cores and 4 GB of memory to conduct the training. Extra hardware obtained from Compute Canada was used for training on board sizes larger than 5×5. With these machines, we had access to 48 CPU cores and up to 128 GB of memory.

Finally, similar projects have also attempted to minimize the training time of an AlphaZero player while maintaining its performance through the experimentation of hyperparameter values. However, this has mainly been tested with games whose boards are not scalable, such as Connect 4 [11], or scalable games are chosen, but the board size is fixed, such as 6×6 Othello [12]. Results from these studies indicate that there is certainly room for hyperparameter optimization in the AlphaZero algorithm. Despite this, there is unfortunately minimal other literature on the subject.

2 Performance Analysis

The OpenSpiel framework comes with a Python script that allows a graphical analysis to be performed on an AlphaZero training session. While training, the algorithm continually updates a file called "learner.jsonl" that contains data about each step of the current training session. The data in this file can then be graphed using the aforementioned Python script. Once run, this script creates a window with 12 graphs describing the performance in terms of both computing and the AlphaZero player. To analyze the performance of the AlphaZero player, we focussed our attention on two of these 12 graphs, specifically the "Training loss" graph (shown in Figure 1) and the "Evaluation returns vs MCTS+Solver with x10^(n/2) sims" graph (shown in Figure 2). Both graphs shown in the figures come from the same training session on a board size of 5×5.

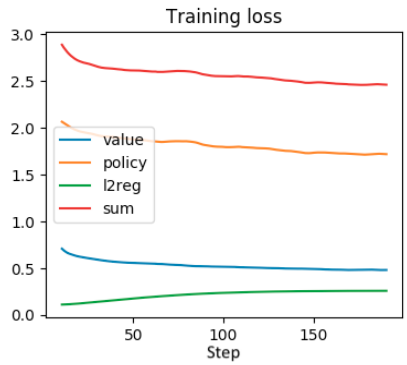


Figure 1: The loss graph of an AlphaZero training session. Total (sum), policy, value, and regularization loss are shown.

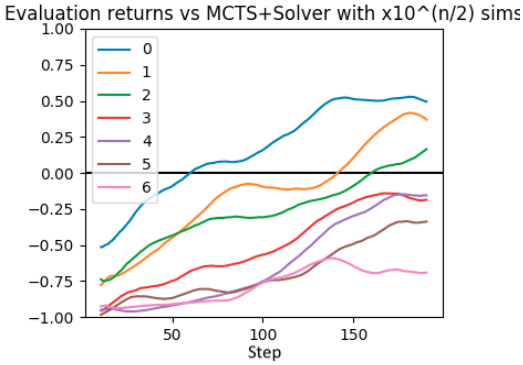


Figure 2: The evaluation graph of an AlphaZero training session. The legend depicts the colours of each difficulty level.

Loss Graph. The loss graph shows the total (sum) loss, policy loss, value loss, and regularization loss of the neural network after each training step. As a result, as the training steps increase, it is best to see a decrease in loss (particularly in the total, policy, and value losses) as this implies that the network is able to more accurately predict the value and policy of a given game state. Figure 1 shows an example of a loss graph that steadily decreases.

Evaluation Graph. The evaluation graph depicts how well the AlphaZero player plays against an MCTS player. Over the course of training, the AlphaZero player continually plays an MCTS player of varying difficulty. The difficulty of the MCTS player is determined by how many simulations it is allowed to run from the current game state of the game. For each move, the AlphaZero player plays by performing `max_simulations` simulations from the current game state. It is combated by an MCTS player of a certain integer level, n, whose simulation count, s, is determined by $s = \text{max\_simulations} \times 10^{n/2}$. Therefore, as the level of the MCTS player increases, so does its simulation count. When it is the MCTS player's turn to play, it conducts s simulations from the current state to determine its move. At each level, the AlphaZero player is given the opportunity to play both first and second so that there is no inherent player advantage. A win for the MCTS player is tabulated as a -1 outcome, and a win for the AlphaZero player is tabulated as a +1 outcome. A buffer of size `evaluation_window` is kept for each difficulty level. These buffers are contained in a list, and the outcomes for each game at a certain level are put into their respective buffer. After each training step, the average value of the buffer is calculated for each difficulty level and plotted on the evaluation graph. As a result, at the beginning of training when the AlphaZero player has not learned enough to play well, it should be expected that the evaluation graph will show negative average outcomes for all difficulty levels. However, as the training progresses, the average outcomes should increase, and the trends for each difficulty level should also diverge. That is, as training progresses, lower difficulty levels should see a higher average outcome than those of higher difficulty. This is because it should be easier for the AlphaZero player to win against an MCTS player undergoing fewer simulations. Notice that this divergence and gradual increase is what is approximately observed in Figure 2.

