Gomoku Agent Based on AlphaZero

Yaoyu Kang, Yufei Wu, Xingyuan Liu
School of Electronic Information and Electrical Engineering
Shanghai Jiao Tong University

***Abstract*—This study delves into Gomoku agent based on AlphaZero, aiming to achieve high winning rates against pure Monte Carlo Tree Search. Besides the basic architecture, we explore various models and structures, including residual blocks, playout cap randomization, global pooling, global attention residual block and boundary enhancement. Results indicate that our best model can beat pure MCTS with 3800 simulations.**

***Index Terms*—MCTS, AlphaZero**

I. Introduction

The research of computer chess is long-standing. Efforts have been made into the game of Go, Shogi, NoGo and so on. In this work, we pay attention to Gomoku with a chessboard of size 9×9.

We found out that Gomoku in some way is very similar to Go, thus neural networks like AlphaZero [6] may suit it well. Gomoku's rules are translation invariant, which means the rules remain the same everywhere on the chessboard. Plus, it's rotationally and reflectively symmetric. These characteristics make it easier to complete data augmentation. Moreover, the end of the game can only be victory or failure. All these features fit AlphaZero well. Hence, we plan to modify and improve based on AlphaZero.

Pure Monte Carlo Tree Search is applied as a baseline in this work. All implemented models are tested by playing against pure MCTS of different number of simulations.

II. Basic Architecture

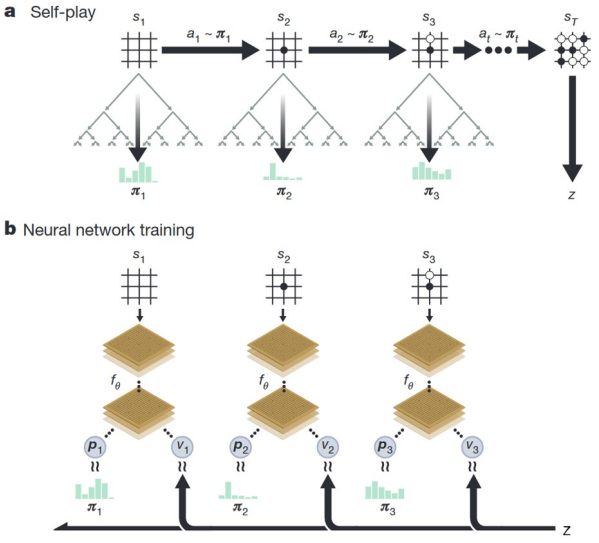


Fig. 1. Architecture of AlphaZero, from the paper of AlphaGoZero [7].

We implemented AlphaZero [6] with reference to a github repository [4] as the basic architecture of the agent, mainly including Monte Carlo Tree Search and Neural Network. The illustration of the architecture is displayed in Fig. 1.

A. State Representation

The size of state is 4 times as much as the size of the chessboard. Hence, the shape of state is $4 \times 9 \times 9$ in this setting.

The first layer of state represents all locations of current player's chess pieces. The second layer displays all locations of opponent player's chess pieces. The last move is marked in the third layer, and the color of chess pieces used by the current player is shown in the fourth one.

B. Neural Network

Neural network in this agent is designed to output both values of a potential next step and a policy guiding next step. Instead of using two separate neural networks like in AlphaGo [5], AlphaZero combines the two functions into a single neural network with two heads.

Considering Gumoku is a relatively easy problem, we use convolutional neural network at first. Three convolutional layers are applied as common layers. In the policy head, an additional convolutional layer and a fully-connected layer are used. We also employ a convolutional layer and two fully-connected layers in the value. Details of the neural network can be seen in Table I.

TABLE I
Details of Architecture of the Neural Network

layer	depth	kernel	stride	pad	activation
common_conv1	32	3×3	1	1	ReLU
common_conv2	64	3×3	1	1	ReLU
common_conv3	128	3×3	1	1	ReLU
policy_conv	4	1×1	1	0	ReLU
policy_fc	81	-	-	-	LogSoftmax
value_conv	2	1×1	1	0	ReLU
value_fc1	64	-	-	-	ReLU
value_fc2	1	-	-	-	tanh

C. Monte Carlo Tree Search

During Monte Carlo Tree Search, self-play is applied to improve the agent by playing against itself.