3 Model Type

For all of our experiments, we used the dual-headed residual neural network available in the OpenSpiel framework. The OpenSpiel framework has three types of dual-headed neural network architectures available for AlphaZero: a residual network, a convolutional network (without residual connections), and a multilayer perceptron (MLP or fully-connected) network. Each

network can be configured with a variably sized input and output, and can also be set with a specific number of layers and filters (in the case of the residual and convolutional networks) or layers and perceptrons (in the case of the MLP network). The residual neural network follows the same basic architecture as described in AlphaGo Zero [10], and the implementation of these three neural networks can be found in the OpenSpiel framework. Notice that the game of Cobber inherently relies on the spatial information between pieces on the board. This can simply be deduced from the rules of the game: a piece cannot be moved unless there is at least one adjacent opponent piece. A multitude of results in various areas of research have previously shown that convolutional layers enhance a neural network's ability to detect this spatial information in an input [5, 8, 14]. Therefore, we decided to not use the MLP network. When choosing between the convolutional and residual neural networks, we chose the residual network due to its ability to combat the Vanishing Gradient Problem [4] in deep neural networks [3].

4 Clobber Implementation

The Clobber implementation in the OpenSpiel framework was built in C++. It allows for a variable number of rows and columns, and an option to invert the colours on the board. In the framework, the player representing the character "o" is Player One and always plays first. Hence, Player Two is represented by "x". OpenSpiel requires each game's state class to define an "Observation Tensor" function. This function returns an encoding of the current game's state that can then be passed to a neural network as input. The encoding of a Clobber game state went through a revision over the course of training for the different sized boards.

First Board Encoding. For the AlphaZero players on the 3×3, 4×4, and some 5×5 boards, we encoded the game in a tensor of rank three. For an $m \times n$ Clobber board, the tensor consisted of three $m \times n$ planes. The first plane consisted of 1's where all of Player One's pieces were, and 0's elsewhere. The second plane consisted of 1's where all of Player Two's pieces were, and 0's elsewhere. Finally, the third plane consisted of 1's where all the empty board cells were, and 0's elsewhere. An example encoding can be found in Appendix A. Notice however that with this encoding, it is difficult to deduce the value of a state. For example, consider a mid game state of an arbitrarily sized board with only two pieces and these pieces are adjacent. Given the encoding of this board, it is impossible to predict the value of this state without knowing whose turn it is. Therefore, some sense of whose turn it is must be added to the encoding. Additionally, keeping a similar perspective of the board was also added. That is, the network always views its own pieces on the same plane of the tensor, no matter if it is playing as Player One or Player Two.

Second Board Encoding. Some 5×6 boards and all boards larger than 5×6 followed this new encoding that depicts the board from a consistent point of view, and also provides information on which player's turn it is. For a Clobber board of size $m \times n$, this was accomplished once again through a tensor of rank three consisting of three $m \times n$ planes. The first plane consisted of 1's where the current player's pieces were, and 0's elsewhere. The second plane consisted of 1's where the opponent's pieces were, and 0's elsewhere. Finally, the third plane consisted of all 1's if it was Player One's turn to play, otherwise it consisted of all 0's. An example of this encoding can be found in Appendix A. This is similar to the encoding used in AlphaGo Zero [10], which is why it was chosen. The only difference is that there are no planes containing previous states of the board in the Clobber encoding. Variations of this type of encoding have also proved useful in chess and shogi [9].

5 Hyperparameters

A complete list of available hyperparameters for OpenSpiel's Python AlphaZero is available in Appendix B. Descriptions and values used in our experiments are also given for each. Note that not all hyperparameters affected the speed of training, the performance of the AlphaZero player, or the scalability of the performance of the AlphaZero player. These hyperparameters are not in bold font in the table and will not be discussed. Also, some hyperparameters were not tweaked due to a lack of time, these are also not bolded. In total, eight hyperparameters were experimented with and will be discussed in this section.