First, we describe the steps during one simulation, which is guided by the neural network $(\boldsymbol{p}, v) = f_\theta(s)$. Every edge in

the search tree maintains a prior probability $P(s,a)$, a visit count $N(s,a)$ and an action value $Q(s,a)$. All simulations start from the root and iteratively selects moves that maximizes $Q(s,a) + u(s,a)$ until a leaf node s' is reached, where $u(s,a) \propto \frac{P(s,a)}{1+N(s,a)}$. Then, $N(s,a)$ and $Q(s,a)$ are updated for all ancestors of s'.

In this process, triplets $(state, \boldsymbol{\pi}, r)$ are stored in a buffer to update parameters θ in the neural network f_θ, where $\boldsymbol{\pi}$ is probabilities of each move and $r \in \{1, -1\}$ depending on the result of the game. Parameters θ are updated to maximize the similarities between $\boldsymbol{p}$ and $\boldsymbol{\pi}$, and v and r.

III. Improved Algorithms

A. Residual Block

Deep learning networks have already been pointed out in paper "Deep Residual Learning for Image Recognition" [2]to introduce the problem of vanishing/exploding gradients, i.e., deeper networks will only work better than shallower ones. However, it has already been verified that such an idea is wrong on the CIFAR-10 dataset.

To solve this problem, residual networks are introduced. A residual network is in a sense constructing a constant mapping.

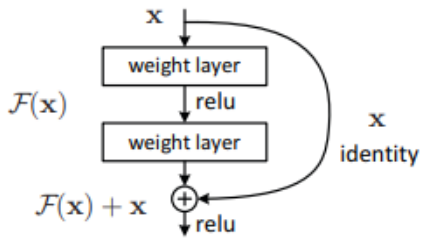


Fig. 2. Residual learning: a building block. [2].

It can be seen in Fig. 2 that it adds a path before connecting to the second layer of the activation function so that the relationship between the input and the output of the activation function changes from the original $H(x) = F(x)$ to $H(x) = F(x) + x$. It additionally adds the constant mapping of the output equal to the input.

The output equation with the addition of the residual network is shown as follows:

$$\mathbf{y} = \mathcal{F}(\mathbf{x}, \{W_i\}) + W_s\mathbf{x} \tag{1}$$

We can get the gradient formula for the residual network in backpropagation as follows:

$$\frac{\partial \boldsymbol{\varepsilon}}{\partial \mathbf{x}_l} = \frac{\partial \varepsilon}{\partial \boldsymbol{x}_L}\frac{\partial \mathbf{x}_L}{\partial \mathbf{x}_l} = \frac{\partial \boldsymbol{\varepsilon}}{\partial \mathbf{x}_L}\left(1 + \frac{\partial}{\partial \mathbf{x}_l}\sum_{i=l}^{L-1}\mathcal{F}(\mathbf{x}, \{\mathcal{W}_i\})\right) \tag{2}$$

We can see that the gradient of back propagation consists of two terms, the direct mapping of x and the results of multi-layer ordinary neural network mapping. We can see that even if the gradient of the new multi-layer neural network is 0, the gradient of the residual network is updated by one more "1". That means the path of identity, which we have defined earlier, is precise because of this shortcut. The gradient from the deep layer can pass directly and unimpededly to the upper layer, making the parameters of the shallow network layer wait for effective training, thus avoiding the problem of gradient dispersion.

B. KataGo

1) Playout Cap Randomization:

The basic AlphaZero performs each search by directly searching for the leaf node, then input the state of the leaf node into the network to obtain an estimated q-value and the probabilities of next action. We update the parameters of the network through these outputs. But there is a problem that the values of some intermediate parent nodes are determined by their subsequent strategy paths that seem to have the highest value, while some other strategy paths have not been fully explored.

To solve the problem, we introduced a new way of exploration as playout cap randomization. Besides of origin complete exploration, we also have local exploration (playout cap). When performing local exploration, we do not directly search for leaf nodes, but instead search for half of the average height of the search tree. The current obtained node state is used as input to the neural network, and the network output is used to update the parameters and potentially expand the node. During each simulation, there is a possibility of 0.5 that both local exploration and complete exploration will be selected (random), thus achieving a more comprehensive exploration of the search tree.

When implementing exploration, each time we need update the average height of the search tree to make convenience for playout cap. When we implement local exploration, some time although we don't want to reach leaf node, we still do that because the height is an average. As we have the limit that the number of playout should be 200, we need see this local exploration as full exploration and reduce the remaining number of playouts by one.

2) Global Pooling:

Another improvement in KataGo is adding global pooling. This enables the convolutional layers to condition on global context, which would be hard or impossible with the limited perceptual of convolution alone.

In KataGo, given a set of c channels, a global pooling layer computes (1) the mean of each channel, (2) the mean of each channel scaled linearly with the width of the board, and (3) the maximum of each channel. This produces a total of 3c output values. These layers are used in a global pooling bias structure consisting of:

- Input tensors X (shape $b \times b \times c_X$) and G (shape $b \times b \times c_G$
- A batch normalization layer and ReLu activation applied to G (output shape $b \times b \times c_G$
- A global pooling layer(output shape $3c_G$
- A fully connected layer to c_X outputs (output shape c_X)
- Channel wise addtion with X(output shape $b \times b \times c_X$

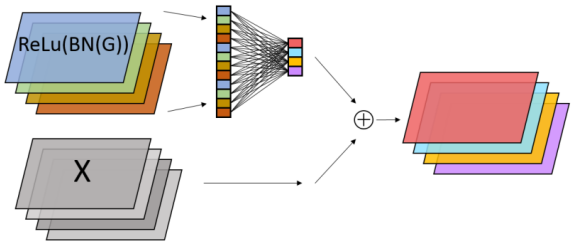


Fig. 3. Global pooling bias structure, globally aggregating values of one set of channels to bias another set of channels.

The illustration of the structure can be seen in Fig. 3.

C. Global Attention Residual Block from NoGoZero+

Models derived from AlphaZero [6] perform well in capturing local information. However, on account of the limited perceptual radius of classical convolution layers applied in the residual blocks, it remains challenging to formulate global strategies. Therefore, an attention-based structure, Global Attention Residual Block, is proposed in NoGoZero+ [1], which can help the model to globally extract important information.

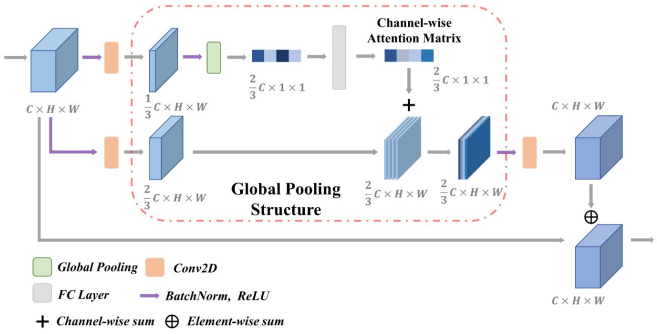
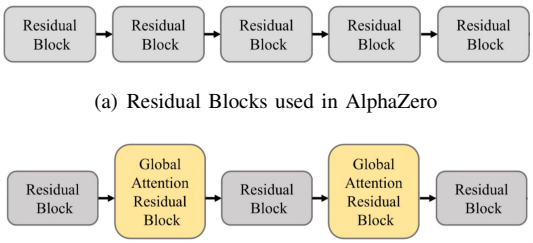


Fig. 4. Global Attention Residual Block, from the paper of NoGoZero+ [1].

GABR inserts in traditional residual block a global pooling layer put forward in KataGo [8], which is also introduced in section B. The detailed architecture is shown in Fig. 4. Given a set of c channels, the input of shape $C \times H \times W$ is split into $X \in R^{C_1 \times H \times W}$ and $G \in R^{C_2 \times H \times W}$. X and G are respectively processed with batch normalization, a ReLU unit and a convolutional layer. The global pooling layer computes the mean and the maximum of each channel of G, thereby producing output with shape $2C_2 \times H \times W$, unlike triple channels in the original global pooling layer. After going through a fully connected layer, G is channelwise added to X. In order to avoid dimensionality reduction, C_1 is set to $\frac{2}{3}C$ and C_2 is set to $\frac{1}{3}C$. The sum of shape $\frac{2}{3}C \times H \times W$ is transformed back into shape of $C \times H \times W$ through a convolutional layer, and added to the input to produce the final output.

Since GARB pays attention to long-range dependencies, classical residual blocks are still essential to extract local information. To make them supplement each other, GARB is inserted between consecutive residual blocks. Hence, the number of residual blocks, no matter classical ones or GARB, has to be odd. An example of 5 residual blocks is shown in Fig. 5.