Replay Buffer Ratio. We define the Replay Buffer Ratio as the value of the ratio of the `replay_buffer_size` to the `replay_buffer_reuse`. The replay buffer size hyperparameter determines the number of experiences the replay buffer can hold. An experience consists of a state, action, and a reward obtained from taking that action in that state. A greater value for the replay buffer size allows more of these experiences to be held in the agent's memory, and the agent can then learn from a more diverse set of experiences. The replay buffer reuse parameter specifies how long the previous experiences are held in the replay buffer. That is, during each iteration of the self-play portion of the AlphaZero training loop, the agent will gather a certain number of experiences and put these experiences into the replay buffer. The number of experiences gathered is exactly the value of the Replay Buffer Ratio. Hence, starting from an empty replay buffer, it takes `replay_buffer_reuse` iterations of the training loop to fill the replay buffer. Notice that the larger the Replay Buffer Ratio, the greater the number of experiences are added to the replay buffer during self-play each training iteration (training step). Additionally, note that on average, an experience will remain in the replay buffer for `replay_buffer_reuse` training iterations. We found that a large Replay Buffer Ratio improves performance drastically as long as the replay buffer reuse is small enough. Explained intuitively, a large Replay Buffer Ratio ensures a greater diversity of experiences are added to the replay buffer every training iteration, and a small replay buffer reuse ensures that old experiences, that are potentially incorrect or not useful, are not learned from anymore. However, it was also found that the Replay Buffer Ratio is heavily proportional to the amount of time a training iteration takes. We found that effective replay buffer reuse values never exceeded 4, and we scaled the replay buffer size with the board size we were training on. For smaller boards (3×3 and 4×4) we used a replay buffer size of 2^{14} and a replay buffer reuse of 4. For large boards, the Replay Buffer Ratio was greater than or equal to 2^{14}; maintaining a replay buffer reuse of no more than 4.

Actors & Evaluators. To perform self-play and evaluation during training, the OpenSpiel Python AlphaZero algorithm spawns actor and evaluator Python processes respectively. The number of actor and evaluator processes spawned can be controlled using the `actors` and `evaluators` hyperparameters. As the number of actors increase, the speed at which self-play games can be simulated also increases. This results in less time per training step. Similarly, as the number of evaluators increases, more games can be played against the MCTS players. Although this will not increase the speed of training, it will provide more accurate results on how well the AlphaZero player is performing throughout training. For our experiments, the sum of the actors and evaluators was equal to the number of CPU cores available. Additionally, the number of evaluators never exceeded the number of actors. Often, the number of evaluators was kept in the range one to four.

Simulation Count. The number of simulations from a state performed by the AlphaZero agent during training is determined by the `max_simulations` hyperparameter. Intuitively, the more simulations performed by the AlphaZero agent at each state, the better its policy approximation

will be. However, an increase in the number of simulations per state requires more time. For smaller boards (3×3 and 4×4), we kept a value of 20 for this hyperparameter. However, for larger boards (5×5 and greater), this number was almost always greater than or equal to 100.

Temperature Drop. To increase exploration during self-play, the OpenSpiel Python AlphaZero algorithm will sometimes choose a random action following the policy distribution obtained from undergoing MCTS instead of the best action obtained from undergoing MCTS. This behaviour is controlled by the temperature drop hyperparameter. If the total number of moves played in the self-play game is less than the value of the temperature drop, then a random action following the policy distribution is performed. Otherwise, the best action obtained from MCTS is performed. The temperature drop value is determined by the `temperature_drop` hyperparameter. For boards larger than 5×6, we based this value on the average game length of randomly simulated Clobber games of the desired board size. To approximate a value for the temperature drop we looked at previous AlphaZero implementations and the games they played. From the temperature drop values in the Connect 4 analysis [11] (Connect 4, an average game length of 36 moves [2] using a temperature drop of 10 for their AlphaZero), DeepMind's AlphaGo Zero [7] (Go, with an average game length of 150 moves [2] using a temperature drop of 30), and the recommended hyperparameter value of the temperature drop for Tic-Tac-Toe in the OpenSpiel framework (Tic-Tac-Toe, an average game length of 9 moves [2], using a temperature drop of 4), a linear relationship between the temperature drop and the average game length was devised through simple linear regression. The relationship was calculated to be:

$$T = 0.182N + 2.85,\ R^2 = 0.998$$

Where T is the temperature drop value and N is the average number length of the game. If the interpolated number was not a whole number, we would round to the nearest whole number. For example, 5×5 Clobber has an average game length of 15.7 moves, resulting in a value of approximately 5.7 for the temperature drop (which is rounded to 6).