(a) Residual Blocks used in AlphaZero

(b) Residual Blocks used in NoGoZero+

Fig. 5. Comparison of residual blocks of AlphaZero and NoGoZero+. Take 5 residual blocks as an example, from the paper of NoGoZero+ [1].

D. Boundary Enhancement

By playing with the agents from all mentioned models, we find out that they can hardly handle chess pieces near the boundary of the chessboard, even if some of them boast high winning rates against the baseline.

Therefore, we increase the padding of the first convolutional layer from 1 to 3 on the basis of Global Pooling. Increased padding will fill zeros and extract more features around the boundary of the chessboard, thereby improving the attention of the agent near the boundary.

IV. Visualization

In order to visualize the game, we refer to a github repository [3], using the pygame library for the visualization of the game. All we need to accomplish is the functions that render graphics and handle events.

The visualization of the graphics, i.e. the background, the board, the pieces and the buttons, is done using pygame's draw function. We encapsulate the visualization in a GUI.py file, where we handle the mouse click event "MOUSEBUTTONDOWN" to implement the game interaction.

The main steps to realize the interaction are as follows:

- Creating loops.
  Loop under the While True loop statement to examine the various things that are happening in the game, and interact according to those things
- Mouse click handling.
  Check event.type == MOUSEBUTTONDOWN, if judge the mouse click event happened, also use event.button == 1 to judge whether it is the left mouse button click, only the left button click can get the mouse position. Then judge the position, distinguish whether it is on the board or on the button, if it is on the board, then we need to play the position of the chess to pass to the backend of the game for processing; if it is on the button, then go to the button corresponding to the triggered event can be all right.
- Mouse swipe handling.

TABLE II
THE PERFORMANCE OF COMBINATIONS OF DIFFERENT IMPROVEMENT STRATEGIES.

	1RBs+GP	2RBs+GP	4RBs+GP	PCR+GP
200	10	9	10	10
300	10	10	10	10
400	10	10	10	10
500	10	9	10	10
600	10	9	9	9
700	10	8	10	10
800	9	10	10	8
900	10	9	10	9
1000	10	9	10	10
1100	10	6	7	8
1200	10			9
1300	10			8
1400	10			7
1500	10			
1600	10			
1700	10			
1800	10			
1900	9			
2000	10			
2100	8			
2200	8			
2300	10			
2400	8			
2500	9			
2600	8			
2700	8			
2800	8			
2900	9			
3000	8			
3100	5			

Get the position of the mouse, and if it's on a button then handle the interactive display of the button.

V. EXPERIMENTS

For all mentioned algorithms, we have trained the corresponding models for 1500 batches and tested them. The performances are recorded in Table IV, and we stop testing of models when hitting the first winning rate 7 or below. Besides, several promising algorithms are combined and tested with results recorded in Table II. When testing, we increase the number of simulations of pure MCTS by 100 every time.

The average time of one move of all tested models against pure MCTS algorithm is recorded in Table V.

In Table IV and Table II, we can see that Global Pooling can achieve the best performance among all implemented models. Therefore, Global Pooling is trained for 7000 batches to achieve higher winning rates, and the final results are recorded in Table III.

In all tests, random seed is set as 42.

A. Basic Architecture

The performance of the basic architecture is excellent and stable when playing against pure MCTS with the number of simulations less than or equal to 600. However, after that, we observe a gradual decrease in winning rate when increasing simulations, and the last complete win appears with 1100 simulations. We stop testing when the first 7 occurs with 1200 playouts.

TABLE III
THE PERFORMANCE OF GLOBAL POOLING OF 7000 TRAINING BATCHES.

simulation	200	300	400	500	600	700	800
	10	10	10	10	10	10	10
simulation	900	1000	1100	1200	1300	1400	1500
	9	10	10	10	10	10	10
simulation	1600	1700	1800	1900	2000	2100	2200
	9	10	10	10	9	10	10
simulation	2300	2400	2500	2600	2700	2800	2900
	10	10	9	10	8	10	7
simulation	3000	3100	3200	3300	3400	3500	3600
	10	9	10	8	9	9	10
simulation	3700	3800	3900				
	9	10	6				

B. Residual Block

We can see from Table IV that the effect of using only one residual block has a more obvious improvement compared to the basic architecture, but the effect of using 2 residual blocks (2RBs) and 3 residual blocks (3RBs) is only slightly better than the basic algorithm. However, it is not stable and doesn't boast much improvement when tested against pure MCTS with more simulations. Besides, the effect of using two residual blocks is significantly degraded compared to that of using three residual blocks.