Neural Network Depth. The number of residual layers (depth) in our residual neural network was determined as a function of the number of rows (R) and columns (C) of the board. For small board sizes, the range of the depth (D) of the neural network was kept within two of the maximum between the number of rows and number of columns, that is:

$$D = \max(R, C) + \delta,\ \delta \in \{-2, -1, 0, 1, 2\}$$

For large board sizes, we attempted to approximate the network depth based on AlphaGo Zero [10]. In their experiments, the smaller network had a depth of 20 for a Go board size of 19×19. Hence, the depth of the neural network for larger Clobber boards was trivially set as:

$$D = \max(R, C) + 1$$

Although in practice, values would range within ±1 of the maximum. The depth of the AlphaZero neural network can be set using the `nn_depth` parameter. With an increase in depth, there is an increase in the amount of time it takes to perform a training step. This is because neural network inference, backpropagation, and gradient descent take more time with more network parameters.

6 Conclusion

This paper discussed our process experimenting with AlphaZero, particularly the OpenSpiel version of the algorithm. We discussed neural network architecture, providing reasons for the choice of a residual architecture over purely convolutional-based or fully-connected networks. Additionally, we touched on the use of two ways to encode a board as a tensor for the neural network. Ultimately settling with a representation that provided a representation from a consistent perspective that contains all the information of the game in the given state. Finally we examined some of the hyperparameters of the algorithm, particularly those that had the most effect on the

time it takes to train the algorithm, and those that most affected the performance of the trained agent. It was inferred that our self-defined Replay Buffer Ratio hyperparameter had the greatest effect on both these metrics. We hope that the information contained in this report and the accompanying data can aid future researchers, and provide a starting point for training their own AlphaZero, or AlphaZero-like algorithm.

Acknowledgements

This research was enabled in part by support provided by WestGrid and Compute Canada. We also acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC). Finally, I would also like to thank Dr. Martin Müller and Dr. Ting-Han Wei for their support and mentorship with this project.

References

[1] Albert, M.; Grossman, J.; Nowakowski, R.; et al. 2005. An introduction to Clobber. *Integers* 5(2): A1.

[2] Allis, V. 1994. *Searching for Solutions in Games and Artificial Intelligence* (Ph.D. thesis). University of Limburg, Maastricht, The Netherlands. ISBN 90-900748-8-0.

[3] Kaiming He, Xiangyu Zhang, Shaoqing Ren, & Jian Sun. (2015). Deep Residual Learning for Image Recognition.

[4] Hochreiter, S. (1998). The Vanishing Gradient Problem During Learning Recurrent Neural Nets and Problem Solutions*International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems, 6*, 107-116.

[5] Krizhevsky, A., Sutskever, I., & Hinton, G. (2012). ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems* (pp. 1097–1105). Curran Associates, Inc..

[6] Lanctot, M.; Lockhart, E.; Lespiau, J.-B.; et al. 2019 OpenSpiel: A Framework for Reinforcement Learning in Games. *CoRR* abs/19008.09453.

[7] Nair, S. (2017, December 29). *A Simple Alpha(Go) Zero Tutorial*. Stanford. https://web.stanford.edu/~surag/posts/alphazero.html

[8] Silver, D., Huang, A., Maddison, C. *et al.* Mastering the game of Go with deep neural networks and tree search. *Nature* 529, 484–489 (2016). https://doi.org/10.1038/nature16961

[9] Silver, D.; Hubert, T.; Schrittwieser, J.; et al. 2017. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. *CoRR* abs/1712.01815.

[10] Silver, D.; Schrittwieser, J.; Simonyan, K.; et al. 2017. Mastering the game of Go without human knowledge. Nature, 550, 7676 (Oct. 2017), 354-359.

[11] Soh, Wee Tee. (2019, April 13). *From-scratch Implementation of AlphaZero for Connect4*. Towards Data Science. https://towardsdatascience.com/from-scratch-implementation-of-alphazero-for-connect4-f73d4554002a.

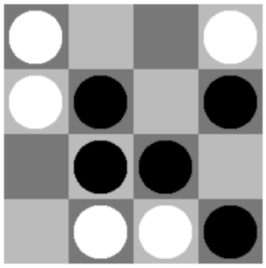
[12] Hui Wang, Michael Emmerich, Mike Preuss, & Aske Plaat. (2019). Hyper-Parameter Sweep on AlphaZero General.

[13] Willemson, J.; and Winands, M. 2006. MILA Wins Clobber Tournament. *ICGA Journal* 28: 188-190. doi:10.3233/ICG-2005-28316.

[14] Yamashita, R., Nishio, M., Do, R.K.G. *et al.* Convolutional neural networks: an overview and application in radiology. *Insights Imaging* 9, 611–629 (2018). https://doi.org/10.1007/s13244-018-0639-9

A Board Encodings

The following 4×4 board shows a Clobber state, mid-game. We assume that it is Player One's (white's) turn to play. Shown are the encodings of this state as tensors in a Python list format according to the first and second types of board encodings.