First, that the addition of residual blocks does not weaken the performance can verify that residual network, also known as constant mapping, can perform better than or equal to the shallow networks. Secondly, the optimization effect is not good, though the residual block is originally used in AlphaZero to achieve better results. The main for our unsatisfactory results is that the 9×9 chessboard of the Gomoku game is much simpler in terms of board size and game rules compared to the 19×19 chessboard of Go. Hence, we don't need such a complex network, and a more complex network leads may lower the performance. This can also be seen in the fact that the model with 4 residual blocks (4RBs) works worse than that with 3 residual blocks (3RBs), and the one with 1 residual block (1RB) achieves the best performance.

C. Playout Cap Randomization

The performance of PCR is relatively good when pure MCTS's playout num is less than or equal to 1300, with 8 rounds of 10-0 wins, 3 rounds of 9-1 wins and 1 round of 8-2 win. Compared with the basic algorithm, PCR has a higher winning rate and the high winning rate can be maintained against pure MCTS of more simulations, which means the local exploration improves the ability to predict the situation.

When the number of playouts of pure MCTS is more than 1300, the performance gradually decreases. The winning rate is generally 9-1 or 8-2, or even lower. The reason is that as the opponent's simulation number increases, the benefits

TABLE IV
PERFORMANCE OF ALL TESTED MODELS AGAINST PURE MCTS ALGORITHM.

	basic	1RB	2RBs	3RBs	4RBs	PCR	GP	2RBs+1GARB	3RBs+2GARB
200	10	10	10	10	10	10	10	8	9
300	10	10	9	10	10	10	10	8	4
400	10	10	10	10	10	10	10	3	
500	10	10	9	9	10	10	10		
600	10	10	8	10	10	9	10		
700	9	8	9	10	8	10	10		
800	9	10	10	10	8	10	10		
900	9	9	9	9	9	9	10		
1000	8	10	10	9	10	9	10		
1100	10	8	8	8	8	8	9		
1200	7	10	8	8	9	10	10		
1300		9	9	9	7	10	10		
1400		8	6	7		8	10		
1500		10				10	10		
1600		9				9	10		
1700		8				9	10		
1800		9				8	10		
1900		6				10	9		
2000						8	10		
2100						5	8		
2200							10		
2300							9		
2400							10		
2500							10		
2600							8		
2700							8		
2800							9		
2900							9		
3000							9		
3100							9		
3200							8		
3300							7		

TABLE V
THE AVERAGE TIME OF ONE MOVE OF ALL TESTED MODELS AGAINST PURE MCTS ALGORITHM(/SECONDS). RATIO COMPARES TIME NEEDED FOR ONE MOVE IN OUR MODEL AND THAT NEEDED BY PURE MCTS.

	basic	3RBs	4RBs	PCR	GP	2RBs+GP	4RBs+GP	PCR+GP
model	1.70	1.20	1.30	1.67	1.66	2.01	1.52	1.81
pure MCTS	1.41	1.51	1.50	1.37	1.51	1.55	1.35	1.37
ratio	1.21	0.79	0.87	1.22	1.10	1.30	1.13	1.32

of random local exploration can't compensate for the depth deficiency.

D. Global Pooling

We can see that the performance of Global Pooling is overwhelming compared to other methods. We can make the following judgments. Firstly, Global Pooling can reduce the number of parameters in the computation, which helps to speed up our convergence. Secondly, Global Pooling provides global information. And we know that in this kind of chess game, knowing all the information plays an important role in our decision making. Combining these two reasons, we can fully understand why Global Pooling can achieve the best performance.

In our current game setup, the tendency is that the simpler the network architecture the better for machine learning.

E. Global Attention Residual Block

We have trained GABR with 3 residual blocks and 5 residual blocks, including both typical residual blocks and GARBs. From Table IV, we can see both of them achieve poor performance, unable to win all against pure MCTS with 200 simulations.