First Board Encoding

```
[[[1, 0, 0, 1],
  [1, 0, 0, 0],
  [0, 0, 0, 0],
  [0, 1, 1, 0]],

 [[0, 0, 0, 0],
  [0, 1, 0, 1],
  [0, 1, 1, 0],
  [0, 0, 0, 1]],

 [[0, 1, 1, 0],
  [0, 0, 1, 0],
  [1, 0, 0, 1],
  [1, 0, 0, 0]]]
```

Second Board Encoding

```
[[[1, 0, 0, 1],
  [1, 0, 0, 0],
  [0, 0, 0, 0],
  [0, 1, 1, 0]],

 [[0, 0, 0, 0],
  [0, 1, 0, 1],
  [0, 1, 1, 0],
  [0, 0, 0, 1]],

 [[1, 1, 1, 1],
  [1, 1, 1, 1],
  [1, 1, 1, 1],
  [1, 1, 1, 1]]]
```

B Table of Hyperparameters

Hyperparameter	Description	Value
Learning Rate (`learning_rate`)	The learning rate used for the neural network.	0.01 for small boards; 0.001 for large boards
Weight Decay (`weight_decay`)	The weight decay of the L2 regularization loss used in the neural network.	0.0001
Training Batch Size (`train_batch_size`)	The number of observations learned on for each weight	128 for small boards; 128, 256, or 1024 for large

	update of the neural network.	boards
Replay Buffer Size (`replay_buffer_size`)	The number of observations the replay buffer can hold.	2^{14} for small boards; 2^{16}, 2^{17}, 2^{18}, or 2^{19} for large boards
Replay Buffer Reuse (`replay_buffer_reuse`)	The number of training iterations that a given observation will remain in the replay buffer on average.	4 for small boards, 2 or 3 for large boards
Maximum Training Steps (`max_steps`)	The maximum number of training iterations that will occur before the program exits. Zero can be entered for an indefinite number of iterations.	0
Checkpoint Frequency (`checkpoint_freq`)	The rate at which the AlphaZero neural network will be saved based on the number of training iterations.	Varied depending on how often we wanted to save the neural network model
Actors (`actors`)	The number of processes spawned that undergo self-play. They collect the experience observations and put them in the replay buffer.	Varied depending on the number of CPUs available
Evaluators (`evaluators`)	The number of processes spawned that undergo evaluation. They play the AlphaZero algorithm against MCTS players, recording the results of the games.	Varied depending on the number of CPUs available
Evaluation Window (`evaluation_window`)	The number of evaluations to maintain in the evaluation buffer per level of MCTS opponent. Evaluation results are the average of the results in these buffers.	50 for small boards; 100 for large boards
Evaluation Levels (`eval_levels`)	The number of levels of difficulty of the MCTS players that the AlphaZero player will play against during evaluation.	7
Exploration Constant (c_{PUCT}) (`uct_c`)	A value proportional to the degree of exploration during MCTS.	1 for small boards; 2 for large boards
Maximum Simulations (`max_simulations`)	The number of simulations done by the AlphaZero algorithm from each state (on each turn).	25 for small boards; 100, 150, or 200 for large boards
Dirichlet Alpha (`policy_alpha`)	The parameter of the Dirichlet Distribution used to add random noise to the available actions during MCTS.	0.25 for small boards; 1 for large boards
Policy Epsilon (`policy_epsilon`)	The extent to which the random noise of an action in MCTS is added to the predicted prior of the action.	1 for small boards; 0.25 for large boards

Temperature (`temperature`)	The neural network policy output is raised to the reciprocal of this value.	1
Temperature Drop (`temperature_drop`)	During training, moves are randomly selected until the total number of moves played exceeds this value. Then, the best action computed is played.	4 for small boards; 10 for 5×5 and 5x6 boards; interpolated for any largerboards
Neural Network Model Type (`nn_model`)	The type of neural network model used for the algorithm. Fully-connected layer-based, convolutional layer-based, and residual layer-based are available.	The residual neural network architecture, a value of “resnet”
Neural Network Width (`nn_width`)	The number of filters or perceptrons in each intermediate layer of the convolutional-based or MLP neural network respectively.	128
Neural Network Depth (`nn_depth`)	The number of intermediate layers in the neural network.	2 for small boards; within ±1 of the larger between the number of rows and number of columns for large boards