The reason of the unsatisfactory results is that the 9×9 chessboard with 4 layer of states doesn't need a complex

neural network. Though the complicated GABR may take more factors into consideration in order to globally improve, it takes much higher risks to fail to fit.

F. Boundary Enhancement

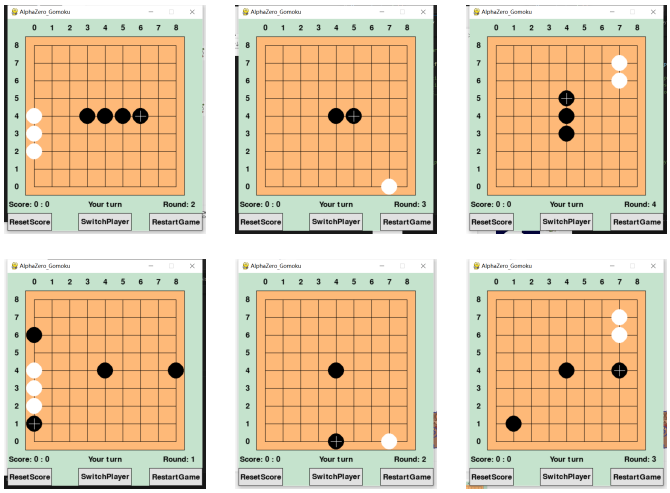


Fig. 6. Three examples of comparison between padding 1 and padding 3. (The first row is of padding one and the second row is of padding 3.)

In both cases, the agent makes an offensive move. Besides, no matter where the agent places its chess pieces, we made the sames move near the boundary in the two settings.

From Fig. 6, we can see that Global Pooling with padding 1 doesn't pay attention to chess pieces near the boundary, and humans can easily win by laying chess pieces on the edges. In contrast, Global Pooling with padding 3 can block potential wining traces near the boundary.

However, when playing against pure MCTS, we observe a dramatic decrease in performance. By increasing padding to 3, the situation is similar to expand the size of the chessboard, which greatly raises the difficulty for the agent to learn. Therefore, a more powerful agent is necessary if we want to tackle the weakness near the boundary while maintaining a high performance.

G. Combination of Different Strategies

From Table IV, we can see that Global Pooling achieves the best performances. Hence, we combine GP with other relatively good algorithms.

However, according to results in Table II, none of these combined models gain better performances than GP. Similar to what is mentioned before, the 9×9 Gomoku problem is a relatively easy one. Therefore, the combinations may exceed the complexity needed for the problem and can not fit.

H. Average Time of One Move

We can observe from Table V that residual blocks and global pooling can decrease the time needed to make a move compared to the basic architecture, while other algorithms will slow down the speed.

The reason for this phenomenon is that residual blocks can raise the processing speed, thus converging to a good policy faster. Global Pooling, which extracts global information, reduces the complexity of making a strategy. PCR's average time of one move is similar to that of basic algorithm, the reason for which is that although random local exploration may intuitively increase search time, the actual results show that the effect is not significant and can even be ignored.

VI. Conclusions

We have carried out wide explorations of Gomoku agent based on AlphaZero, including a basic architecture and further improvements. We implemented AlphaZero, a typical and traditional architecture in various computer chess games. Residual blocks are added handle the problem the vanishing/exploding gradients. In order to fully explore potential nodes, Playout Cap Randomization is applied. We also employed Global Pooling and Global Attention Residual Block, trying to formulate global strategies. Also, we enhanced the performance on the boundary of the chessboard by increasing padding.

Besides, we applied pygame library so as to visualize the Gomoku game.

Global Pooling turns out to perform best among all the implemented models. It can obtain a complete victory against pure MCTS with 3800 simulations.

References

[1] Yifan Gao and Lezhou Wu. Efficiently mastering the game of nogo with deep reinforcement learning supported by domain knowledge. *Electronics*, 10(13):1533, 2021.

[2] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[3] initial h. Alphazero gomoku mpi. https://github.com/initial-h/AlphaZero_Gomoku_MPI. (2023, Dec 29).

[4] junxiaosong. Alphazero gomoku. https://github.com/junxiaosong/AlphaZero_Gomoku. (2023, Nov 20).

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

[6] 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. Mastering chess and shogi by self-play with a general reinforcement learning algorithm, 2017.

[7] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *nature*, 550(7676):354–359, 2017.

[8] David J. Wu. Accelerating self-play learning in go, 2020.